Agda as a Platform for the Development of Verified Railway Interlocking Systems

Karim Kanso

A thesis submitted to Swansea University in candidature for the degree of Doctor of Philosophy

Swansea University
Prifysgol Abertawe

Department of Computer Science
Swansea University

August, 2012

Abstract

This thesis identifies a technological framework that aids the development of verified railway interlocking systems in the Agda theorem prover. The thesis is in two parts, Part I deals with integrating interactive and automated theorem proving in type theory, and Part II addresses verification in the railway domain.

Part I presents a selection of techniques that combine automated and interactive theorem proving paradigms. On the automated side, a novel, type theoretic connection between interactive theorem provers and external theorem provers is theoretically developed and implemented for the interactive theorem prover Agda. Also, Part I evaluates the technique against the current state-of-the-art techniques for integrating interactive and automated theorem provers. The greatest betterment of the techniques is that it can be feasibly applied to larger industrial problems than existing techniques. When exploring problem sets—mathematical and industrial—we obtained promising results.

Two cases studies of the integration have been carried out. These are SAT solving and CTL model-checking. Then CTL model-checking is refined to symbolic model-checking, and subsequently further refined into a customised logic for verifying programs that are definable by decidable Boolean valued transition functions.

The part concludes by exploring, and implementing a more traditional integration. This is where the external theorem prover provides a certificate that Agda checks to be correct, and then converts into a proof-object. A numerical comparison between these implementations is presented.

(continued on next page)

Part II discusses the railway domain and developing verified interlocking systems. This involves applying interactive theorem proving to prove that a selection of signalling principles (lemmata from the railway domain) is sufficient to guarantee high-level safety requirements. This reduces the validation problem by narrowing the gap between the verified statement and the requirements. Then, for a given (concrete) interlocking system programmed using ladder logic, it is shown how to determine, using automated theorem proving, whether it fulfils the required signalling principles. This results in a proof that the interlocking system fulfils the safety requirements. Following the proofs-as-programs paradigm, verified, executable programs are obtained, which we used for simulation purposes.

All work is carried out inside Agda; thus the obtained proofs are tractable. Agda also extracts the verified programs from the proofs. Working within a single tool improves the soundness assurances when compared to the alternative, where there are questions about the correctness of translations between the tools.

This framework has been successfully applied to two different systems: a digital interlocking and a mechanical interlocking.

Declaration

This work has not been previously accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date .....

Statement 1

This thesis is the result of my own investigations, except where otherwise stated. Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date .....

Statement 2

I hereby give my consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date .....

Contents

Acknowledgements

I would like to thank Simon Chadwick, Nick Smith, Peter Duggen and David Johnson from Invensys Rail Group for their time and patience with myself; prompt replies to questions and their guidance throughout the project.

I would also like to thank Mike Dickenson and the other volunteers at Gwili Steam Railway for their kindness in allowing me to use their railway as a case study.

Most importantly I would like to thank my first and second supervisors, Anton Setzer and Faron Moller from Swansea University for their continued guidance, supervision and support throughout the project. My thanks go as well to the rail verification group at Swansea University, especially Markus Roggenbach, Monika Seisenberger, Philip James, Andrew Lawrence and the many other involved students.

I would very much like to thank the examiners, Ulrich Berger and Connor McBride for their effort and high-quality comments that have improved the quality of the thesis.

On a different note, I would like to thank all those that have taken the time to help with my dyslexia over the course of my education. One woman that I owe many thanks to is Jeanne Taylor; a specialist teacher that enjoyed educating children whom had been officially classed as uneducable. Without her support at such an early point in my life, it is doubtful that I would have entered academia.

List of Figures

Chapter 1

Introduction

Verifying the railway domain has been identified to be a *grand challenge* in computer science [Bjø04], it falls under the UKCRC *GC6 Dependable Systems Evolution* grand challenge. Challenges are 'grand' when they are suitably novel and revolutionary, feasible with today's knowledge and technology, require many thousands of man hours to complete, and, of course, challenging. Regardless of the success or failure of a grand challenge, it is the undertaking that is crucial because it promotes research into new areas. The use of grand challenges to focus research has been applied to many disciplines; e.g. mapping the human proteome, putting a man on the moon and unifying the four forces of physics. More information regarding grand challenges in computer science can be found in [Hoa03].

The railway domain is captivating as it is large and encompasses many topics, from the hardware and software, to the organisation and scheduling of staff and trains. The concept of safety within the railway domain is explored throughout this thesis, specifically with respect to a class of control systems known as *interlocking systems*. These systems are responsible for preventing undesirable events from occurring by restricting which commands are sent to the hardware.

This thesis builds-up a type theoretic framework for the development of verified interlocking systems, with the goal of producing verified, executable interlocking systems. First, this entails abstractly defining models of the railway (e.g. tracks, signals, trains), formalising *signalling principles*[1] and proving that the safety requirements follow by these models and principles. Second, interlocking systems are modelled, and for a given interlocking system, it is verified that it fulfils the necessary signalling principles to obtain a proof that the system is safe. This is advantageous as it reduces what must be validated, thus increasing trustworthiness of the verification.

[1] Abstract rules that describe how signals operate safely. See page 3 for more information.

What makes this project interesting is that all the specifications, models and verifications take place within the same framework, namely, the dependently typed theorem prover and programming language Agda[2]. The framework presented has been successfully honed on a sizable modern interlocking system, obtained from the project's sponsor. Once honed, this resulted in the framework being efficient and feasible to use with today's technology.

When verifying large, complex systems, the limiting factor is that vast quantities of large proof obligations arise [WLBF09]. Many of these proof obligations are theorems over finite domains. To prove these obligations by hand, such that the proof can be mechanically checked, requires significant effort, to such an extent that the process becomes unfeasible. However, as these theorems are finite, there are purpose built theorem provers that can determine the validity of these obligations automatically. During an early phase of the project, it became apparent that for the framework to be usable, Agda would need to be connected to external automated theorem provers (ATP). To achieve this, we developed a novel technique to integrate external provers in type theory. The novelty allows a better trade-off between usability, efficiency and soundness when compared to existing techniques, especially for industrial problems sets.

This thesis is structured into two parts: the first discusses integrating external tools into Agda and the second discusses the development of verified interlocking systems.

Introduction Structure. The remainder of the introduction gives the motivation of the thesis, main achievements, and an elaborated overview of the thesis.

1.1 Motivation

Accompanying the first generation of written computer programs, were software bugs[3] [Sha87, Wik12]. The notion of software errors dates back to at least 1843, when Ada Lovelace described the difficulties she experienced writing correct software for Babbage's Analytical Engine [Men43]. These errors were due to a lack of understanding from early on of how to organise uncomplicated commands into a program so that its behaviour is predictable. The study of *computer science* aims to understand these behaviours with mathematical rigour. Similarly, *software engineering* is a discipline that provides

[2] `http://wiki.portal.chalmers.se/agda/`

[3] The name 'bug' was taken from engineering, and referred to a defect in an apparatus or its operation.

a rigorous set of techniques and processes to develop software, one of which is the project *life cycle*.

Traditionally a project's life cycle would follow the well-known iterative *design-develop-test* philosophy. If testing fails, then the project restarts at the design phase. However, this was found to be expensive, and time consuming as erroneous behaviours could only be found at the end of the project. Most pertinent, with respect to critical systems is the issue of incompleteness of the test cases—hence the application of *formal methods* to software projects. These are methods that apply mathematical rigour at various points in the life cycle. These methods range from mathematically specifying and modelling the design of the software, to correctly producing the implementation and verifying that the implementation is correct with respect to the model. There is a significant amount of literature on the subject, and a multitude of orthogonal and complimentary techniques that can be applied at different points in the life cycle.

A plethora of tools have been developed to help apply these methods. One of the longest-established tool sets is the Vienna Development Method (VDM) [Jon90]; it provides a formal language for specifying software. Another landmark tool is the Prototype Verification System (PVS) [ORS92]; it provides an interactive system to model and prove properties about models. The practical use of these tools (and the tools not mentioned) has helped to identify a number of mistakes and shortcomings in specifications and programs [WLBF09].

The application of formal methods to software development raised a new challenge of *validation*. Validation is the process by which it is ascertained that the specification against which the program is verified is in accordance with the requirements. Notably, the specifications and models are determined to be correct with respect to the domain of interest. Validation is crucial for the result of the formal methods to be trustworthy, i.e. a proof corresponds to its intended meaning. Complications arise as validation is typically non-trivial, and it is required to be performed by experts from that domain.

One aim of this work is to simplify the validation process. Here, "simplify" means the amount of information that needs to be validated is reduced. This results in an increased trustworthiness of the formal methods, which in the case of this thesis are used for the verification of safety requirements.

In the railway domain, safety is expressed in the form of *signalling principles*. These are principles that explain, in general, the safe operation of signals. For example, an intuitive signalling principle is that *two opposing signals should not both show green*[4] *at the same time*. These principles are

[4]Correctly, the signals show proceed or danger aspects and not a colour to prevent

heuristics that pervade the whole railway domain. Railway experts developed them over time by experience, sometimes following accidents. They are used while designing, developing and testing the interlocking systems. It is essential that they are correct. It is a straightforward process to determine for a given interlocking system whether it models a given signalling principle, but to determine which signalling principles should be verified in-order to know the interlocking system is safe, requires validation by experts. Even with the expert's validation, there are no assurances that an unconsidered situation could occur, in which a safety requirement is violated.

Remark

It is assumed in this thesis that trains obey the signals. There are safety related devices outside the scope of the thesis that enforce obedience, usually by automatically applying the breaks, to either prevent the train passing the signal at danger or by stopping the train in a buffer zone (overlap) beyond the signal.

Part of the framework defined in this thesis is to provide a mechanism to formalise the signalling principles and high-level safety requirements. Then it is proved that the signalling principles imply the safety requirements. In doing so, the signalling principles might be found to be insufficient or inconsistent. This mitigates the uncertainty of knowing what to validate. The intention is that the safety requirements are formulated at a high-level which is intuitive and amenable to validation. For example, it is proved in Theorem 8.4.2 that if the necessary signalling principles hold, then it is never the case that two trains occupy the same region of the train line.

The second, aim of this work is to explore how to develop and verify a substantial interlocking program in Agda. This aim is further split into two goals:

1. Identify a framework for verified interlocking systems.

2. Provide tool support for this framework in the theorem prover Agda.

By tool support, it is meant to formalise the framework as an Agda library and facilitate the use of external theorem provers for theorems over finite (or finitisable) domains.

In summary, safety on the railway is guaranteed in two parts. First, a direct, formal proof is given that the signalling principles and models imply

ambiguities.

Remark

Martin-Löf type theory [MLS84, NPS90], of which Agda is an implementation, offers a powerful mechanism to construct mathematical formulæ and write functional programs [TD88a, TD88b, NPS90, Ran94, Nor09]; it is essentially typed λ-calculus with the dependent product and algebraic datatypes. By the Curry-Howard correspondence [Cur34, CFCC58, How80], propositions can be represented as types, where an element of the type is a proof of the proposition. Another perspective in type theory is that a type is a specification of a problem such that its elements are programs that satisfy the specification.

In general, this thesis aims to provide the first steps towards developing Agda into a platform to develop and verify substantial software projects. An emphasis is placed on efficiency and usability, so that it becomes feasible to apply this method to extensive case studies.

the safety requirements. Second, automated tools are used to prove that a concrete interlocking system fulfils the signalling principles.

1.1.1 Pelicon Crossing—A Motivating Example

The following is a basic example to illustrate the technique of verifying that a control system is safe. Although the example is not from the railway domain, it is amenable to the same verification techniques. The example used is the **Pe**destrian **Li**ght **Con**trolled (Pelicon) road crossing. These crossings are of British design—although most countries have similar designs that facilitate the local, municipal requirements. (See Section 10.4 for more information and a formal treatment of the Pelicon crossing.) This example was chosen as these crossings are prevalent throughout much of the world, and thus it is presumed to be intuitive for most readers. An example from the railway domain, at this point in the thesis, would require significant explanation, however, the interested reader is directed to Part II of this thesis.

Scenario. A Pelicon crossing has a pedestrian operated button that when pressed indicates that they want to cross the road. There are two sets of lights: one is for pedestrians and the other is for road traffic, see Figure 1.1.

The Pelicon crossing is modelled by 5 areas:

$$\text{Area} := \{\text{T1}, \text{T2}, \text{P1}, \text{P2}, \text{MUX}\}$$

Remark

In the following, some code and other selected details for the Pelicon crossing will be previewed without giving much explanation, this is in order to get a first glimpse at the techniques. The full details will be given later in the thesis.

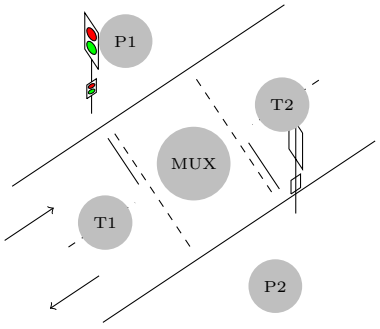


Figure 1.1: Layout of a Pelicon crossing. They consist of two sets of lights, the smaller set for pedestrians and the larger set for road traffic. In this figure, only two aspects are shown for road traffic, but in practice a third aspect for warning the lights are about to change would also present. There is also a pedestrian operated button present, but not depicted. The areas T1 and T2 are for road traffic, P1 and P2 are for pedestrians, and MUX (mutual-exclusion) represents the area of the crossing used by both road traffic (travelling between T1 and T2 through MUX) and pedestrians (travelling between P1 and P2 through MUX). See Section 10.4 for more details.

See Figure 1.1 for the location of these areas. The state of the crossing is abstractly modelled by (1) the number/positions of cars and pedestrians using the crossing, and (2) what aspects the traffic and pedestrian signals display. Formally, for a given discrete time t the state is modelled as follows:

$$\begin{array}{lcl} \text{numbercars}_t & : & \text{Area} \to \mathbb{N} \\ \text{numberpeds}_t & : & \text{Area} \to \mathbb{N} \\ \text{movingcars}_t & : & \text{Area} \to \text{Area} \to \mathbb{N} \\ \text{movingpeds}_t & : & \text{Area} \to \text{Area} \to \mathbb{N} \end{array} \qquad \begin{array}{lcl} \text{traffic}_t & : & \{\text{green}, \text{red}\} \\ \text{pedestrian}_t & : & \{\text{green}, \text{red}\} \end{array}$$

Initially, at time 0 it is assumed that there is no road traffic in, or moving

into MUX; similarly for pedestrians. The initial axioms for road traffic are:

$$\begin{aligned} &\text{numbercars}_0\ \text{MUX} \equiv 0 && (\textit{pelicon-init})\\ &\text{movingcars}_0\ \text{T1}\ \text{MUX} \equiv 0 \\ &\text{movingcars}_0\ \text{T2}\ \text{MUX} \equiv 0 \end{aligned}$$

The axioms for cars travelling between areas T1 and T2 (via MUX) are:

$$\begin{aligned} \text{traffic}_t \equiv \text{red} \rightarrow \text{movingcars}_{(t+1)}\ \text{T1}\ \text{MUX} \equiv 0 \quad & (\textit{taxm1})\\ \wedge\ \text{movingcars}_{(t+1)}\ \text{T2}\ \text{MUX} \equiv 0 \quad & \\ \text{movingcars}_{(t+1)}\ \text{MUX}\ \text{T2} \equiv \text{movingcars}_t\ \text{T1}\ \text{MUX} \quad & (\textit{taxm2})\\ \text{movingcars}_{(t+1)}\ \text{MUX}\ \text{T1} \equiv \text{movingcars}_t\ \text{T2}\ \text{MUX} \quad & (\textit{taxm3})\\ \text{movingcars}_{(t+1)}\ \text{T1}\ \text{MUX} \leq \text{numbercars}_t\ \text{T1} \quad & (\textit{taxm4})\\ \text{movingcars}_{(t+1)}\ \text{T2}\ \text{MUX} \leq \text{numbercars}_t\ \text{T2} \quad & (\textit{taxm5}) \end{aligned}$$

$$\begin{aligned} \text{numbercars}_{(t+1)}\ \text{MUX} \equiv\ & (\text{numbercars}_t\ \text{MUX}) && (\textit{taxm6})\\ & + (\text{movingcars}_{(t+1)}\ \text{T1}\ \text{MUX}) \\ & + (\text{movingcars}_{(t+1)}\ \text{T2}\ \text{MUX}) \\ & \dot{-} (\text{movingcars}_{(t+1)}\ \text{MUX}\ \text{T1}) \\ & \dot{-} (\text{movingcars}_{(t+1)}\ \text{MUX}\ \text{T2}) \end{aligned}$$

where $n \dot{-} m = \max(n - m, 0)$. The axioms for the pedestrians travelling between areas P1 and P2 (via MUX) are symmetric to the above 6 axioms but not presented, i.e. substitute T1 for P1, T2 for P2, numbercars for numberpeds and movingcars for movingpeds. Implicit in the axioms, especially axiom (*taxm6*), are the well-formdness conditions: *cars are never in P1 or P2*, and *cars do not travel directly between T1 and T2*. There are symmetric well-formedness conditions for pedestrians.

Safety Requirements. In this setting, the high-level safety requirement—which still remains to be validated (along with the other axioms) by domain experts—is that, *at any point in time, exclusive use of the crossing is given to pedestrians or to road traffic*. The requirement is formalised as:

pelicon-safety

$$\forall t\ .\ \text{numbercars}_t\ \text{MUX} \equiv 0 \vee \text{numberpeds}_t\ \text{MUX} \equiv 0$$

This is the desired property that we wish to prove.

In the following, *safety principles* are lemmata which all together imply the safety requirements. A *safety principle* is an intermediate lemma, typically deduced from principles, rules, or standards in the target domain. *Safety conditions* are concrete formulæ (formulæ where the atomic propositions are variables in a concrete program), which reduce (e.g. via induction) to formulæ over finite domains that are provable by ATP, and imply the safety principles. For example, in the railway domain there is a large amount of literature detailing various *signalling principles*, which in this thesis are synonymous with safety principles.

A safety principle which implies this safety requirement is that, *at any point in time, only road traffic or pedestrians are allowed to enter the crossing, but not both*, and is formalised as follows:

pelicon-principle

$$\forall t \,.\, \big(\text{movingcars}_t\ \text{T1}\ \text{MUX} \equiv 0 \wedge \text{movingcars}_t\ \text{T2}\ \text{MUX} \equiv 0\big) \\ \vee \big(\text{movingpeds}_t\ \text{P1}\ \text{MUX} \equiv 0 \wedge \text{movingpeds}_t\ \text{P2}\ \text{MUX} \equiv 0\big)$$

For a given Pelicon control system, giving a direct proof of the safety principle or safety requirement would be a cumbersome activity as ATP tools do not typically yield abstract solutions, but concrete solutions for concrete problems. Our solution is to use the ATP tool to give a concrete proof of the following safety condition:

pelicon-condition

$$\forall t \,.\, \text{traffic}_t \equiv \text{red} \vee \text{pedestrian}_t \equiv \text{red}$$

This condition has the semantics: *for all time, pedestrians have a red light or road traffic has a red light*. In this setting the safety condition can be viewed as signalling principle, however, in practice, signalling principles are more complicated. It is provable that this implies the safety principle (and in turn the safety requirement). It should be noted that the safety condition is representable in terms of a concrete control system's output (and possibly input/internal) variables, whereas the safety principle is an abstract formalisation.

We have shown in Agda that the stated safety condition implies the safety principle which in turn implies the safety requirement. We have also shown in Agda (using ATP) that a standard implementation of the Pelicon crossing in Figure 1.2 implies this safety condition, via an inductive proof. Note that the variables `tlight.g` and `plight.g` in the ladder program at time t are mapped to the abstract variables traffic_t and pedestrian_t, respectively.

Therefore, we have shown in Agda using a combination of ITP and ATP that the implementation is safe up to the validation of the models. Using the fact that Agda not only type-checks, but also compiles programs, we used the compiler to extract a fully verified, executable Pelicon crossing control system.

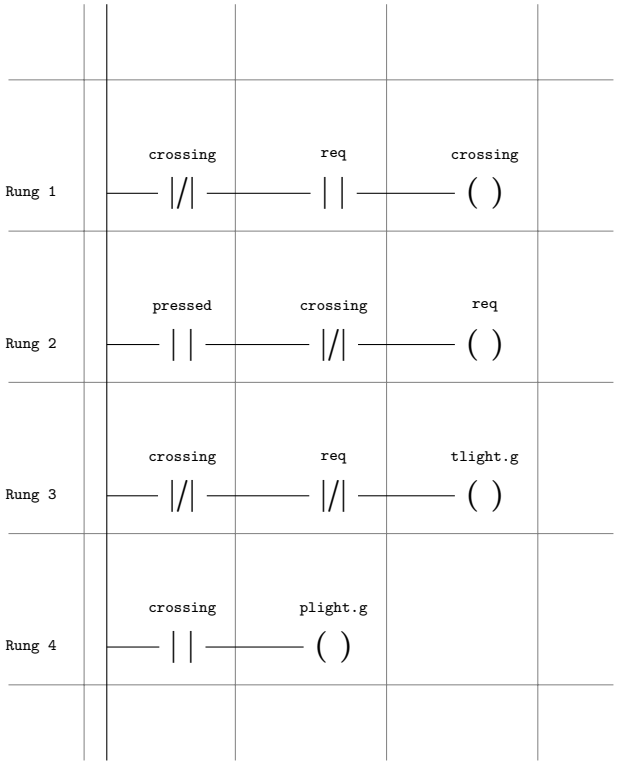


Figure 1.2: Pelicon Ladder. Here, −| |−, −|/|−, and −( ), mean holds, does not hold, and becomes true, respectively. E.g. rung 1 is formalised as: crossing := ¬ crossing ∧ req. See Section 9.2 (page 191) for a full explanation of the semantics of this diagram.

1.2 Main Achievements

The following itemises the main achievements of this thesis. Continuing in the two part spirit of the thesis, the achievements are presented for each part separately. In the next section, background information is provided, and the thesis is summarised.

Part I – Integrating Automated tools into Agda:

- New integration of ATP into ITP.
  - Implemented in Agda.
  - Feasible to use for industrial verification problems.
  - Numerical evaluation.
  - Mitigated risk of executing malicious ATP programs.
- Implementation of a traditional approach.
  - Numerical evaluation, and compared with previous approach.
- Consistency analysis of built-in mechanism in Agda.
- Integration of SAT solving and CTL model-checking.

Part II – Verification in the Railway Domain:

- Formalisation of two real world interlocking systems.
  - Modern digital interlocking programmed using ladder logic.
  - Historic mechanical interlocking system.
- A two-step approach to verify the domain safety of interlocking systems.
  - First step reduces the validation problem.
  - Signalling principles are shown to imply safety requirements.
  - Interlocking systems are shown to fulfil signalling principles.
  - Fully tractable industrial scale proofs in Agda.
- Produced fully verified and executable interlocking systems.
- Analysed the completeness of mechanical interlocking systems.

1.3 Background and Overview

In this section, a brief overview of the thesis is presented along with the corresponding chapters/sections which elaborate the discussions. Continuing with the two part nature of this thesis, first, chapters integrating ATP into ITP are summarised to describe the theoretical basis of the work, and then chapters discussing verification in the railway domain are summarised to describe application of the theory to real world problems.

1.3.1 Part I – Theorem Provers

Before summarising Part I, background information relating to automated and interactive theorem provers is presented.

Theorem proving tools can be placed into one of two categories: interactive or automatic [Bou97]. The first category, ATP tools, attempt to prove a theorem by automatically deducing the proof from already proved lemmata. In some cases, intermediate lemmata are introduced and proved automatically. The user has no direct influence over the derivation and proving process. In this work, only ATP tools that coincide with decision procedures for logics are considered as they are valid intuitionistically; examples are SAT solving and model-checking. Conversely, the second category is formed by ITP tools, i.e. proof assistants or proof checkers; they work by allowing the user to guide the derivations and proofs of lemmata, culminating in a proof of the desired theorem.

ATP Tools

These tools are very powerful when dealing with concrete theorems over finite domains as in SAT and finitisable domains as with temporal logics. In some cases ATP tools can be applied to theorems over infinite domains, as in SMT [BSST09] and first-order provers, but this class of tools typically become semi-decidable decision procedures [dMB09] and are not considered in this work. Industrial hardware and software verification is archetypal of finite concrete theorems—large but not inherently complex problems[5]. ATP tools often allow the system to be modelled using an intuitive language, and the desired properties of the system to be specified in the tool's logic. The tool will attempt to prove these properties. When this is not possible the tool will either provide a counterexample of the property or declare an unknown result. These unknown results typically occur as a result of attempting to prove a theorem that has an infinite component and not knowing which lemma to apply[6] or when a resource (time or space) is not sufficient to complete the proof.

ITP Tools

These tools are powerful when dealing with theorems over infinite domains, which are not known, at least currently, how to mechanise (automate) their proofs. To clarify the purpose of ITP tools, consider a theorem of the form

[5] We refer to this type of verification as industrial.

[6] Note that these semi-decidable theorem provers are not considered in this thesis.

$\forall n.\varphi$ in which φ has an infinite component. It could be possible to prove it using standard induction, but often the theorem needs to be strengthened such that a new theorem $\forall n.\varphi \wedge \psi$ implies the desired theorem. The choice of this strengthened theorem, in general requires input from a human being. The reason is that when proving the inductive step, $\varphi(n)$ might not be sufficient to imply $\varphi(n+1)$; whereas a stronger theorem $\varphi(n) \wedge \psi(n)$ might be sufficient. In general, choosing ψ such that it is strong enough to allow the theorem to be proved, without being so strong that it hinders the proof is a complex task, and cannot be mechanised. ITP tools have the advantage that unknown results do not occur as the user guides the proof and the tool checks that the proof is correct. There is a useful body of work relating to automating inductive proofs, however, the techniques are inevitably incomplete [BM90, Bun99].

Limitation of Exclusive use of ATP or ITP

As indicated, each class of theorem prover has advantages and disadvantages. The following exemplifies why only using one class of theorem prover is not adequate for industrial applications.

Consider a simple reactive system realised using Boolean valued equations which determine the next state from the current state and input variables. The most natural way to verify such a system is using a SAT based verification. Assume a safety property[7] P. One would have to create the propositional formula which expresses that P holds in all reachable states, i.e. $\forall \textit{reachable state } s.P(s)$. For small state spaces, it is possible to enumerate all states and take their conjunction, but, for realistic systems, this is not feasible[8]. Thus, the fastest method to determine whether the system models P is to apply induction. This would yield two proof obligations which take into account reachable states, namely the base case (initial state) and inductive step (transition function). After the user has entered these into a SAT solver and determined the validity of both cases, there is still a meta-step to be performed by the user. The meta-step here is to prove the validity of induction outside the SAT solver. The user can then assemble the three proofs to verify that P always holds in all reachable states. SAT solving alone is not sufficient to prove this theorem.

This example, perhaps contrived, shows the limitation of ATP alone, and in general this final task of assembling proofs is more complicated. See Sec-

[7] A safety property is a property that must hold in all reachable states.

[8] Consider a system with 600 state variables (not uncommon within industrial applications), there would be 2^{600} conjuncts which is larger than the number of atoms in the observable universe.

tion 10.4 and Chapter 11 for substantial examples where various ATP proofs are assembled to show that the Pelicon crossing, and a railway interlocking system are safe, respectively.

When using ITP tools on large, finite, concrete, theorems the work delegated to the user is exorbitant [JGB10]. Industrial verification is a particular case where large numbers of mechanisable, and a small number of un-mechanisable proof obligations arise [WLBF09]. Thus, it would be beneficial to use ATP tools for the mechanisable proof obligations and ITP for the un-mechanisable ones, and it will be shown how this can be achieved for Agda.

Agda

Agda[9] belongs to a family of theorem proving tools based on intuitionistic type theory developed by the Swedish logician Martin-Löf. The first of the family was called *Another Logical Framework* (Alf) [ACN90], developed in 1992. Alf was chronologically proceeded by Half, CHalf, Agda and Alfa. Recently in 2007 Ulf Norell at Chalmers University started the current implementation of Agda [Nor09]. Although it is the second version, Agda 2 is known simply as Agda, and the original Agda is known as Agda 1.

Agda is a dependently typed (λ-calculus with the dependent product and algebraic data-types) functional programming language and interactive proof assistant. This work will extend Agda into a platform for specifying, developing and verifying railway interlocking systems. The TYPES group at Swansea University is involved with the development of Agda which provides an ideal opportunity to extend Agda for our purposes. For information about how Agda was extended see Chapter 5.

Part I Overview – Integrating ATP Tools into ITP

Over the past 30 years, there have been many attempts to combine theorem proving tools (see Section 2.3.1 for a discussion of these methods). Briefly these are: Oracle, Reflection, Certificate. The use of an ATP tool as an *oracle* is when the ATP tool determines the validity of a theorem as: yes, no, maybe, or unknown. *Reflection* is when an ATP is formalised and proved to be correct in the ITP, and then a verified ATP tool is extracted. The final approach is when the ATP tool provides *certificates* or justifications that are then checked to be correct in the ITP. This last approach is widely regarded as the state-of-the-art with respect to integrating ATP's into ITP's.

[9] See: `http://wiki.portal.chalmers.se/agda/`

The approach formalised in Chapter 3 does not fit into these categories; it is a mixture of the first two methods, i.e. combining an oracle with reflection. This entails writing a decision procedure for a logic in Agda, proving its correctness in Agda, and then overriding it for closed terms by an external ATP. At first glance this is cheating as the correctness proof used does not refer to the ATP tool, however, this approach is sound as the ATP tool should coincide with the Agda implementation. The soundness of the technique is analysed in Section 3.2.1, and the technical implementation details of this approach are contained in Chapter 5. The implementation of the technique makes use of built-in functions in Agda; these are functions which have their implementation overridden for closed terms. To determine whether the technique is sound, it was required to formalise these functions, see Section 5.1 for an analysis of the consistency of built-in functions. In Section 5.5 it is considered how to prevent this integration from being used maliciously to execute undesirable programs. For example, consider the situation where an Agda module is obtained from the Internet, and then when type-checking it, a malicious program is executed.

In Chapter 4, two principal case studies were selected to evaluate the integration defined in Chapter 3, these are SAT solving (cf. Section 4.1) and CTL model-checking (cf. Section 4.2). These theories were selected as they are fundamental for industrial verification. CTL model-checking was refined into symbolic model-checking (cf. Section 4.3), and then further refined in Section 9.4.1 to a custom logic for systems realised by Boolean valued transition functions.

The certificate method identified above is implemented in Chapter 6. This implementation is orthogonal to the previous technique in Chapter 3. It allows for a very high-level of soundness assurances. These assurances are that the external tool provides a trace of the proof of a theorem, and then the trace is type-checked by Agda. The efficiency, and soundness of this integration are evaluated against the previous technique. The result of the analysis shows that for industrial problems the first integration performs better, however, when the obtained certificates are not too complicated, the second method performs better.

1.3.2 Part II – Railway Domain

The railway domain [Noc02, Lea03, KR01] is vast. In this subsection, a brief background of the relevant parts of the domain is presented. This includes an introduction to the modelling of the topology and interlocking systems. For more information, there are in line references to the corresponding chapter/section.

Remark

See Appendix A for glossary of relevant railway terminology.

Systems in the railway domain are archetypal of industrial problems; a good understanding of these problem sets will benefit many different communities. In our case, it will aid the formal development and verification of critical systems. Regarding the railway domain and formal methods, there are the following correspondences:

Signature	≡	Topological layout of railway network
Specification	≡	Operational constraints of a topology
Model	≡	Computational model of an interlocking system
Implementation	≡	Instance of the model, i.e. an interlocking system

Previously, in the author's masters thesis [Kan08, KMS09] an interlocking verification process was developed. It applied SAT solver technology to verify (finite domain) first-order theories with respect to safety. This thesis builds upon that previous work by not only considering the verification (Chapter 10), but also how to prove that the verification implies the safety requirements. This requires a formalisation of the signatures, specifications, models and implementation. See Chapter 8 and Chapter 9 for these formalisations, respectively. Then, in Chapter 8 using these formalisations, abstract theorems in the railway domain that relate the verification to the high-level safety requirements are proved. Chapter 11 contains an elaborated example of the whole process.

In the following sub-sections, these four correspondences are considered in detail.

Topology

The topology of a Railway is concerned with how the physical "nuts and bolts" of the railway network are configured. Briefly this consists of track segments, connections between these segments and various types of signals. Track segments are atomic components of the railway network.

The following diagram depicts common types of track segments. Track segments are composed to form the railway lines, and then signals can be placed between connected segments.

With just using the 'linear' and 'set of points' segments it is possible to create complicated track-plans, i.e. the 'set of points' segments allow for diverging and converging lines.

Remark

In practice, although not considered in this work, the topology also entails civil aspects of the railway network. This includes: bridges, tunnels and level crossings.

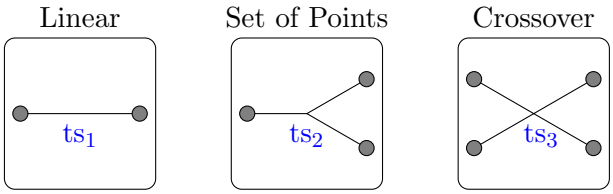


Figure 1.3: Common Track Segments.

The following track-plan is based on an existing station that was modelled during this work (cf. Section 10.5). It depicts a terminal station with two platforms. Trains arrive and leave on the right-hand side. The arrows (on the right) indicate the direction of travel on the lines. The lines are subdivided by hatch marks into track segments. There are four sets of points that allow trains to transition between the top line and the bottom line. Signals are placed between some of the segments. Note the orientation of the signal relates to the direction that they are must be obeyed. For more information about the topology, and the modelling see Chapter 8.

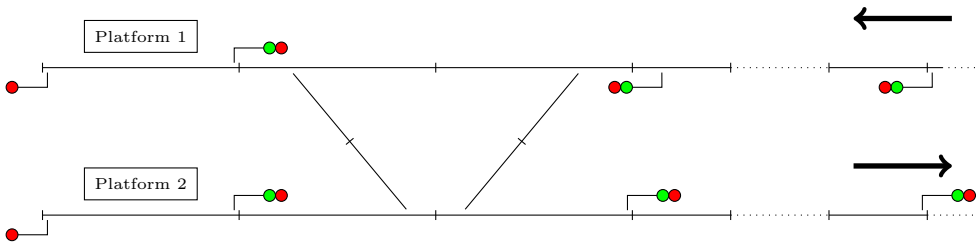


Figure 1.4: Example railway network. The leftmost signals are fixed at danger and are not safety controlled, or modelled in this thesis, they are only for illustrative purposes.

Operational Constraints and Safety Requirements

Within standard railway design and development, the functionality of the topology is specified by documents known as control tables. In their simplest form, control tables list train routes. A route starts at a signal and terminates at the proceeding signal on the line. All the track segments that a route spans are specified; if one of these segments has multiple configurations, such as a set of points, then the required configuration is also specified. The tables also specify which signal aspect to display (e.g. danger, proceed, or proceed with caution) and under what circumstances, which in turn (by the signalling scheme) specifies the speed limit of the line ahead of the signal. In the case of a danger/red aspect, the speed limit is constrained to zero. See Chapter 8 for a formalisation of these tables.

Thus, these control tables form specifications for the railway control systems. They are formal documents that are signed-off early in a project's life cycle, and then used to develop the control systems and derive test-cases (among other things). It should be noted that the safety of the railway is delegated to these tables.

From a control table, it is possible to determine incompatible route combinations. For example, any two routes that contain the same track segment are incompatible. This notion of incompatible routes forms a basis of the *signalling principles*.

Signalling principles are general rules that the interlocking system must fulfil. In the UK, each railway line requires a slight variation of these rules. This is for historical reasons as most of the train lines were originally built by and operated by private companies that formed their own standards in lieu of standardisation from government. For information regarding the history of British railway signalling, see Section 9.1. In this thesis, signalling principles are represented by Agda formulæ that formalise these rules. From experience, they are expressible as first-order formulæ that quantify over the components of the topology and routes. Each of these components has a number of associated atomic propositions which represent attributes of the components. The signalling principles define relations between these atomic propositions. For instance, some of these atomic propositions express whether a track segment is occupied, or not, and others relate to the positions of sets of points and signal aspects.

As previously described, an abstract safety requirement is required. Typically it is also a first-order formula. The abstract safety requirements for the railway domain considered in this thesis are given as follows:

- trains do not collide, and

- trains do not derail (see Section 4 of [HP00]).

This safety requirement is intuitive. However, proving that it holds for a given interlocking system, and railyard is non-trivial. In this thesis, the proof is carried out in two parts. First, an abstract proof that the safety follows by the signalling principles is performed once for each set of signalling principles. Then it is proved that the interlocking system fulfils the necessary signalling principles to guarantee the safety. See Section 8.4 for a detailed discussion and an example of the abstract proof.

Computational Model and Implementation

Before formal verification of an interlocking system, the system must first be formalised. This requires a computational model of the interlocking. The model includes a language (syntax and semantics) used to program the interlocking and relevant hardware. The hardware that needs to be formalised depends on the interlocking design. For example, timers are part of the hardware, but not necessarily directly supported by the language.

The interlocking systems explored in this thesis are realised using ladder logic, a graphical representation of a Boolean circuit, specially tailored for reactive systems, see Figure 1.2 for an example. This low-level language of Boolean equations is particularly amenable to off-the-shelf verification without much effort. See Chapter 9 for a formalisation of ladder logic programs.

The actual formalisation of the implementation of an interlocking system is then an instance of this model. For the verification to be valid, these models are required to define a decidable transition system, see Section 9.3. For now it is noted that a well-formed ladder logic program produces a decidable transition system.

Verification

The underlying technology used is a SAT solver[10], so proving that the interlocking system fulfils the required signalling principles (first-order theorems) requires translating the theorem into an equivalent propositional theorem. This is possible as the signalling principles are built over finite sets of signals, track segments and routes, i.e. there are no infinite topologies. The verification requires a link between the abstract world of the topology-model and control tables, to the concrete world of the interlocking. This link specifies the meaning of the variables in the interlocking system; that is which

[10]CTL model-checking is also explored, but for efficiency reasons the use of SAT solving is predominant in this thesis.

variables control which pieces of hardware. See Chapter 10 for a technical discussion of the actual verification, this includes defining correctness of a ladder program and executing an external SAT solver.

It is also possible to determine whether the interlocking system correctly refines a control table. This is a straightforward process that entails translating each entry in the control table into a verification condition. Provided the link between the abstract models and the interlocking is complete, then the translation is canonical. See Section 10.2 for information relating to control table verification.

Thus, two types of verification are performed; the first verifying that the interlocking system is safe, and the second verifying that the interlocking system fulfils its specification.

Part II Overview – Railway Domain

Starting in Chapter 8 the railway domain is modelled. This includes formalising the topology and control tables, and then signalling principles and abstract safety requirements are formulated. In Section 8.4.1, it is proved that the safety requirements follow by the signalling principles.

Chapter 9 formalises interlocking systems defined by ladder logic programs. The chapter starts by discussing the history and motivation of British railway signalling, including a summary of significant accidents, such as the worst railway accident in British history: Armagh, 1889, where many school children died in a collision. The remainder of the chapter formalises ladder logic programs and provides a translation from ladder logic programs into decidable transition systems. These decidable transition systems are required to relate verified concrete statements (about a concrete interlocking) to a corresponding statement in the abstract models.

The two previous chapters introduced abstract models of railways, and interlocking systems. In Chapter 10, SAT based safety verification of interlocking systems is discussed. Briefly this involves creating a data-type of reachable states, and then formulating the safety properties as *for all reachable states ... holds*. Validity of these conditions is determined by the use of an efficient SAT solver; typically, this is achieved by reducing the safety conditions, by induction, to a base case and inductive step.

Chapter 10 also details the toy example of the Pelicon crossing, this includes producing a verified implementation that is simulated. Also, a number of details about the industrial interlocking system that was verified are presented.

Finally, in Chapter 11 the full case study of Gwili Steam Railway is presented. The case study follows the techniques of the last three chapters. The

topology and control table are modelled, and the interlocking system is modelled, and then it is verified that it fulfils the relevant signalling principles.

The interlocking system at Gwili is of a mechanical design; thus it was required to translate from mechanical interlocking systems into ladder logic programs. This translation is elucidating as it clearly shows the weakness of the mechanical interlocking systems, and why digital interlocking systems are more powerful. The reverse of the translation, where ladder logic programs are encoded into mechanical interlocking systems is considered in Appendix B.

1.4 Overview of Results

The main result of the thesis is that fully verified, executable, industrial scale interlocking systems are produced. This is a fully worked out, practical illustration of the *proofs-as-programs* paradigm.

Furthermore, from Part I, the main result is a new technique of integrating ATP into ITP. This technique offers a better trade-off between soundness, efficiency and usability than existing techniques when used for industrial problem sets. This technique is evaluated on mathematical problem sets in Section 5.4.1, and compared to a traditional, state-of-the-art technique in Section 6.3.2. The evaluation clearly shows that the size and complexities of the prime formulæ theorems that Agda can feasibly solve is increased by a factor of 10–100, depending on the complexities of the problem set, than was possible without the integration. However, the observed performance is significantly worse than manually executing the solver outside of Agda, in part this is due to inefficiencies in Agda (and the GHC run-time system).

The interface was also evaluated on industrial verification problem sets of interlocking systems, see Section 10.5.1. It was successfully applied to a ladder logic program with approximately 300 rungs and 250 inputs variables, which when unfolded for the inductive step contained approximately 1500 propositional variables. This shows that it is feasible to use Agda as a formal development platform for verified software. The traditional interface that was also evaluated could not successfully determine the validity of the industrial problem sets due to inefficiencies.

The main result for Part II is the feasible technique of the two-step verification procedure for railway interlocking systems. This is where, as a first step, the signalling principles are proved to imply the safety requirements, and then as a second step, a concrete interlocking system is shown to fulfil the signalling principles, thus a tractable proof showing that the system fulfils the safety requirements is obtained.

This technique was successfully applied to two interlocking systems. This entailed formalising the topologies and interlocking systems, proving that the safety requirements follow from the signalling principles, and verifying that the interlocking systems fulfil the signalling principles. These verified interlocking systems were also compiled and executed, this allowed the verified interlocking system to be simulated. See Appendix C for a simulation of the Pelicon crossing example. In Chapter 11, a fully worked out example of the verification of Gwili Steam Railway is presented. Also see Section 10.5 for information relating to the verification of a modern interlocking system, although only partial results are presented for this interlocking system to protect the project sponsor, and not publish information that could potentially be maliciously used to subvert safety.

The results of this thesis clearly show that the techniques which were applied, facilitate feasible verification of substantial software projects within a type theoretic setting. This is a respectable result as practical implementations of type theoretic tools will often result in exceeding large terms for large concrete data structures, to such an extent that they become unfeasible to use for industrial applications.

1.5 Notations Used

Remark

The two parts of this thesis are independent, however, the chapters in each part are intended to be read sequentially.

This thesis follows standard mathematical syntax, with a few deviations. In this section the notations used are explained. Much of the notation is close to Agda notation. More information about Agda notation can be found in Ulf Norell's thesis [Nor07].

For a complete understanding of the formulæ provided in this thesis, well-known inductive definitions of natural numbers, vectors and existential quantification are provided in Agda-like syntax; also a summary of other language constructs is provided. The following notations are used:

Set For historic reasons, this is what is meant by type in traditional programming languages. So all small types (e.g. Booleans, natural numbers) are elements of Set. As Agda uses type hierarchies, large types

are elements of Set_1; in this work higher levels than Set_1 are not needed.

a* : *A A typing judgement that a has type A.

_ Underscores are used for two purposes, first, as placeholders for mix-fix definitions. Secondly, they are expressions, such that they indicate that the expression they are in-situ of is inferable by the type-checker.

∀*x* . *B*

(*x* : *A*) → *B* Both of these notations introduce the dependent function (Π) type. Elements of this type are functions, which when applied to an argument $a : A$ yield an element of type B with all occurrences of x substituted by a. In general, $\forall x$ is syntax for $(x : \_)$.

∀{*x*} . *B*

{*x* : *A*} → *B* Both of these notations introduce the hidden function (Π) type. Hidden parameters are used when they are inferable by the type-checker. In general, $\forall\{x\}$ is syntax for $\{x : \_\}$.

{*x*} Applies a hidden parameter to a function with a hidden argument. This is required when the hidden parameter is not inferable by the type-checker. For instance, assume a function $f : \{x : A\} \; . \; B \; x$ and $y : A$, then $f \; \{y\} : B \; y$.

1.5.1 Definitions

The majority of definitions in this thesis are formulated using Agda syntax, albeit with a small number of improvements. For example, sub and super script parameters are used liberally. In the following, the basic syntax of inductively defined data-types is introduced. To define the set

$$\{\mathrm{red}, \mathrm{blue}, \mathrm{orange}\}$$

in Agda, the following code is declared:

```
data Colour : Set where
  red blue orange : Colour
```

Here, Colour is defined to be an element of Set; with red, blue and orange defined as elements of Colour.

Natural numbers. However, to define more complex sets such as the natural numbers (without dependent types), the following code is declared:

$$\begin{aligned}&\text{data } \mathbb{N} : \text{Set where}\\ &\quad \text{zero} \;:\; \mathbb{N}\\ &\quad \text{suc} \;:\; \mathbb{N} \to \mathbb{N}\end{aligned}$$

Here, $\mathbb{N}$ is defined to be an element of Set; $\mathbb{N}$ is inductively defined as the least set containing zero and closed under suc.

Remark

Every data-type in Agda corresponds to a standard mathematical freely-generated (co)inductively defined set, i.e. least (or greatest) set closed under the constructors.

Vectors of type A of length n. The definition of Vec introduces a dependent type, depending on an arbitrary set A and a natural number.

$$\begin{aligned}&\text{data Vec } (A : \text{Set}) : \mathbb{N} \to \text{Set where}\\ &\quad [] \;:\; \text{Vec } A \text{ zero}\\ &\quad \_::\_ \;:\; A \to \text{Vec } A\, n \to \text{Vec } A\, (\text{suc } n)\end{aligned}$$

Vec is inductively defined using two constructors: [] constructs the vector of type A of length 0 and the mixfix $\_::\_$ concatenation of an element of type A to a vector of type A of length n constructs an element of $\text{Vec } A\ (\text{suc } n)$. The underscores denote the positions of the arguments and are not part of the constructors name.

In the definition of Vec two different flavours of dependent types are used. The first flavour is the named parameter A; it is fixed throughout the definition of Vec. Both constructors refer to A and the resultant type depends on it. Secondly, $\text{Vec } A : \mathbb{N} \to \text{Set}$ is indexed by elements of $\mathbb{N}$ where the arguments and resultant type can refer to different indices.

In the definition of $\_::\_$, a hidden argument is used, namely, n. Hidden arguments can be omitted when the type-checker can infer its value. In this case, n is inferred from the context and/or the second parameter. To clarify

the situation of hidden arguments, the definition of vectors is precisely given as follows:

$$\begin{array}{l}\text{data Vec } (A : \text{Set}) : \mathbb{N} \to \text{Set where} \\ \quad [] \;\; : \text{Vec } A \text{ zero} \\ \quad \_::\_ \;\; : \; A \to \{n : \mathbb{N}\} \to \text{Vec } A\, n \to \text{Vec } A \,(\text{suc } n)\end{array}$$

Here the curly braces denote hidden arguments. If a hidden argument is to be given explicitly, to help the type-checker when it cannot be inferred automatically, curly braces are also used. So a hidden parameter would be specified by: $\{n\}$. For example, assume an $a : A$ and $v : \text{Vec } A\, n$ for some n, then the hidden parameter in the constructor $\_::\_$ as follows:

$$\_::\_\; a\; \{n\}\; v$$

Functions. In Agda, functions are typically defined by case-distinction, using a *Haskell-like* syntax, although with significant differences. As an example of an inductive function definition in Agda, consider the Agda code of the truth (or atom) function:

$$\begin{array}{l}\text{T} : \text{Bool} \to \text{Set} \\ \text{T true} \;\; = \; \top \\ \text{T false} \;\; = \; \bot\end{array}$$

The above definition is by case-distinction on the set of Booleans. Agda requires that the case-distinction is total, that is, the function fulfils the inductive elimination principle of the types in question (above it is Bool). It is also required that the definition passes a termination check, see below.

Recursion Inductive data-types allow for recursive function definitions. In the case of natural numbers, a recursive function is defined by the use of pattern matching:

$$\begin{array}{l}\text{double} : \mathbb{N} \to \mathbb{N} \\ \text{double zero} \quad\;\; = \; \text{zero} \\ \text{double } (\text{suc } n) \;\; = \; \text{suc } (\text{suc } (\text{double } n))\end{array}$$

Agda requires and checks that a function terminates. The termination checker follows certain termination principles based upon primitive recursion in higher types [Abe98].

Records. Agda provides support for grouping values together, similar to a tuple in other languages or a data-type with one constructor. This support is by the record type. Consider the following example which pairs two natural numbers, and defines the constructor pair.

```
record ℕPair : Set where
  field
    A : ℕ
    B : ℕ
  constructor
    pair
```

Records are equipped with projections, and as it is known that they only have one constructor they will unfold. Thus, they are simpler to use as case-distinctions are not required to unfold. From the above definition two projections are obtained, one for each field.

$$\mathbb{N}\text{Pair.A} : \mathbb{N}\text{Pair} \to \mathbb{N}$$
$$\mathbb{N}\text{Pair.B} : \mathbb{N}\text{Pair} \to \mathbb{N}$$

An improved notation is used for the projections: assume $x : \mathbb{N}\text{Pair}$, the values of the fields are identified by A_x and B_x.

Remark

It is allowed for the fields to be dependently typed.

Moreover, elements of record types are given in one of two ways: First, by the use of the constructor (if defined, as it is optional), and secondly by a special record syntax. In the case of ℕPair, assume two naturals n and m, the first method is as follows

$$x = \text{pair } n\ m$$

and secondly, when no constructor has been defined:

$$x = \text{record } \{\ \text{A} = n\ ;\ \text{B} = m\ \}$$

1.5.2 Formulæ

Implicitly, the mathematical formulæ in this thesis are within the model of Agda unless explicitly stated otherwise. These formulæ are presented

Remark

In this thesis, every proof has been carried out in Agda and corresponds to a theorem in standard mathematics—continuous functions are not used.

The proofs of theorems proved in Agda are presented in a standard mathematical way, albeit, in significant detail. This is to provide an intuition of the proofs in Agda. Also, well-known mathematical theorems are proved, for example, the pigeonhole principle.

using standard mathematical notation with a few exceptions where it is more intuitive to use Agda notation.

In Agda formulæ are represented as sets. Following the BHK interpretation [TD88a] of the logical connectives, a proof of a formula is an element of its set. That is, true is represented as the singleton set $\top$, with the canonical element $\mathtt{tt} : \top$, falsehood is represented as the empty set $\bot$. Conjunction of two formulæ φ and ψ is given as a pair (p, q) such that $p : \varphi$ and $q : \psi$. Disjunction of two formulæ φ and ψ is represented by a tagged union, i.e. $\mathrm{inj}_1\ p$ or $\mathrm{inj}_2\ q$, where $p : \varphi$ and $p : \psi$. Universal quantification is represented by the dependent function type. A standard universally quantified formula of the form

$$\forall x \forall y \forall z\ \varphi$$

is written as

$$\forall x\ y\ z\ .\ \varphi$$

to be close to the Agda representation.

The final connective is existential quantification. However, before introducing it, relations are considered.

Relations are given as functions that define sets. That is, in standard mathematics the relation

$$R \subseteq A \times B$$

has the Agda type:

$$R : A \to B \to \mathrm{Set}$$

In Agda, the type $R\ a\ b$ is inhabited *iff* in standard mathematics, $(a, b) \in R$.

Existential quantifier. Assume $A : \mathrm{Set}$ and $\varphi : A \to \mathrm{Set}$. An element of the type $\exists\ A\ \varphi$ (existential quantification) is a constructive proof that

there exists an $x : A$ such that $\varphi\ x$ is inhabited. It is given as the pair $(x\ ,\ q)$, such that $q : \varphi\ x$. In Agda, it is defined using a record as follows:

```
record ∃ (A : Set) (φ : A → Set) : Set  where
  constructor
    _,_
  field
    π₀ : A
    π₁ : φ π₀
```

This definition can be confusing at first glance. To help clarify its use consider the statement: *there exists an even natural number*. To prove this statement in Agda, define a unary relation $Even : \mathbb{N} \to \text{Set}$ with canonical semantics; instead of Boolean values *Even* maps a natural number onto the singleton set $\top$ or empty set $\bot$. The above statement would then become $\exists\ \mathbb{N}\ Even$. A proof would be given as an element of this type, namely $(2\ ,\ \text{tt})$, where tt is an element of $Even\ 2$. Conversely, there is no element of $\exists\ \mathbb{N}\ Even$ of the form $(3\ ,\ p)$ because p would need to be an element of $Even\ 3$ which unfolds to $\bot$ (the empty type).

Remark

The definition of existential quantification is also known as the dependent pair type or Σ type.

Ex falso quodlibet. When the principle of *ex falso quodlibet* is required, the function efq is applied. This has the canonical signature:

$$\text{efq} : \forall\{A\}\ .\ \bot \to A$$

1.5.3 Infinite Data

There are a couple ways to represent infinite data in Agda. The first method is by the use of streams, i.e. for some type A, a stream of type A is an element of the type $\mathbb{N} \to A$. In addition Agda has special support for representing infinite data-types, or equivalently co-algebras. This is by the use of three special built-in functions. These functions are used to box a computation, and then unfold it, provided the computation is productive. These functions are required to create objects by co-recursion, and then destruct these objects

with co-induction. The functions have the signatures:

$$\begin{aligned}\infty &: \forall\{A\}\ .\ A \to \mathrm{Set}\\ \sharp &: \forall\{A\}\ .\ A \to \infty\ A\\ \flat &: \forall\{A\}\ .\ \infty\ A \to A\end{aligned}$$

Consider the example of a co-list, these are lists that have no nil element. They are defined as follows:

$$\begin{aligned}&\mathrm{data}\ \mathrm{CoList}\ (A : \mathrm{Set}) : \mathrm{Set}\ \mathrm{where}\\ &\quad \_::\_\ :\ A \to \infty\ (\mathrm{CoList}\ A) \to \mathrm{CoList}\ A\end{aligned}$$

It is possible to provide elements of a CoList by using co-recursion, i.e. a co-list that is always 0 is defined as follows:

$$\begin{aligned}&\text{const-0} : \mathrm{CoList}\ \mathbb{N}\\ &\text{const-0}\ =\ 0\ ::\ (\sharp\ \text{const-0})\end{aligned}$$

or a co-list that counts is given as follows:

$$\begin{aligned}&\mathrm{cosuc} : \mathbb{N} \to \mathrm{CoList}\ \mathbb{N}\\ &\mathrm{cosuc}\ n\ =\ n\ ::\ (\sharp\ \mathrm{cosuc}\ (\mathrm{suc}\ n))\end{aligned}$$

The co-lists can be destructed by using co-induction, such that each step is productive. For instance two co-lists of natural numbers can be combined into a single list by applying addition.

$$\begin{aligned}&\mathrm{plus} : \mathrm{CoList}\ \mathbb{N} \to \mathrm{CoList}\ \mathbb{N} \to \mathrm{CoList}\ \mathbb{N}\\ &\mathrm{plus}\ (n\ ::\ ns)\ (m\ ::\ ms)\ =\ (n + m)\ ::\ (\sharp\ \mathrm{plus}\ (\flat\ ns)\ (\flat\ ms))\end{aligned}$$

Here each unfolding of plus produces a constructor of a co-list.

It is required that co-recursive functions fulfil the guarded recursion property. That is, each defining equation is a constructor applied to one recursive call. For example, assume a co-inductive type $A : \mathrm{Set}$ with the constructor $C : B \to \infty A \to A$, and define:

$$\begin{aligned}&f : B \to A\\ &f\ b\ =\ C\ b\ (\sharp\ (f\ b'))\end{aligned}$$

Here, there is no restriction on the size of b' to be less than b. Therefore, attempting to write a function that breaks the guardedness constraint, such as summing a co-list into a number, which is defined as follows:

$$\begin{aligned}&f : \mathrm{CoList}\ \mathbb{N} \to \mathbb{N}\\ &f\ (n\ ::\ ns)\ =\ n + (f\ (\flat\ ns))\end{aligned}$$

will fail the Agda termination checker.

Remark

This is only a fragment of Agda, other important features are the built-in and primitive functions, they are explained in Chapter 5 of thesis as they are directly relevant to the research. There are other features of Agda not considered as they are not relevant such as universe polymorphism and irrelevance. The interested reader is directed to [Agd12, Nor07, BDN09] for more information.

1.5.4 Font Faces

The following convention is used with regards to font faces,

Roman	standard text
Italic Roman	newly introduced concepts
Bold Roman	headings, highlighting important information
$\mathcal{M}$ath font and Roman	Agda definitions, e.g. data-types, functions
Sanserif	programming code, usually Haskell
`Teletype`	listing technical details

Chapter 2

Review of Literature

In this chapter an overview of related work is presented. It is given in two parts. First, the work related to railways is presented. Then work pertaining to combining interactive and automated theorem proving paradigms is presented, with a focus on Agda and type theory.

2.1 Railways – A Grand Challenge

In 2004, Dines Bjørner declared understanding and effectively utilising the Railway Domain as a *grand challenge* within the field of computer science [Bjø04, Hoa03]. The first selection of reviewed literature pertains to the identification of the grand challenge at IFIP 2004, and then other related work will be looked at.

2.1.1 Articles from IFIP 2004

The TRain topical day was held at the 18th IFIP World Computer Congress, Toulouse, France and organised by Bjørner. Papers relating to it are [Bjø04, Sab04, MW04, PB04, ROTS04, GL04, Ogi04, BCJ+04], of which the opening article [Bjø04] is of especial interest. That article lays out the problem of domain analysis with respect to railways as a grand challenge called TRain[1]. All subsequent papers further the topic.

The first paper [Sab04] is by Sabatier, it focuses on reusable formal models in the railway domain; it also provides advice for developing models that are reusable. It advises designers to ask questions as the model is being constructed, such as "why do we want this property?" and "what can we assume from the external environment?". The paper also emphasises financial issues

[1]Project web page: http://www.railwaydomain.org/

for the construction of reusable models. Notably it is suggested that creating reusable models costs more than creating tailored (non-reusable) models. However, over time more money is saved by reusing models, than is invested when compared to non-reusable models. The paper concludes by suggesting that a central repository of such models should be setup.

The second paper [MW04] discusses how to apply Communicating Sequential Process (CSP) to create a high-level model of an Austrian railway signal[2]. The abstract models are shown to be correct by applying model-checking. Then they are successively refined to produce a working, verified implementation. At a higher-level, the paper pays particular attention to the issue of the abstract models being fail-safe. This means that if any part of the system should fail, then it should fail into a safe state.

The third paper by Penicka and Bjørner [PB04] discusses a partial model of the railway domain that covers many areas, although a short paper, its references include full details of the model shown [Bjø03, BBM99, PSB03, SPB03]. The model is constructed using the Rigorous Approach to Industrial Software Engineering (RAISE) specification language (RSL), and it encompasses many aspects of the railway domain. These aspects include timetables, staff rostering, routes and the underlying topographical anatomy of the railway. It contains as well a discussion of the author's experiences using other techniques such as sequence charts, state charts and Petri nets within the railway domain.

The Fourth paper is by Reif et al. [ROTS04], it is about application of formal methods to train systems to ensure safety properties. The paper contains a short case study about the verification of decentralised radio controlled level crossings used by the German railway organisation *Deutsche Bahn*. These level crossings have a decentralised control system. Briefly the operation of these crossings is as follows. The train uses a radio signal to inform the level crossing that it is approaching, then the level crossing uses a sequence of timers to calculate when, and for how long it should lower the barrier, and then safely raise it. The control system is also responsible for setting the road facing signals. Reif et al. applied interval temporal logic (ITL) along with fault tree analysis (FTA) and failure modes and effects analysis (FMEA) to verify the device. The device was modelled using state charts that were developed in a top-down manner. First, entities such as the train, level crossing, communication system and environment were modelled, then these components were decomposed. Furthermore, using an appropriate model-checker, the safety requirements formalised as ITL formulæ were verified. The verification found some of the timers were incorrect, meaning the

[2] *Montigel Dwarf Signal*

barrier could be raised too soon. The specification was amended, and then safety verification was successful. The paper concludes with a discussion of applying quantitative measures to the model. It states that FTA and FMEA can be easily extended such that their quantitative values can be calculated.

The fifth paper is by Giras and Lin [GL04], in which they applied the *axiomatic safety-critical assessment process* framework to a hypothetical case study of a Maglev system. The result of this is a stochastic risk assessment for the Maglev system. The following underlying techniques were used: fault tree analysis, Markov chains and the use of a Monte Carlo risk assessment algorithm.

CyberRail The final two papers [Ogi04, BCJ+04] discuss the CyberRail project. The first of these introduces the aims and goals of CyberRail, the second starts to define a RSL model of CyberRail. The CyberRail project ambitiously attempts to modernise public transport by exploring how ubiquitous computing could affect journeys. It describes a scenario where travellers have a *virtual travel companion* which: provides them with the necessary information, books tickets when required, reschedules the journeys in case of a missed connection, arranges taxis, and keeps the passenger entertained while travelling. The name CyberRail was chosen because it exists within "cyberspace" and it communicates with entities in the physical world such as the railway network, staff and passengers. CyberRail can be decomposed into three classes of objects:

Demand Objects that represent actual user demands, such as passenger profiles, e.g. where they are travelling to, preferences, travel history and current position.

Carrier Objects that represent transport mediums, e.g. trains, automobiles and aeroplanes. The objects should have specific attributes such as where they are, which passengers are travelling within the object, and maintenance information.

Route This class represents objects that could be seen as timetables of various transport mediums. For example, railway, motorcoach and aviation routes along with times and constraints. These constraints could be seasonal or sporadic, e.g. referring to national holidays or weather conditions.

2.1.2 Railway Specification and Modelling

The scope of the attempts originally pioneered by Bjørner [Bjø04, BGP95], aim to capture the entire domain from the physical components, to staff rostering, rescheduling a delayed train, timetable scheduling during holiday seasons and much more. This comprehensive approach is known as *domain analysis* and has been applied to other domains. Some domains of interest include the aviation and maritime industries. It is suggested by Bjorner that these domains (and others) can be combined to create an overarching specification of the transportation domain [Bjø06].

Over the past 30 years, there have been many attempts to formally model various aspects of railways. Some, including the grand challenge mentioned previously, attempt to capture as much information as possible while creating the models, these models are typically written using a specification language such as VDM, RSL, B/Event-B or CASL [Jon90, Geo91, Abr96, ABK+02].

A well-known example of this algebraic specification approach is the Paris metro, *Métro Est Ouest Rapide* (METEOR) [BBFM99]. METEOR is a driverless metro line that was partly built by Matra Transport International (now part of Siemens Mobility). The line is protected by an automatic train protection (ATP) system and controlled by an automatic train operation (ATO) system. The control systems for this line are distributed over a large geographical area; some of the control is centralised, and some is by decentralised control units that are located beside the track and onboard the trains. This was an extremely complex system to develop, and the fact that it was a critical system means that there was no margin for error. Matra Transport decided to use the *B-Method* during development [BBFM99, BG00, Sab04]; B [Abr96] is a formal language which is the successor to Z. It is a complex language to use partly because the specification and implementation are deeply intertwined [Abr96] which requires successive refinements to obtain an implementation. There is extensive tool support that simplifies the process; one such tool set is called Rodin [ABH+10]. Perhaps Matra Transport decided to use the B-Method as it was developed in France, so they had a wealth of local expertise. Using the B-Method Matra Transport were able to verify 100% of the safety and liveliness requirements of the ATP/ATO systems. It was claimed that no bugs were found during the validation, in house testing, on-site testing and since METEOR went live [BBFM99]. This is clearly an exceptionally meritorious result for such a complicated, critical system.

Other work relating to the specification of railways acutely focus on the topology and operational aspects. For example, the approach taken by Hansen which originated from Monigel uses graph theory [Han98, Mon92].

The basic idea is that track segments are vertices, and the edges are connections between the segments. The edges are directed; hence they must be doubly linked. The reason they are directed is that it allows the edges to be annotated in only one direction. For example, signals are added to the model by annotating edges, this allows the model to represent that the signal is only visible in one direction (which is a requirement in the railway domain). Hansen [Han98, Jon90] specified Danish interlocking systems using the Vienna Development Method (VDM) specification language. This specification was simulated so that domain experts could validate its correctness. The first attempt omitted manual overrides that are used for shunting and controlling sets of points. The second attempt was augmented to provide these manual overrides, then verified against the safety requirement *that the interlocking could not allow trains to collide or derail*. Although safety conditions that are more precise were not verified, this high-level requirement is the ultimate goal of the interlocking with respect to safety (not liveliness) properties. See Section 12.1 for a detailed comparison between Hansen's work and this thesis.

It is clear from Hansen's work that using a graph as an underlying datatype to represent the topology is flexible and extensible. There are a number of different operations that can be applied, for example, the vertices relating to routes can be coloured to determine incompatible combinations, or the graph can be traversed to determine whether the topology is well-formed.

Another approach is to use predicates. They express a richer view of information than the graph based approach. Predicate logic has been successfully applied in several projects to model railways [Eri97a, Kan08]. Note that a first-order predicate logic underpins the previously mentioned algebraic specification languages. It is straightforward to define n-ary relations between objects: a *Prolog-like* syntax is suited well to this. For example, this is the approach taken by Eriksson when modelling the Swedish rail topology [Eri97a, Eri97b] and myself while modelling a London Underground station [Kan08]. Eriksson provides an overview of a case study in [Eri96]. Simply put, consider two track segments, `ts1` and `ts2`, then `connected_to(ts1,ts2)` and `connected_to(ts2,ts1)` are defined *iff* they are connected in the physical world. Using objects and relations between these objects, the physical world and abstract concepts (such as routes) can be formally modelled. Provided the model adheres to a suitable nomenclature the model will be readable by humans. Having a formal model of the topology allows for information to be automatically deduced, e.g. pairs of incompatible routes can be deduced as in [Kan08].

RaCoSy/PRaCoSy

The UNU International Institute for Software Technology (IIST) undertook the People's Republic of China Railway Computing System (PRaCoSy) project, which was sponsored by the Chinese Ministry of Railways. The project goes a long way to developing an RSL model of the railway domain [BGP95]. The model is a precursor to the to the model presented in [PB04]. PRaCoSy was developed from 1993 to 1996. It was primarily aimed to allow in-house development of arbitrarily advanced railway related software, and secondarily had the aim to internationalise commercialised railway software[3]. During this project, a *running map* tool was developed. The tool helps to schedule trains based on constraints entered by the user, the tool recommends solutions, as opposed to only validation/simulation. This tool was developed from the RSL model and delivered to the Chinese Ministry of Railways.

EURIS and LARIS

European Railway Interlocking Specification [BMS92] (EURIS) is a graphical method of designing interlocking systems. It consists of four sub-languages; three of these languages are used to model the topology and routes. The final language models how the actual entities, such as a signal, operate. The semantics of EURIS has not been fully defined, thus proving theorems about a EURIS specification is difficult due to ambiguities. An attempt has been made to verify safety properties of a EURIS program by translating the program into a Petri net which has a precise mathematical meaning. There are many tools that simulate and show properties of Petri nets, for example, one such tool is *ExSpect*. A thorough discussion of this technique can be found in [BBV95].

The lack of a formal definition of EURIS resulted in a second language being defined. This language is called: Language for Railway Interlocking Systems (LARIS). LARIS 1.0 was developed by Fokkink et al., at the CWI, Amsterdam [FGHvV98]. LARIS modularises the interlocking. A module sends messages to other modules or to the real world. Modules also receive messages from the real world. All these modules are assumed to be asynchronous; therefore, the order the messages are processed in is not fixed. The language is designed to be easily formulated in a process calculi to aid in later verifications.

[3]Typically developing countries will acquire railway systems from industrialised countries.

2.1.3 Railway Verification

The formal verification of railway interlocking systems has been studied a number of times over the past 20 years. Some attempts have already been identified, such as the METEOR line in Paris; a selection of other attempts is reviewed below.

The first verification attempts focused on determining whether an interlocking system satisfied concrete safety properties, i.e. formulæ represented in terms of the input and output variables of the system. During the mid-nineties, a notable body of work relating to the verification of the interlocking systems at Hoorn-Kersenboogerd and Heerhugowaard (Netherlands) was undertaken by Groote, Fokkink and Martens in [GvVK95, Fok95, Mer96]. In that work, the process algebra μCRL (Common Representation Language) was applied to model the interlocking systems. Although process algebra is amenable to model-checking techniques, the available model-checkers of the day were not able to explore the state space feasibly, so the formulæ were translated into propositional logic, and then SAT solving technology was applied instead. It is noted that the same SAT based techniques were successfully applied in Sweden by Eriksson [Eri97a, Eri97b], where a serious flaw in the interlocking system was identified. A number of years later in 2002, the same stations/problem sets of Hoorn-Kersenboogerd and Heerhugowaard were revisited by Eisner in [Eis02], where CTL model-checking was successfully applied in-place of SAT solving. The paper claims that interlocking systems with approximately 600 variables were efficiently explored using the symbolic model-checker *RuleBase*.

Orthogonally during the mid-nineties an effort was made to apply model-checking to a more complicated class of interlocking systems that are programmed using the Geographic Data language [Mor96, SWD97]. One notable piece of work pertaining to the Geographic Data is Mathew Morley's thesis [Mor96]. In that thesis, he applies the Calculus of Communicating Systems (CCS) and μ-Calculus [Sti01] to verify safety properties[4] of Westinghouse's Solid State Interlocking (SSI). The SSI is based around a data driven loop, there is one generic component akin to an interpreter that executes the various commands specific for the interlocking. Morley defined how the interlocking and its interpreter can be modelled using CCS and also defined a translation from the SSI's command set into CCS, thus creating a complete model of the interlocking. Then using μ-calculus and the Concurrency Workbench, he verified safety properties. The project also verified the communications between interlocking systems. An example of this inter-interlocking communication occurs when a train route is spanned through

[4] μ-calculus can also be used for liveliness properties.

multiple interlocking systems, and the interlocking systems must communicate information regarding the whereabouts of the train and route status. The transition from verifying a single interlocking system, to verifying the communications between multiple interlocking systems was greatly simplified through the use of CCS.

The verifications described thus far do not make safety verification tractable. They only provide a mechanism to perform verification of safety and/or liveliness properties for interlocking systems. In [HP00] Anne Haxthausen applied the RAISE specification language (algebraic specification) to perform safety verification on distributed interlocking systems. Notably, in that work the high-level formalisations of safety are introduced. The safety formalisation used is that *trains do not collide, or derail* (the same high-level formalisation used in this thesis). Although this is a noble safety requirement, little attention was paid to signalling principles, and what they had to do with safety, as is done in this thesis. A good discussion of signalling principles in relation to formal methods is found in [TRN02, RN03], where control tables are synthesised based on a selection of signalling principles.

There have been a few similar attempts to model interlocking systems and verify them using high-level formalisations. One of the earliest attempts was by Bernardeschi in [BFG+98], where process algebra and ACTL (fragment of CTL) was applied to verify a level crossing control system. In that work, the use of high-level formalisations was not considered, the examples given were of a significantly lower-level than proving that trains do not collide. Winter also implemented a similar interlocking verification process in [Win02]. He modelled a number of interlocking systems using CSP and proved them to be safe with respect to a selection of signalling principles using the model-checker FDR. The selection of principles included the previously mentioned high-level formalisation. The author notes that process algebra is not best suited to model the constraints in control tables. Later Winter et al. in [WR03] applied the model-checker NuSMV [CCG+02] to larger problem-sets; however, it is noted that the issue of an exploding state space was encountered which limited the feasibility of such an approach.

Remark

During this work the same issues (using the same tool) were encountered with model-checking interlocking systems as was identified in [WR03].

More recently, and closer to this aims of this thesis is the work by Sabatier et al. in [SBRG12]. Although it is only a summary paper (4 pages), it is clear

that Sabatier is working on a formal proof of a New York City subway control system. The high-level safety requirements used are that trains do not collide, derail or exceed the speed limit. The control system is distributed; each train has an on-board control system. In this work, the underlying logic is Event-B. However, no explicit reference is made to signalling principles; instead they are generalised as assumptions from the target domain. The paper further discusses rules-of-thumb to determine if they are good or bad assumptions. For instance, one of these rules is that the validity of an assumption should be obvious.

A comparison of this thesis to a selection of these papers [Han98, HP00, Win02, SBRG12] will be carried out in Section 12.1. This comparison is deferred until the end of thesis so that it is clear what has been done the same, and what has been done differently.

Swansea Railway Verification Group

At Swansea University, we have formed a group of researchers dedicated to exploring verification in the railway domain. The projects range from safety verifications, to safely optimising capacity. These areas have been explored using a multitude of techniques, some of which include SAT solving and model-checking for the verifications, and creating formal models of the railway domain for the specifications.

The rail group initially focused on safety verification of interlocking systems [Kan08, Jam10, Law11]. In these three projects, the techniques of the safety verifications were explored and honed, in respective order, they used: inductive SAT solving (the author's masters thesis), SAT based model-checking and the SCADE tool set[5].

Subsequently the goals of the group were extended to include safely optimising the capacitance of the railway network [MNR+12a, MNR+12b]. This work explores different variants of the signalling principles, and what effect they have on the capacitance of the network.

Another project is to explore the use of domain specific languages in the railway domain [JR11]. This includes defining a graphical language that the topology can be represented within, then automatically deriving from one of these diagrams the required constraints for the network to be safe. This information is then automatically translated into a control table, i.e. a high-level specification of an interlocking system.

[5]SCADE is a propitiatory tool set developed by Esterel Technologies.

2.2 Development of Verified Software

A few attempts have been made to automate the process of producing control systems. These approaches are similar to Dines Bjørner's domain analysis. Typically, a specification of the railway is made using a formal specification language such as RSL, then some refinement method is applied to produce a working system that fulfils the specification. Assuming the specification has been validated to be correct, then the control system will meet these conditions, as well.

This is the procedure followed by the previously mentioned METEOR project [BBFM99]. It was also followed by Anne Haxthausen and Jan Peleska while attempting to automate the creation of tram control systems [HP02], and distributed railway control [HP00] in Germany. The tramway domain and distributed railway domain specification were created using RSL. Also, each track-side functional module such as a signal or set of points is modelled abstractly, and most importantly the states of these modules are modelled.

In the tramway example, these abstract models of the individual track side modules are defined by small deterministic sequential state machines that determine how a module switches from one state to another. Verification can then proceed by deciding which states of the track-side functional modules were in conflict, e.g. two opposing signals should not both show the proceed aspect. All possible transitions into these "bad states" are computed from the state machines, these transitions are then used to build a "safety monitor" state machine.

The formal development of the distributed railway control systems was performed in a more traditional manner. The specifications were made executable, then using these high-level specifications of the distributed railway domain, safety requirements such as "no collisions" and "no derailments" were verified. Subsequent refinements were made preserving these properties until a working system was produced.

Verification is carried out on multiple levels, one level of interest are control tables. It is suggested that model checking could be used for this verification in [HP02]. This article also suggests that the generated code can be automatically verified against the control tables. In [FHG+98], Fokkink et al. demonstrate how this can be implemented.

2.2.1 Frameworks

In recent years there have been a number of frameworks aimed at bringing formal methods within the reach of the standard critical system developer [Jon90, Geo91, ORS92, Abr96, ABH+10, RL12]. One of the first viable frame-

Remark

The word framework (in software engineering) means a collection of reusable and generic tools and libraries. However in this thesis a loose characterisation of a framework is used, which is almost synonymous with a platform. A framework is characterised as a collection of tools that support the development of verified software, for instance the functionality should include, but is not limited to: type-checking, formal methods, pretty printing, interactive proving, support for automation and obtaining fully functioning programs.

works of this kind was the Prototype Verification System (PVS) [ORS92]. Although VDM and RAISE predate PVS as development methods, PVS was a framework, as opposed to a collection of tools. The goal of PVS was to provide an integrated interface that allowed developers to specify their control systems at a high-level during an early phase of the design. The system then facilitated specifying properties about the control system, essentially proving that the models fulfilled the specifications is performed using an interactive proof system. The system supported a number of tools that would automatically attempt to discharge proof obligations. These obligations typically arose from the use of refinement types, for example, the divide operator has a refined type such that the denominator is not 0, then everywhere that division is used a proof obligation must be discharged, namely that the provided denominator is not 0.

The basic techniques used in PVS have been repeated in subsequent frameworks. One of the more recent frameworks of this style is the Rodin tool set [ABH+10]. It is based on Event-B and is still under active development at the time of writing. A number of improvements have been made since PVS; most notable of these is to provide a WYSIWYG (point-and-click) interface implemented using the Eclipse environment. The idea is that the interface is closer to what an end-user is accustom to (e.g. an office suite), such that it provides helpful feedback and prevents making syntactical mistakes. After the user has axiomatised the control system and provided a number of specifications to prove, the user is presented with an interactive proof interface. The interactive proof interface is predominately tactic orientated. There are a number of buttons which trigger the tactics. The available tactics depend upon which plug-ins have been installed. The user is also presented with a basic proof tree that shows the user where in the proof they are, and what assumptions are available.

The Rodin tool set appears to be a beneficial step towards bringing formal methods to critical system developers. However, there are questions of certification for using Rodin for critical system development. While reviewing Rodin, I found the interface confusing, and overly constraining, perhaps as I am not accustomed to proving properties by tactics. Especially the functionality of these tactics is often not clear, at least not without looking in the documentation for their definitions. Often it is necessary to prove the theorems on paper first. This is a common problem with proof user interfaces: it is hard to facilitate expressive mathematical access while still maintaining usability for programmers. In fact, Owre notes in [ORS92] that there is a well-known trade-off between the expressiveness of the logic used in a framework and the amount of automation that is possible.

Finally, another relevant framework was created by Russo and Ladenberger [RL12]. The framework is called VeRaSiS and is specifically tailored for the (Brazilian) railway domain. The framework is also based upon Event-B, where the topology of the railyard is graphically formalised. The graphical models also specify constraints on the equipment that must always be fulfilled (i.e. specifications). Then these models are translated into a Boolean equation system, and further translated into an Event-B model. Once represented as an Event-B model, the model-checker *ProB* is applied to determine if the topology is correct, if it is not correct, a counterexample is emitted. This project is developing; however, they have provided some early results from a case study where an error was identified in an existing railway layout. It appears that no attempt is made to prove the correctness of the translations, especially when translating from the graphical language into Boolean equations. Similarly, no assurances are given about the correctness of the specifications. Simulation was used to validate the specifications.

2.3 Theorem Proving

Theorem proving tools can be placed into one of two categories, interactive or automatic, see Section 1.3.1 for more information about this distinction. A good introduction to the various different flavours of theorem proving can be found in [Har08], and a more technical analysis in [Bou97]. A review of formal methods relating to industrial projects can be found in [WLBF09].

2.3.1 Integrating ITP and ATP

Since N. de Bruijn introduced AUTOMATH in the late 1960's [dB70], the ITP community has studied the issue of automatically solving problem sets

Remark

In this thesis, the ATP tools that are of interest coincide with decision procedures for logics.

many times [BGHL10, BN10, Bou97, FMM+06], and many more. During this research the following three existing approaches were identified, none of which categorises our approach, which is the subject of the next chapter. Note the abbreviations in brackets, these will be used throughout the thesis to refer back to the categories defined here.

Oracle (Only Oracle)

The use of an oracle, which is an operation that provides a proof of a theorem, and which when invoked calls an external tool. See for instance approaches by Owre and Rush in PVS; Tverdyshev, Müller and Nipkow in Isabelle; and Bierman, Gordon and Langworthy in M [ORS92, MN95, BGHL10]. There are too many interactive tools to list that use external tools as oracles, but relevant to this project is the Rodin tool set [ABH+10] which is similar to PVS in that it tries to help developers write verified software. The problem with this approach is that the proof-object obtained from the oracle does not have any reduction rules, and therefore it destroys the ability to execute proofs as programs. Another problem is that it is not clear whether the external tool is correct and whether the result of the external tool was interpreted correctly.

Reflection (Only Reflection)

The correctness of a decision procedure is verified in the ITP tool, and then from this proof a verified ATP tool is extracted [Bou97]. The extracted program is then called by a tactic in the ITP tool to build a proof-object for each application. This approach has for instance been taken by Verma; Hendriks; and Lescuyer and Conchon [Ver00, Hen02, LC08], and has been widely applied within the Coq [The04] community. Reflection requires a correctness proof for an ATP tool in the ITP tool. Proving the correctness for state-of-the-art theorem provers would be a cumbersome activity as efficient solvers are typically unverified and utilise low-level machine code; also, one would expect the stat-of-the-art tools to have significantly improved by the time the proof is complete.

Oracle with Justifications (Oracle + Justification)

One uses an ATP tool which provides in case of a positive answer a proof-object [DT84], which can then be translated automatically into an ITP proof of the corresponding ITP formula. Since the ITP proof is now checked, the correctness relies entirely on the correctness of the ITP checker. Furthermore, for each application a proof-object is kept, and therefore, the ATP has to be executed only once. This approach has been used for instance by [PS07, BN10, DRS03, Web06, FMM+06, AGST10, YL97]. Problems are that creating the proof-object slows down the ATP tool and that many ATP tools do not provide a proof-object. Furthermore, the proofs provided by the ATP tool might be particularly large (proof sizes of several hundred Megabytes have been reported for Satisfiability Modulo Theories (SMT) solving [Stu09]). Therefore, checking translated ITP proofs might be infeasible, especially when dealing with type theoretic ITP tools. This is due the size of the proof terms and garbage collection issues.

In Chapter 6 an implementation of this technique taken by Armand et al. [AGST10] for integrating a SAT solver in Coq is explored, and re-implemented in Agda. However a number of efficiency problems were encountered.

2.3.2 Agda and ATP Tools

There have been a small number of successful attempts to integrate Agda with automated theorem provers, these attempts are aimed at trying to prove arbitrary Agda goals, whereas our approach is aimed specifically at industrial verification. One of these attempts produced an extension to Agda known as *AgSy* (Agda Synthesiser) [LB06]. AgSy performs simple inductive proofs. It is sometimes necessary to provide *hints* to AgSy to help the proof search, they are specified by function names. AgSy is categorised as approach (Oracle + Justification). If it successfully proves the goal, then it will fill in the goal in question with a normalisable proof-object.

Another early approach was by Makoto Takeyama [Tak09]. The motivation is the same as ours; he wants to facilitate feasible industrial verification using Agda. However, the approach is slightly different. The first approach he tried was (Only Oracle), where external tools are executed to determine validity of theorems. An Agda program was written to correctly decompose the goal to prove into sub-goals, then using an ATP tool these sub-goals were proved. This is a two step procedure: first, during type-checking the goal is correctly decomposed into postulated sub-goals, then secondly, after type-checking the postulated sub-goals are discharged by ATP tools. This is

achieved by writing an Input/Output (IO) Agda program that decomposes the goal, executes the external tool and examines its output. This program is type-checked, compiled and executed. If the tool could prove the sub-goals, then the program will print out a success message, otherwise it will printout a failure message. Therefore, checking that the Agda code is correct is a two step process, first, type-check using Agda, then as a second step, compile the program and execute it to determine the validity of the sub-goals.

Takeyama further extended his approach to (Oracle + Justification), this is where the external tools produce certificates that Agda checks are correct. This entailed augmenting the above procedure by writing an IO Agda program that would check the certificate, this program is then type-checked and compiled. Using the same method as before, the compiled program executes the external tool. Instead of only examining the output, this time a verified Agda program checks that the result of the tool is correct. This method was successfully applied to a number of ATP tools, notably the model-checker SPIN and the SMT prover Yices. This method is good as it is generic and does not require modifications to the Agda source code. However, as the programs are wrapped in the IO monad, it means that the tools cannot be executed in interactive mode. Takeyama claims that there is little point running ATP tools in interactive mode as type-checking could take too long to complete.

A recent attempt was by Simon Foster, where the theorem prover Waldmeister was integrated into Agda [FS11]. This work involved modifying Agda's sources so that it will parse the output of the tool and reconstruct the proof-object from it, possibly by introducing intermediate lemmata. This approach is also categorised as (Oracle + Justification).

Another attempt has been made by Bove et al. [BDSR12]. This approach is categorised as (Only Oracle), in this work they focus on proving first-order properties of recursive functions. It is claimed that the technique is applicable to inductive and co-inductive definitions. The integration is achieved by creating a dependently typed model of type theory that is more general than Agda's model, then translating the goal to be shown into this model. They have defined translations from theorems in this model into TPTP format files, and then these files can be proved (or not) by any first-order TPTP prover.

Part I

Theorem Provers

AUTOMATH is not intended for automatic theorem proving. Theorem proving is a difficult and time-consuming task for a machine. Therefore it is almost imperative to devise a special representation of mathematical thinking for any special kind of problem. Using a general purpose language like AUTOMATH would be like using a 'contraption that is able to catch flies as well as elephants and submarines'.

– N.G. de Bruijn
1970
[dB70]

Chapter 3

Oracles and Reflection

The three different approaches for composing ATP and ITP tools each suffer from different issues outlined in the previous chapter. Notably, the current *gold standard* of oracles that justify their results (Oracle + Justification) can be surprisingly inefficient when the justifications become several hundred megabytes in size. This approach will be referred to as (Oracle + Reflection).

When performing industrial verification, it is often not required to explore these justifications; it is only necessary that a proof exists (oracle). These issues led to the exploration of a new technique that aims to be light-weight, fast (oracle) and sound (reflection). An oracle is used to provide one-step reductions of decision procedures on ground terms, and reflection is used to construct proof-objects when they are inspected.

In Section 3.3 an exploration into extending this technique to allow for oracles with justifications is presented.

3.1 General Technique (Oracle + Reflection)

Assume a logic (such as propositional logic), and define the set of formulæ and the satisfaction relation (semantics) with respect to the logic.

$$\begin{aligned} &\mathrm{Formula} : \mathrm{Set} \\ &[\![\_]\!] : \mathrm{Formula} \to \mathrm{Set} \end{aligned}$$

Then the decision procedure is implemented in Agda's logic with the focus on an easy proof of correctness rather than efficiency, in fact, it could be naïve and highly inefficient. Later the decision procedure is overridden by a call to an external tool, hence the Boolean valued result.

$$\mathrm{check} : \mathrm{Formula} \to \mathrm{Bool}$$

It is then proved to be correct with respect to the satisfaction relation.

$$\text{correct} : \forall \varphi \;.\; \mathrm{T}\,(\text{check}\;\varphi) \leftrightarrow [\![\, \varphi \,]\!] \qquad (*)$$

where the function T maps Boolean values into Set, i.e. true maps to $\top$ and false maps to $\bot$, see Section 1.5 for the definition.

In Agda, a function's implementation can be overridden for closed terms by a native Haskell implementation. The native implementation is specified in the Agda is source code and assumed to be correct, also the Agda function is checked to fulfil a number of axioms before it is overridden, these axioms are also specified in the source code, see Section 5.1 for more information. The terminology *built-in* refers to overriding a function in such a way. Using the built-in mechanism, the inefficient decision procedure is replaced by a call to an external ATP tool. Evaluation of the decision procedure is as follows: if applied to a closed term, the efficient ATP tool is executed; if applied to an open term, the naïve (inefficient) Agda implementation is evaluated. Since the correctness proof $(*)$ refers to open terms, it refers to the Agda implementation and not the ATP tool. If the resulting proof-object is inspected, it is lazily evaluated using the naïve implementation; otherwise it behaves as if it has been postulated.

To simplify validating that the native implementation is used correctly, the input language of the overridden function should be an Agda data-type that represents, as closely as possible, the input language of the tool. If the original problem requires a translation into this input language, then we define the correctness of the problem using the input language of the tool, and define a translation of the original problem into this input language. Then it is proved that if the correctness of the problem related to the input language holds, then the correctness of the intended original problem also holds. Therefore, the translation is provably correct.

An example of translating the input is performed in Section 4.2.2, with CTL model-checking, where the transition relation is made total before executing the external tool. This is because the ATP tool requires that the transition relation is total. The problem for the non-total transition relation is inside of Agda reduced to this situation.

To increase trustworthiness, we log the inputs and outputs of the ATP tool, thus, allowing a user to verify manually that the translation and the tool's result are correct. Our approach yields a high-level of soundness and efficiency when using certified ATP tools. See Chapter 5 for technical details on the embedding and Section 3.2.1 for a discussion of the soundness.

This concludes the conceptual overview. The following sections explore aspects of the technique.

3.1.1 Logic

When checking whether a formula holds in the logic, it is common that the truth of the formula depends upon a model $\mathcal{M}$ (what's constant) and environment ξ (what varies). Thus, the satisfaction relation becomes:

$$[\![\_ \vDash \_]\!]_{\_} \;:\; \text{Model} \to \text{Formula} \to \text{Environment} \to \text{Set}$$

For example, in CTL model-checking (see Section 4.2) the model consists of a transition system and a state[1]; the environment is an infinite run of the transition system from the state identified in the model. In SAT (see Section 4.1), the model is trivial, but the environment assigns Boolean values to variables in the formula. In the case of first-order theorems, the model consists of the semantics of the signature, and the environment is an assignment to variables in the formula.

3.1.2 Formulæ

Formulæ in the logic are inductively defined types who's elements are finite. These formulæ are defined over a model $\mathcal{M}$ and an environment ξ.

For example, in propositional logic the formulæ are inductively defined as the least set closed under atomic propositions, conjunction, disjunction and negation.

3.1.3 Decision Procedure

When performing tautology checking of a formula, the formula is checked to hold for all environments; when satisfiability testing, a test is made that there exists an environment such that the formula holds. In our experience, an inefficient simplistic definition that recurses over the structure of the formula, model and environment is preferable as this helps with proving the correctness. The decision procedure depends on the model and has the following type:

$$\text{check}_{\mathcal{M}} : \text{Formula} \to \text{Bool}$$

Correctness of the decision procedure must then consider the following quantification scheme:

$$\text{correct}_{\mathcal{M}} : \forall \varphi \,.\; \mathrm{T}\,(\text{check}_{\mathcal{M}}\;\varphi) \leftrightarrow \overset{\forall}{\underset{\exists}{}}\; \xi.[\![\, \mathcal{M} \vDash \varphi \,]\!]_{\xi}$$

[1] It is a common misconception that the state is part of the environment, the state s is fixed, i.e. $\forall$ or $\exists$ an infinite run from s in $\mathcal{M}$ such that ...

Here, the choice of quantification depends upon the ATP theory: $\exists$ is satisfiability testing and $\forall$ is tautology checking. For example, in the case of the Boolean satisfiability problem, we have "there exists a satisfying assignment" (satisfiability testing) or "all assignments are satisfying" (tautology checking). It is possible to define more complicated quantification schemes, whereby the environment is split into sub-environments; however this is not considered in this thesis.

3.1.4 Evaluation

The type theoretic implementation of $\text{check}_{\mathcal{M}}$ is typically inefficient compared to purpose written tools, because $\text{check}_{\mathcal{M}}$ is defined naïvely to simplify the proof of correctness. This inefficiency is exaggerated by many implementations of proof systems; specifically relating to this work, type systems make heavy use of rewriting and normalisation resulting in large terms, which would consume vast resources in attempting to evaluate $\text{check}_{\mathcal{M}}$ on all but the simplest examples. For this reason, we override $\text{check}_{\mathcal{M}}$ with an external ATP tool for an efficient implementation. See Chapter 5 for more information regarding the implementation.

In order to maintain consistency, the ATP tool overriding the type theoretic implementation of $\text{check}_{\mathcal{M}}$ needs to be consistent with the decision procedure. Therefore, most examples of semi-decision procedures are excluded, since a naïve implementation and an external ATP tool could disagree on when to return an undefined result. A precise formulation of the consistency of built-in functions is given in Theorem 5.1.1.

3.2 Comparison with Existing Techniques

Obviously, compared with only using an oracle (Only Oracle) the approach has the advantage that proofs normalise and programs can be extracted. Both the oracle and our approach (Oracle + Reflection) become inconsistent if the result of the external tool is incorrect, see Theorem 5.1.1.

As pure reflection (Only Reflection) does not use any external tools, it has the highest level of soundness of all approaches. It does however extract from a proof of a correct decision procedure, a verified decision procedure that is executed as if it were an external tool. In comparison, our approach (Oracle + Reflection), combines reflection with an oracle by using a (potentially unverified) external tool in-place of the extracted decision procedure. So in the case of open terms it is equivalent; but in the case of closed terms it weakens soundness and significantly increases efficiency.

The third approach (Oracle + Justification) where the external ATP tool provides a justification is motivated in the literature [FMM+06] by the fact that, in many cases, non-trivial translations into the ATP tool's input language are defined outside the logic of the ITP tool, making it hard to prove that they preserve correctness. This problem is avoided in our approach by defining all translations inside Agda, mitigating this requirement. It follows by this, that in approach (Oracle + Reflection), the tools do not need to compute justifications, and hence the tools can be more efficient, and the choice of the tool is less restrictive. The chosen ATP tool can range from unverified state-of-the-art tools, to certified, but technologically less sophisticated tools. In summation, approach (Oracle + Reflection) trades the high-level of soundness assurances that the justifications provide, for an increase in efficiency, flexibility and usability; this soundness trade-off is minimal when using certified external tools. The flexibility of the approach is that it easily allows for updates/changes in the external theorem provers, and the addition of new logics. Also in approach (Oracle + Reflection) there is no need for the ITP tool to store and check the justifications.

One should as well note that whatever we do in order to guarantee the correctness of proofs carried out using ITP, we can never obtain absolute certainty. We will always rely on the correctness of the checker of theorems in the ITP (which are usually not formally verified), and on the correctness of its logic (most ITP tools substantially deviate from the underlying logical theories in order to be more user friendly and efficient). Then we rely as well on the correctness of the compiler (it is well known that most compilers have errors) used to compile the ITP tool, and on the underlying operating system. Ultimately, Gödel's incompleteness theorem shows that it is impossible to guarantee in an absolute way that the underlying mathematical theory is consistent.

3.2.1 Soundness.

This approach is sound, provided the ITP tool (Agda) is sound and the ATP tool gives the correct output, which means that it returns true, if the formula to be proved is valid, and false if it is not. The reason is that one shows in Agda that the inefficient function, which is to be overridden, fulfils this property. Since there is only one such function, the overridden function returns the same result as the original inefficient function defined in Agda. So Agda with the function overridden is equivalent to Agda without it, and if the latter is sound, so is Agda with the overriding mechanism.

The logging of the answers of the ATP tool gives the user the possibility to check whether the instances used by the ATP tool gave the correct answer

(e.g. by checking using alternative tools), and therefore reduces the reliance on the soundness of the ATP tool.

We note as well that the input to the ATP tool will be done in the language of the ATP tool (using a syntactic representation in Agda). The translation of the original problem into the ATP tool's input language is done inside Agda, and thus shown to be correct. This avoids the problem of an erroneous translation, which might for instance happen if the translation is carried out by a program outside Agda.

We think as well that many ATP tools are at least as trustworthy as Agda itself (if not even more), so this approach will not weaken the correctness of Agda.

Intuitionism and Classic Provers.

We note that the use of SAT solvers based on classical logic are compatible with the intuitionistic type theory of Agda: the SAT solver is only applied to formulæ formed from decidable prime formulæ (e.g. formulæ of the form $T(b)$ for a Boolean term b), for which the principle of *tertium non datur* (law of excluded middle) holds intuitionistically. The principle of tertium non datur holds for all propositional formulae, provided it holds for the atomic formluæ that these formulæ are built from, so for these formulæ, classical logic holds in intuitionistic type theory. In the case of CTL, provided that it holds for the atomic formulæ it is defined over, one can show as well that the principle of tertium non datur holds.

In fact, the decidability of the validity of formulæ expresses constructively that tertium non datur holds. Therefore, any constructive theory with decidable validity fulfils the principle of tertium non datur, and vice versa.

3.3 Efficient Proof Reconstruction

With approach (Oracle + Reflection), it was remarked previously that it is not efficient to explore the resulting proof-objects in entirety, and there are issues when the decision procedure and ATP tool do not coincide. To remedy this situation a second integration of automated theorem provers was carried out. This second interface is classified as approach (Oracle + Justification), and based upon the work of Armand et al. [AGST10]. In that work, they apply reflexive methods to type-check SAT solver resolution traces produced by zChaff [MMZ+01] and construct proof-objects in Coq [The04].

These techniques formed the basis of an alternate—traditional—interface (cf. Chapter 6), where efficient proof-object reconstruction is possible by

type-checking the justification. However, our implementation is considerably more generic as it is not restricted to resolution proofs. This requires that first the ATP tool provides a justification, and secondly that the two functions below are defined in Agda. The first function decides whether a justification is correct with respect to the logic and original formula/theorem. The second function constructs a proof-object from a correct justification. The justifications are obtained by processing the tools output, in fact, the Agda parser and type-checker are used to parse the tools output and translate the result into an Agda term. Thus, it is possible for the tool to directly output Agda terms that witness the proof, these terms are then checked to be correct proofs. The proof that a justification is correct can be inferred by the type-checker due to the Boolean nature of the check. Essentially the following two functions are defined:

$$\text{check} \;:\; \text{Proof} \to \text{Formula} \to \text{Bool}$$

$$\text{reconstruct} \;:\; \forall \varphi\, p\,.\; \mathrm{T}\,(\text{check}\, p\, \varphi) \to \forall \xi\,.\,[\![\varphi]\!]_\xi$$

For ground terms, reconstruct replaces the soundness proof of the previous technique. It is remarked that in contrast to the previous method, when applied to industrial problem sets, very high-levels of soundness assurances are obtained at the expense of efficiency. If the obtained justification is not correct, then a warning is generated; specifically, Agda warns that it cannot infer a proof of $\bot$.

As before, the proof-object is only evaluated when inspected; otherwise it behaves as if it were postulated. However, this time it is evaluated efficiently, provided the justification is efficient. Full details are given in Chapter 6.

Chapter 4

Embedded Theories

During the project, two principal theories were embedded using (Oracle + Reflection): Boolean tautology checking (SAT) in Section 4.1 and CTL model-checking in Section 4.2.

SAT was selected because it is an uncomplicated and useful theory. While verifying control systems, it is often the case that obligations of the following form arise:

$$\mathrm{T}\ a_1\ \square_1\ \mathrm{T}\ a_2\ \square_2 \cdots \square_n\ \mathrm{T}\ a_n$$

where $\square_i \in \{\wedge, \vee, \rightarrow\}$ and $a_i : \mathrm{Bool}$. Using a SAT solver to discharge them was an ambition of the project.

On the other hand, it was also beneficial to explore different, more expressive theories. CTL model-checking is one such theory. Notably with regard to this thesis, it allows properties not only concerning safety (as is the case with SAT), but also liveness to be investigated of the control system.

Subsequently, CTL model-checking was later refined to symbolic CTL model-checking (see Section 4.3); and in turn symbolic CTL was refined to provide an interface for verifying ladder logic programs in Section 9.4.1, but this last refinement will be presented after ladder logic programs have been introduced in Chapter 9.

4.1 SAT

In this thesis, standard Boolean satisfiability is not applied; instead tautology testing of an arbitrary Boolean formula with variables is explored, which is an equivalent problem. Tautology checking was chosen over satisfiability testing because SAT verification typically relates to checking safety properties, that is, that something undesirable will never happen.

In the case of SAT, the model contains no information, and for completeness, it is the canonical element from a singleton set. The environment

assigns Boolean values to the variables in the formula. Before introducing the definition of Boolean formulæ with variables, the notion of finite sets are introduced, they are used to index the variables. Finite, or enumeration sets, $\mathrm{Fin} : (n : \mathbb{N}) \to \mathrm{Set}$ are sets with n distinct elements, namely $\mathrm{Fin}\ n \coloneqq \{0, \ldots, n-1\}$; notably $\mathrm{Fin}\ 0 \equiv \varnothing$. Finite sets are defined in Agda as follows:

$$\begin{aligned}&\mathrm{data}\ \mathrm{Fin} : \mathbb{N} \to \mathrm{Set}\ \mathrm{where}\\ &\quad \mathrm{zero}\ :\ \forall\{n\}\ .\ \mathrm{Fin}\ (\mathrm{suc}\ n)\\ &\quad \mathrm{suc}\ \ :\ \forall\{n\}\ .\ \mathrm{Fin}\ n \to \mathrm{Fin}\ (\mathrm{suc}\ n)\end{aligned}$$

Boolean formulæ are defined in module `Boolean.Formula` of Appendix F as follows:

$$\begin{aligned}&\mathrm{data}\ \mathrm{BooleanFormula}\ (n : \mathbb{N}) : \mathrm{Set}\ \mathrm{where}\\ &\quad \mathrm{const}\ \ :\ \mathrm{Bool} \to \mathrm{BooleanFormula}\ n\\ &\quad \mathrm{var}\ \ :\ \mathrm{Fin}\ n \to \mathrm{BooleanFormula}\ n\\ &\quad \neg\ \ :\ \mathrm{BooleanFormula}\ n \to \mathrm{BooleanFormula}\ n\\ &\quad \_\wedge\_\ \_\vee\_\ \_\Rightarrow\_\ :\ \mathrm{BooleanFormula}\ n \to \mathrm{BooleanFormula}\ n\\ &\qquad\qquad\qquad\qquad\qquad\qquad \to \mathrm{BooleanFormula}\ n\end{aligned}$$

Here the underscores ($\_$) denote syntactic positions of required arguments. In the following we write x_n for $(\mathrm{var}\ n)$.

By the Curry-Howard correspondence [Cur34, CFCC58, How80], the semantics of $\varphi : \mathrm{BooleanFormula}\ n$ with respect to an environment $\xi : \mathrm{Fin}\ n \to \mathrm{Bool}$ is given by the Agda type $[\![\ \varphi\ ]\!]_\xi$, which is built from $\top$ and $\bot$ by $\times$, $+$ and $\to$. It is defined as follows:

$$\begin{aligned}&[\![\ \_\ ]\!]_\_ : \forall\{n\} \to \mathrm{BooleanFormula}\ n \to (\mathrm{Fin}\ n \to \mathrm{Bool}) \to \mathrm{Set}\\ &[\![\ \mathrm{const}\ b\ ]\!]_\xi = \mathrm{T}\ b \qquad\qquad [\![\ \varphi \wedge \psi\ ]\!]_\xi = [\![\ \varphi\ ]\!]_\xi \times [\![\ \psi\ ]\!]_\xi\\ &[\![\ \ x_i\ \ ]\!]_\xi = \mathrm{T}\ (\xi\ i) \qquad\qquad [\![\ \varphi \vee \psi\ ]\!]_\xi = [\![\ \varphi\ ]\!]_\xi + [\![\ \psi\ ]\!]_\xi\\ &[\![\ \ \neg\varphi\ \ ]\!]_\xi = [\![\ \varphi\ ]\!]_\xi \to \bot \qquad\qquad [\![\ \varphi \Rightarrow \psi\ ]\!]_\xi = [\![\ \varphi\ ]\!]_\xi \to [\![\ \psi\ ]\!]_\xi\end{aligned}$$

Here $\times$, $+$ and $\to$ are the product, tagged union, and Agda function types, respectively. See Section 1.5 for more information regarding Agda formulæ.

To define the decision procedure, first, define the function

$$\mathrm{instantiate} : \{n : \mathbb{N}\} \to \mathrm{BooleanFormula}\ (\mathrm{suc}\ n) \to \mathrm{Bool} \to \mathrm{BooleanFormula}\ n$$

that instantiates all occurrences of x_0 with the second (Boolean valued) argument; all other variables are shifted down by one, i.e. $x_{n+1} \mapsto x_n$.

Lemma 4.1.1 (instantiate)**.** *Assume* $n : \mathbb{N}$, $\varphi : \mathrm{BooleanFormula}\ n$ *and* $\xi : \mathrm{Fin}\ n \to \mathrm{Bool}$*, then the following holds*

$$[\![\ \varphi\ ]\!]_\xi \leftrightarrow [\![\ \mathrm{instantiate}\ \varphi\ (\xi\ 0)\ ]\!]_{(\xi \circ \mathrm{suc})}$$

Proof. Proof is by induction on φ, the only case considered here is when $\varphi = x_i$ as the other cases follow from the induction hypothesis. Note that this case implies that $n > 0$. The obligation to prove is

$$\mathrm{T}\ (\xi\ i) \leftrightarrow [\![\ \text{instantiate}\ x_i\ (\xi\ 0)\ ]\!]_{(\xi \circ \mathrm{suc})}$$

which follows by side induction on i. There are two further cases to consider:

case $i = 0$**:** Trivial as the obligation unfolds to $\mathrm{T}\ (\xi\ 0) \leftrightarrow \mathrm{T}\ (\xi\ 0)$

case $i = \mathrm{suc}\ i'$**:** Also trivial as the obligation unfolds to

$$\mathrm{T}\ (\xi\ (\mathrm{suc}\ i')) \leftrightarrow \mathrm{T}\ ((\xi \circ \mathrm{suc})\ i')$$

□

The decision procedure for tautology checking a BooleanFormula n is defined naïvely when compared to using the Davis-Putnam approach [DPR61]. The reason is for this naïvety is that it simplifies the correctness proof. It is defined by 2^n applications of instantiate, then canonically with respect to the logical operators.

$$\begin{array}{lllll}
\text{tautology} : \forall n \to \text{BooleanFormula}\ n \to \text{Bool} & & & & \\
\text{tautology zero} & (\text{const}\ b) & = & b & \\
\text{tautology zero} & (\neg\ \varphi) & = & \neg(\text{tautology}\ \varphi) & \\
\text{tautology zero} & (\varphi \square \psi) & = & (\text{tautology zero}\ \varphi) \square (\text{tautology zero}\ \psi) & \\
\text{tautology}\ (\mathrm{suc}\ n)\ \varphi & & = & \text{tautology}\ n\ (\text{instantiate}\ \varphi\ \text{true}) \wedge & \\
 & & & \quad \text{tautology}\ n\ (\text{instantiate}\ \varphi\ \text{false}) &
\end{array}$$

Here $\square \in \{\wedge, \vee, \Rightarrow\}$, and when $\square$ is on the left-hand side, it is a constructor of BooleanFormula and when on the right-hand side is a Boolean function.

Theorem 4.1.2 (tautology-correct)**.**
Assume $n : \mathbb{N}$ *and* $\varphi : \text{BooleanFormula}\ n$*, then the following holds*

$$\mathrm{T}\ (\text{tautology}\ n\ \varphi) \leftrightarrow ((\xi : \mathrm{Fin}\ n \to \text{Bool}) \to [\![\ \varphi\ ]\!]_\xi)$$

Proof. Proof follows by induction over n, and applying Lemma 4.1.1 in the inductive step. □

The above embedding of Boolean tautology checking has been implemented in Agda requiring 39 lines of code for the decision procedure and dependent definitions (including natural numbers and Booleans). The proof of correctness requires an additional ≈ 100 lines of code which includes several basic lemmata about products, sums and the function-type (Curry-Howard isomorphism). The decision procedure is then overridden by a call to an external SAT solver. See Section 5.4.1 for a discussion of the results. See module `Boolean.SatSolver` in Appendix F for the Agda code of this solver.

4.1.1 Use of Non-Dependent Functions

During the initial phase of the project, the SAT algorithm used was a non-dependent version of the one presented here. This was for technical reasons about linking dependently typed Agda functions to non-dependently typed Haskell implementations; the restriction was lifted with the advent of pseudo built-ins. These are special built-ins that are not overridden, but instead known to the type-checker, and must fulfil a number of axioms. This meant that the problematic functions that have dependently typed results, such as instantiate, which are not required when calling the external tool, are not overridden. See Section 5.2.1 for more information.

The non-dependent version required that formulæ are ranked by the index of the greatest free variable it contains. This ranking is used to bound the recursion of the function tautology. The environment would be a map from $\mathbb{N}$ into Bool given by a list of Booleans where true values are assumed for out-of-bound indices. Also, the correctness proof is slightly more involved as it has to deal with lists that are shorter/equal/longer than the formula's rank.

Remark

Later parts of this thesis make use of a non-dependently typed version of these definitions (i.e. Boolean formulæ and semantics); this is for efficiency and usability reasons. Non-dependent Boolean formulæ type-check significantly faster than the dependent versions as the type-checker does not need to be concerned with inferring hidden parameters, especially in the case of the finite variable indices. However, the decision procedure is derived from the dependently typed version presented here by first computing the formula's rank.

4.2 CTL Model-Checking

The model-checking problem as introduced by Clark et al. [CGP99] is described as follows: "given a Kripke structure $\mathcal{M} = (S, \rightarrow, L)$ that represents a finite-state concurrent system and a temporal logic formula φ expressing some desired specification, find the set of all states in S that satisfy φ." Note: the transition relation in a Kripke structure is left-total, i.e. $\forall s \exists t \,.\, (s, t) \in \rightarrow$.

The temporal logic formulæ considered in this thesis are known as Computation Tree Logic (CTL) formulæ, and correspond to statements intuitively

wanted when verifying programs, for example: eventually P will hold, Q always holds. There are many variants of CTL model-checking, for instance plain, symbolic, and SAT-based. There are also different syntactic variants of the CTL formulæ, although semantically equivalent. The variant presented in this section is plain CTL model-checking that has a minimal set of combined CTL connectives. That is the exists next ($EX\ \varphi$), exists globally ($EG\ \varphi$) and exists until ($E[\varphi_1 U \varphi_2]$) connectives. Exists next means that there exists a next state (by $\rightarrow$) such that φ holds. Exists globally means that there exists a sequence of states $\langle s_1 \rightarrow s_2 \rightarrow \ldots \rangle$, such that φ holds for each state. Exists until means that there exists a sequence of states $\langle s_1 \rightarrow s_2 \rightarrow \ldots \rangle$, such that φ_2 holds for some state s_k, and for all $j < k$, φ_1 holds for s_j. See [HR04] for more information about this variant. Subsequently CTL model-checking will be formalised in Agda, but first the semantics is provided using standard mathematics. Let $\mathcal{M}$ be as above, s be a state in $\mathcal{M}$ and $\varphi, \varphi_1, \varphi_2$ be CTL formulæ. The semantics is as follows:

$$\Big((\mathcal{M}, s) \vDash \top\Big) \wedge \Big((\mathcal{M}, s) \nvDash \bot\Big)$$

$$\Big((\mathcal{M}, s) \vDash p\Big) \Leftrightarrow \Big(p \in L(s)\Big)$$

$$\Big((\mathcal{M}, s) \vDash \neg\varphi\Big) \Leftrightarrow \Big((\mathcal{M}, s) \nvDash \varphi\Big)$$

$$\Big((\mathcal{M}, s) \vDash \varphi_1 \wedge \varphi_2\Big) \Leftrightarrow \Big(\big((\mathcal{M}, s) \vDash \varphi_1\big) \wedge \big((\mathcal{M}, s) \vDash \varphi_2\big)\Big)$$

$$\Big((\mathcal{M}, s) \vDash \varphi_1 \vee \varphi_2\Big) \Leftrightarrow \Big(\big((\mathcal{M}, s) \vDash \varphi_1\big) \vee \big((\mathcal{M}, s) \vDash \varphi_2\big)\Big)$$

$$\Big((\mathcal{M}, s) \vDash \varphi_1 \Rightarrow \varphi_2\Big) \Leftrightarrow \Big(\big((\mathcal{M}, s) \nvDash \varphi_1\big) \vee \big((\mathcal{M}, s) \vDash \varphi_2\big)\Big)$$

$$\Big((\mathcal{M}, s) \vDash EX\varphi\Big) \Leftrightarrow \Big(\exists\langle s \rightarrow s_1\rangle\big((\mathcal{M}, s_1) \vDash \varphi\big)\Big)$$

$$\Big((\mathcal{M}, s) \vDash EG\varphi\Big) \Leftrightarrow \Big(\exists\langle s_1 \rightarrow s_2 \rightarrow \ldots\rangle(s = s_1)\forall i\big((\mathcal{M}, s_i) \vDash \varphi\big)\Big)$$

$$\Big((\mathcal{M}, s) \vDash E[\varphi_1 U \varphi_2]\Big) \Leftrightarrow \Big(\exists\langle s_1 \rightarrow s_2 \rightarrow \ldots\rangle(s = s_1) \wedge \exists i\Big(\big((\mathcal{M}, s_i) \vDash \varphi_2\big) \wedge \big(\forall(j < i)(\mathcal{M}, s_j) \vDash \varphi_1\big)\Big)\Big)$$

This variation was chosen as it is the easiest to implement and prove correct in Agda. Plain model-checking is inherently the least efficient variant because the model to be checked looses much of the structure of the original problem it represents. In the subsequent section (4.3), the definitions in this section are lifted to symbolic model-checking. The aim of this improved representation is to preserve more structure of the original problem that the model represents,

and increase usability.

Another approach would have been to use SAT-based bounded model-checking. SAT-based model-checking crafts a number of propositional formulæ that represent the problem as a SAT problem using k-induction[1]. The benefit of SAT-based model-checking is that the resulting decision procedures are more efficient, particularly when a counterexample exists after a small number of steps, than traditional model-checking decision procedures.

Remark

Before formally defining CTL model-checking in Agda, a comment is made about the following definitions. When dealing with decision procedures that have substantial structure, such as model-checking (that is defined over a transition system), it is essential to choose definitions that simplify the embedding process. Moreover, the transitions systems used in this section are finite and can deadlock, i.e. the transition relation is not total. Whereas, the definition of CTL model-checking given above relies upon deadlock-free transition systems. This design decision was taken to simplify the process of defining the transition relation in Agda.

To account for the non-total transition relation, in the following definitions, an additional condition is placed on the semantics of all CTL path. That is, they operate on infinite runs. This is noticeable in the formalisation of the *EX* operator, where above only one-step is considered, but below, one-step into an infinite path is considered, thus preserving deadlock freedom.

Efficient implementations of CTL model-checking decision procedures are also defined over deadlock-free transition systems. Therefore, when executing the external model-checker, the transition relation is made total by adding a *sink* state, and a transition from each state into this sink state. See Section 4.2.2 for more information.

Finite state machines (FSM) are the transition systems used in this work. They are defined by the number of states, the number of (global) atomic propositions, an initial state, a transition relation between states and a labelling of the states. The transition relation is given by two functions: *arrow* which determines for each state the number of transitions from it, and *transition* which determines for each state and arrow from this state, the successor state. The FSM's are defined in module `CTL.TransitionSystem`

[1] $P_0 \wedge \ldots P_{k-1} \wedge (\forall n \,.\, P_n \wedge \cdots \wedge P_{n+k-1} \to P_{n+k}) \to \forall n \,.\, P_n$

of Appendix F as follows:

$$
\begin{array}{l}
\text{record FSM} : \text{Set where} \\
\quad \text{constructor} \\
\quad\quad \text{fsm} \\
\quad \text{field} \\
\quad\quad \begin{array}{ll}
\text{state} & : \mathbb{N} \\
\text{atom} & : \mathbb{N} \\
\text{arrow} & : \text{Fin state} \to \mathbb{N} \\
\text{initial} & : \text{Fin state} \\
\text{transition} & : (s : \text{Fin state}) \to \text{Fin (arrow } s) \to \text{Fin state} \\
\text{label} & : \text{Fin state} \to \text{Fin atom} \to \text{Bool}
\end{array}
\end{array}
$$

The model of the CTL model-checking is a pair consisting of the transition system $\mathcal{M}$ and current state s_0. The environment (under combined operators) is an infinite run $< s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \cdots >$ rooted at the current state s_0, where $a_i : \text{Fin (arrow}_{\mathcal{M}}\ s_i)$; defined by means of the co-algebraic data-type:

$$
\begin{array}{l}
\text{data Run}_{\mathcal{M}}\ (s : \text{Fin state}_{\mathcal{M}}) : \text{Set where} \\
\quad \text{next} : (a : \text{Fin (arrow}_{\mathcal{M}}\ s)) \to \infty\, \text{Run}_{\mathcal{M}}\ (\text{transition}_{\mathcal{M}}\ s\ a) \to \text{Run}_{\mathcal{M}}\ s
\end{array}
$$

Here, ∞ prefixes a term that can potentially be unfolded infinitely many times, see Section 1.5 for more information.

In the following, run_i is the i^{th} state in $run = < s_0 \overset{a_0}{\to} s_1 \overset{a_1}{\to} \cdots s_i \overset{a_i}{\to} \cdots >$. π is used for finite paths and π_i is the i^{th} state in π.

CTL formulæ depend upon the number of atom propositions in the model and are defined using a minimal set of combined CTL operators [HR04]: EX - exists next, EG - exists globally, E[_U_] - exists until and P - state proposition. CTL formulæ are defined in module `CTL.Definition` of Appendix F as follows:

$$
\begin{array}{l}
\text{data CTL}\ (n : \mathbb{N}) : \text{Set where} \\
\quad \begin{array}{ll}
\text{false} & : \text{CTL}_n \\
\neg & : \text{CTL}_n \to \text{CTL}_n \\
\_\vee\_ & : \text{CTL}_n \to \text{CTL}_n \to \text{CTL}_n \\
\_\wedge\_ & : \text{CTL}_n \to \text{CTL}_n \to \text{CTL}_n \\
\text{P} & : \text{Fin } n \to \text{CTL}_n \\
\text{EX} & : \text{CTL}_n \to \text{CTL}_n \\
\text{EG} & : \text{CTL}_n \to \text{CTL}_n \\
\text{E[\_U\_]} & : \text{CTL}_n \to \text{CTL}_n \to \text{CTL}_n
\end{array}
\end{array}
$$

The semantics of a CTL formula is as follows: false, $\neg$, _$\vee$_ and _$\wedge$_ are the same as in propositional logic. P a has the meaning that atomic proposition

a holds in the current state. The remaining cases specify properties about infinite runs of the transition system rooted at some state s. Exists next (EX φ) holds when there exists a run *run* from s such that φ holds at run_1; a run is required here because the transition system can dead-lock. Exists globally (EG φ) holds when there exists a run from s such that at each point i on the run, φ holds. Exists until (E[φ U ψ]) holds when there exists a run from s such that there exists a point k where ψ holds, and for all points $j < k$, φ holds.

The semantics of a CTL formula with respect to a model is formally given as follows:

$$
\begin{aligned}
&[\![\_ , \_ \vDash \_]\!] : (\mathcal{M} : \mathrm{FSM}) \to (\mathrm{Fin}\ \mathrm{state}_{\mathcal{M}}) \to \mathrm{CTL}_{\mathrm{atom}_{\mathcal{M}}} \to \mathrm{Set} \\
&[\![\mathcal{M} , s \vDash \quad \mathrm{false} \quad]\!] = \bot \\
&[\![\mathcal{M} , s \vDash \quad \neg\varphi \quad]\!] = [\![\mathcal{M} , s \vDash \varphi]\!] \to \bot \\
&[\![\mathcal{M} , s \vDash \quad \varphi \vee \psi \quad]\!] = [\![\mathcal{M} , s \vDash \varphi]\!] + [\![\mathcal{M} , s \vDash \psi]\!] \\
&[\![\mathcal{M} , s \vDash \quad \varphi \wedge \psi \quad]\!] = [\![\mathcal{M} , s \vDash \varphi]\!] \times [\![\mathcal{M} , s \vDash \psi]\!] \\
&[\![\mathcal{M} , s \vDash \quad \mathrm{P}\ a \quad]\!] = \mathrm{T}\ (\mathrm{label}_{\mathcal{M}}\ s\ a) \\
&[\![\mathcal{M} , s \vDash \quad \mathrm{EX}\ \varphi \quad]\!] = \exists\ (run : \mathrm{Run}_{\mathcal{M}}\ s)\ [\![\mathcal{M} , run_1 \vDash \varphi]\!] \\
&[\![\mathcal{M} , s \vDash \quad \mathrm{EG}\ \varphi \quad]\!] = \exists\ (run : \mathrm{Run}_{\mathcal{M}}\ s)\ (\forall i\ .\ [\![\mathcal{M} , run_i \vDash \varphi]\!]) \\
&[\![\mathcal{M} , s \vDash \mathrm{E}[\ \varphi\ \mathrm{U}\ \psi\ ]]\!] = \exists\ (run : \mathrm{Run}_{\mathcal{M}}\ s)\ \exists\ (k : \mathbb{N})\ (\\
&\qquad\qquad (\forall j\ .\ j < k \to [\![\mathcal{M} , run_j \vDash \varphi]\!]) \\
&\qquad\qquad \times [\![\mathcal{M} , run_k \vDash \psi]\!])
\end{aligned}
$$

Here, $\mathrm{label}_{\mathcal{M}}$ is *label* projected from $\mathcal{M}$, and the environment ($\mathrm{Run}_{\mathcal{M}}\ s$) is existentially quantified.

Determining whether a CTL formula holds in the logical cases is canonical with respect to the connectives and is not discussed further. The decision procedure for the first substantial connective, EX φ (exists next), is implemented by searching for a path of length $\mathrm{state}_{\mathcal{M}} + 1$ and checking that φ holds at the second point, i.e. there exists a successor state where φ holds. The argument for the correctness of this procedure is a simpler case of the correctness proof for exists globally and thus follows by Lemma 4.2.1.

In the case of EG φ (exists globally), an infinite run is required such that, at each point on this run, φ holds. Naïvely, checking each point on this run would take an infinite amount of time; thus, the problem is finitised.

By the pigeonhole principle[2] [Ded63] (which underlies the proof of the pumping lemma [BHPS64]) and the finiteness of the transition system, the decision procedure for EG only checks for a finite path of fixed length from the state s such that φ always holds. If a path π of length $\mathrm{state}_{\mathcal{M}} + 1$ exists

[2]The pigeonhole principle states: *if you put n things into m boxes where $n > m$, then there exists at least one box that contains more than one thing.*

from s such that φ holds at each point, then it can be extended infinitely many times into a run. This follows by Lemma 4.2.1.

Lemma 4.2.1 ($\mathcal{M}, s \vDash \mathrm{EG}\ \varphi$)**.** *Assume a finite transition system $\mathcal{M}$ with n states. There exists an infinite run from state s such that φ holds at each point iff there exists a path π of length $n+1$ from state s such that φ holds at each point on π.*

Proof.

$\Rightarrow$ An infinite run where φ holds at each point can be truncated to a path of length $n+1$.

$\Leftarrow$ By the pigeonhole principle, at least one state has been repeated in π, i.e. $\exists(0 \leq i < j \leq n)\ .\ \pi_i = \pi_j$. Therefore a (possibly trivial) loop in the transition system exists containing π_i, this loop can be repeated infinitely many times, and we obtain an infinite run.

□

In the case of exists until, things are a little more complicated. E[φ U ψ] means there exists an infinite run *run* such that at some point k in the future ψ must hold, but up to and not including that point, φ must hold. Intuitively, the decision procedure checks for a path π^φ with length $\leq \mathrm{state}_\mathcal{M}$ such that φ holds at each point, and then checks for a path π^ψ of length $\mathrm{state}_\mathcal{M}+1$ starting at the end of π^φ such that ψ holds at π^ψ_1. This follows by Lemma 4.2.2.

Lemma 4.2.2 ($\mathcal{M}, s \vDash \mathrm{E}[\ \varphi\ \mathrm{U}\ \psi\ ]$)**.** *Assume a finite transition system $\mathcal{M}$ with n states. $\mathcal{M}, s \vDash \mathrm{E}[\ \varphi\ \mathrm{U}\ \psi\ ]$ holds iff there exists a path π^φ with length $\leq n$ from the state s such that φ holds at each point of π^φ, and there exists a path π^ψ of length $n+1$ such that the end of π^φ equals the beginning of π^ψ and ψ holds at π^ψ_1.*

Proof.

$\Rightarrow$ There exists a point k on the infinite run *run*, such that for all points $j < k$, φ holds and at point k, ψ holds. We show that π^φ and π^ψ exist by course-of-value induction on k:

case $k \leq n$**:** We are done, π^φ is a prefix of the run, and π^ψ equals the succeeding $n+1$ states from k.

case $k > n$**:** By the pigeonhole principle there exists two points $0 \leq l < m < n+1$ such that $run_l = run_m$. Therefore a loop exists before point k, this loop can be removed such that φ holds up to point $k-(m-l)$ and ψ holds at point $k-(m-l)$. Let run' be the resulting run. By the induction hypothesis and run', the assertion follows.

⇐ By Lemma 4.2.1, path π^{ψ} can be extended infinitely many times, thus an infinite run can be constructed consisting of π^{ψ} extended infinitely many times concatenated to π^{φ}. As φ holds along π^{φ}, and ψ holds at π^{ψ}_1, the infinite run satisfies φ until ψ.

□

Corollary. *Following by Lemma 4.2.1 and Lemma 4.2.2, the decision procedures for* EG *and* E[_U_] *can be implemented by bounded traversals of the transition system and taking disjunctions between choice points in the traversals. The decision procedure for* EX *is a simplified case of* EG*. Let*

$$\text{ctlcheck} : (\mathcal{M} : \text{FSM}) \to (\text{Fin state}_{\mathcal{M}}) \to \text{CTL}_{\text{atom}_{\mathcal{M}}} \to \text{Bool}$$

be the obtained decision procedure.

See module `CTL.DecProc` in Appendix F for the implementation of the decision procedure, and module `CTL.Proof` for the correctness proof of the decision procedure.

The implementation used here requires ≈75 lines of Agda code, which includes definitions of Bool, ℕ, Fin, transition system, CTL formulæ and the decision procedure. Proof of correctness requires > 1000 lines of code. This includes the proof of the pigeonhole principle (≈ 300 lines of code) and many lemmata reasoning about finite sets and the transition system.

4.2.1 Pigeonhole Principle

The pigeonhole principle states: if you put n items into m holes where $n > m$, then there exists at least one hole that contains more than one item.

Lemma 4.2.3 (Pigeonhole principle)**.** *Assume a function $f : A \to B$ where A and B are finite, and the cardinality of A is greater-than that of B. Then f is not injective. I.e. there exists $a, b \in A$ such that $a \neq b$ and $f\ a = f\ b$.*

Remark

Although the pigeonhole principle is canonical for a human to understand to be correct, the proof in Agda is non-trivial.

Proof. Induction on the cardinality of A, and side induction on the cardinality of B. Without loss of generality assume $A = \text{Fin } n$ and $B = \text{Fin } m$, such that $n > m$. The base cases where $n \in \{0, 1\}$ are trivial because no such f exists. The inductive step remains where $n = n' + 1$, there are three cases. Note that $f\ 0 \in \text{Fin } m$, therefore $m = m' + 1$.

Case 1: $\exists (a\ b : \text{Fin } n)\ .\ a \neq b,\ f\ a = f\ b = 0$. We are done.

Case 2: $\exists (a : \text{Fin } n)\ .\ f\ a = 0,\ (\forall (b : \text{Fin } n)\ .\ b \neq a \to f\ b \neq 0)$. Without loss of generality assume $a = 0$. Let

$$\begin{aligned} &g : \text{Fin } n' \to \text{Fin } m' \\ &g\ x\ =\ f\ (x + 1) - 1 \end{aligned}$$

By the main induction hypothesis, there exists $a', b' : \text{Fin } n'$ such that $a' \neq b'$, $g(a') = g(b')$. Since $f\ (a' + 1) > 0$ and $f\ (b' + 1) > 0$, we get $f\ (a' + 1) = f\ (b' + 1)$.

Case 3: $\forall (a : \text{Fin } n)\ .\ f\ a \neq 0$. Let

$$\begin{aligned} &g : \text{Fin } n \to \text{Fin } m' \\ &g\ x\ =\ (f\ x) - 1 \end{aligned}$$

By side induction hypothesis, there exists $a', b' \in \text{Fin } n$ such that $a' \neq b'$, $g\ a' = g\ b'$. Therefore $f\ a = f\ b$.

□

This proof searches for the two items that have been placed into the same hole by consistently mutating f until the two items have been placed into hole 0.

See module `Data.Fin.Pigeon` in Appendix F for the full formalisation of the above proof.

4.2.2 CTLSink

The CTL theory presented allows for the transition systems to dead-lock, but in practice the external model-checkers used require that the transition system is dead-lock free by enforcing that the transition relation is total. This is achieved by implicitly transforming the input passed to the tool by adding a sink state to the FSM and adjusting the formulæ accordingly. A sink (or error) state is one that is reachable from every state (including the sink) in one transition. A new proposition is added that only holds in the sink

state so that the formulæ can identify the sink state. Thus, the formulæ are translated to prevent the characterised runs from passing-through the sink state. All definitions in this section are in module `CTL.Sink` of Appendix F.

The CTL model and formula are translated into SinkCTL by two functions: first, mkSink, translates a model, and secondly liftCTL, translates a formula.

$$
\begin{aligned}
&\text{mkSink} : \text{FSM} \to \text{FSM} \\
&\text{mkSink}\ \mathcal{M} = \text{fsm}\ (\text{suc}\ \text{state}_{\mathcal{M}}) \\
&\qquad\qquad (\text{suc}\ \text{atom}_{\mathcal{M}}) \\
&\qquad\qquad \{0 \mapsto 1\ ;\ \text{suc}\ n \mapsto \text{suc}\ (\text{arrow}_{\mathcal{M}}\ n)\} \\
&\qquad\qquad (\text{suc}\ \text{initial}_{\mathcal{M}}) \\
&\qquad\qquad \left\{ s\ a \mapsto \begin{cases} 0 & \textbf{if}\ s = 0 \vee a = 0 \\ \text{suc}\ (\text{transition}_{\mathcal{M}}\ (s-1)\ (a-1)) & \textbf{otherwise} \end{cases} \right\} \\
&\qquad\qquad \left\{ s\ p \mapsto \begin{cases} \text{label}_{\mathcal{M}}\ (s-1)\ (p-1) & \textbf{if}\ s \neq 0 \wedge p \neq 0 \\ s = 0 \wedge p = 0 & \textbf{otherwise} \end{cases} \right\}
\end{aligned}
$$

The result of the above function is the following transformation:

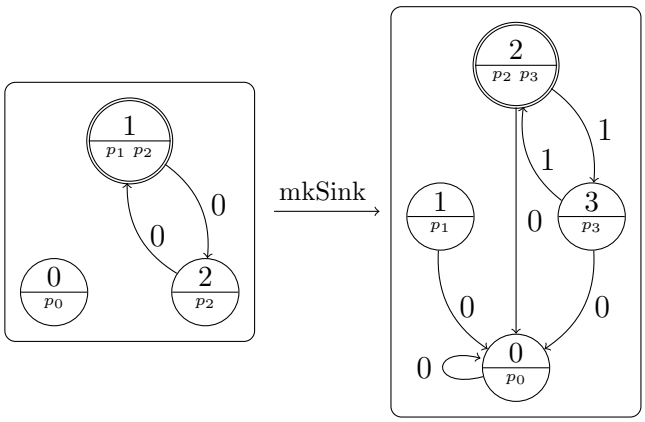


Figure 4.1: CTL Sink State

Let sink = P 0, the proposition that only holds in the sink state. Formulæ

are lifted by the following function:

$$
\begin{array}{lcl}
\text{liftCTL} : \forall\{n\}\ .\ \text{CTL}_n \to \text{CTL}_{(\text{suc}\ n)} & & \\
\text{liftCTL false} & = & \text{false} \\
\text{liftCTL}\ \neg\varphi & = & \neg(\text{liftCTL}\ \varphi) \\
\text{liftCTL}\ (\varphi \vee \psi) & = & (\text{liftCTL}\ \varphi) \vee (\text{liftCTL}\ \psi) \\
\text{liftCTL}\ (\varphi \wedge \psi) & = & (\text{liftCTL}\ \varphi) \wedge (\text{liftCTL}\ \psi) \\
\text{liftCTL}\ (\text{P}\ x) & = & \text{P}\ (\text{suc}\ x) \\
\text{liftCTL}\ (\text{EX}\ \varphi) & = & \text{EX}\ ((\text{liftCTL}\ \varphi) \wedge \text{EG}\ \neg\,\text{sink}) \\
\text{liftCTL}\ (\text{EG}\ \varphi) & = & \text{EG}\ ((\text{liftCTL}\ \varphi) \wedge \neg\,\text{sink}) \\
\text{liftCTL}\ \text{E}[\ \varphi\ \text{U}\ \psi\ ] & = & \text{E}[\ (\text{liftCTL}\ \varphi) \wedge \neg\,\text{sink}\ \text{U}\ (\text{liftCTL}\ \psi) \wedge \text{EG}\ \neg\,\text{sink}\ ]
\end{array}
$$

The formulæ are translated such that they carve out the original structure of the model in the translated model. In the cases of the CTL connectives it is essential that the quantified runs never pass-through the sink state as this does not exist in the original model. EX is an example of this as it requires that a run exists such that in the second state of the run φ holds; the translation of EX requires that from the second state where φ holds there also exists a run such that it never passes-through the sink state. Note that this does not apply to the first state of the run, as this is enforced by Theorem 4.2.4 below and would be a truism otherwise. The translation of EG is canonical as it restricts the run to never pass-through the sink state. The final operator $\text{E}[\ \varphi\ \text{U}\ \psi\ ]$ splits the carved out run into two runs, the first is a run that does not pass-through the sink state (and φ holds) up to point k where ψ holds, then it is unconstrained. The second run starts from point k in the first run, and it never passes-through the sink state.

Remark

In the translation of $\text{E}[\ \varphi\ \text{U}\ \psi\ ]$, the constraint that the first run never passes-through the sink state is not required. This is because once a run enters the sink state there is no possibility of it leaving, so the constraint on the second run to never pass-through the sink state is sufficient to ensure that the first run never passes-through the sink state.

However, explicitly stating that the first run does not pass-through the sink state simplifies the following proof of correctness.

This is not a trivial transformation and if not correct would destroy correctness of this technique. As it is assumed that the external tool is correct and that the input to the tool is correct, it is required to prove that the transformation preserves the correctness of CTL.

Theorem 4.2.4 (SinkCTL)**.** *Assume a* FSM $\mathcal{M}$*, state* s *and formula* φ*, then following holds:*

$$[\![\, \mathcal{M}, s \vDash \varphi \,]\!] \Leftrightarrow [\![\, \text{mkSink}\ \mathcal{M}, \text{suc}\ s \vDash \text{liftCTL}\ \varphi \,]\!]$$

The proof is by induction on φ. The three cases of the logical connectives are trivial and not presented, also in the case that $\varphi = \text{P}\ x$ the proof follows immediately by unfolding the definitions. The remaining cases of the CTL connectives are quantified over runs; thus an equivalence is required between runs of the original and runs of the translated transition systems. The equivalence requires two functions that translate runs between these two systems, then proving the translated runs preserve the semantics of CTL. The first function translates a run from the original transition system into the sinked transition system by applying suc to all the states and actions, i.e.

$$\begin{aligned}
&\text{liftRun}_{\mathcal{M}} : \forall\{s\}\ .\ \text{Run}_{\mathcal{M}}\ s \to \text{Run}_{(\text{mksink}\ \mathcal{M})}\ (\text{suc}\ s) \\
&\text{liftRun}_{\mathcal{M}}\ (\text{next}\ a\ r)\ =\ \text{next}\ (\text{suc}\ a)\ (\sharp\ \text{liftRun}_{\mathcal{M}}\ (\flat\ r))
\end{aligned}$$

The second function translates runs in the other direction (by removing a suc from each state and arrow), however it is required to also have a proof that the run never passes through the sink state.

$$\begin{aligned}
&\text{downRun}_{\mathcal{M}} : \forall\{s\}\ .\ (r : \text{Run}_{(\text{mksink}\ \mathcal{M})}\ (\text{suc}\ s)) \\
&\qquad\qquad \to (\forall n\ .\ [\![\, \text{mksink}\ \mathcal{M}, r_n \vDash \neg\, \text{sink} \,]\!]) \\
&\qquad\qquad\quad \to \text{Run}_{\mathcal{M}}\ s \\
&\text{downRun}_{\mathcal{M}}\ (\text{next zero}\ r)\ p\ =\ \text{efq}\ (p\ 1\ \text{tt}) \\
&\text{downRun}_{\mathcal{M}}\ (\text{next}\ a\ r) \quad p\ =\ \text{next}\ (\text{suc}\ a) \\
&\qquad\qquad\qquad\qquad\qquad (\sharp\ \text{downRun}_{\mathcal{M}}\ (\flat\ r)\ (p \circ \text{suc}))
\end{aligned}$$

The following three lemmata are required to prove the translations are consistent with the semantics of the CTL connectives EX, EG and EU. Only the lemma for EG is proved here as the other proofs are similar, however, as usual the full proofs are in Appendix F.

Lemma 4.2.5 (sink-EG)**.** *Assume a transition system* $\mathcal{M}$ *and two* CTL *formulæ* φ *and* ψ*, such that the following holds*

$$\forall s\ .\ [\![\, \mathcal{M}, s \vDash \varphi \,]\!] \leftrightarrow [\![\, \text{mksink}\ \mathcal{M}, \text{suc}\ s \vDash \psi \,]\!]$$

then the proof can be lifted to the EG connective, i.e.

$$\forall s\ .\ [\![\, \mathcal{M}, s \vDash \text{EG}\ \varphi \,]\!] \leftrightarrow [\![\, \text{mksink}\ \mathcal{M}, \text{suc}\ s \vDash \text{EG}\ (\psi \wedge \neg\, \text{sink}) \,]\!]$$

Proof.

⇒ Assume s and $(r, p) : [\![M, s \vDash \mathrm{EG}\ \varphi]\!]$. Let $r' = \mathrm{liftRun}_{\mathcal{M}}\ r$. Then the proof follows by showing:

$$\forall n\ .\ [\![\mathrm{mksink}\ \mathcal{M}, r'_n \vDash \psi \wedge \neg \mathrm{sink}]\!]$$

Assume n, the left-hand conjunct follows by the assumption

$$\forall t\ .\ [\![\mathcal{M}, t \vDash \varphi]\!] \rightarrow [\![\mathrm{mksink}\ \mathcal{M}, \mathrm{suc}\ t \vDash \psi]\!]$$

applied to r_n. N.B. $r'_n = \mathrm{suc}\ r_n$ by definition of liftRun. The right-hand conjunct follows by the definition of liftRun.

⇐ Assume s and $(r, p) : [\![\mathrm{mksink}\ M, \mathrm{suc}\ s \vDash \mathrm{EG}\ (\psi \wedge \neg \mathrm{sink})]\!]$. Let $r' = \mathrm{downRun}_{\mathcal{M}}\ r\ (\lambda n \rightarrow \pi_1\ (p\ n))$. Then the proof follows by showing:

$$\forall n\ .\ [\![\mathcal{M}, r'_n \vDash \varphi]\!]$$

Assume n, the proof follows by the assumption

$$\forall t\ .\ [\![\mathcal{M}, t \vDash \varphi]\!] \leftarrow [\![\mathrm{mksink}\ \mathcal{M}, \mathrm{suc}\ t \vDash \psi]\!]$$

applied to r'_n. N.B. $\mathrm{suc}\ r'_n = r_n$ by definition of downRun.

□

Lemma 4.2.6 (sink-EX)**.** *Assume a transition system $\mathcal{M}$ and two* CTL *formulæ φ and ψ, such that the following holds*

$$\forall s\ .\ [\![\mathcal{M}, s \vDash \varphi]\!] \leftrightarrow [\![\mathrm{mksink}\ \mathcal{M}, \mathrm{suc}\ s \vDash \psi]\!]$$

then the proof can be lifted to the EX connective, i.e.

$$\forall s\ .\ [\![\mathcal{M}, s \vDash \mathrm{EX}\ \varphi]\!] \leftrightarrow [\![\mathrm{mksink}\ \mathcal{M}, \mathrm{suc}\ s \vDash \mathrm{EX}\ (\psi \wedge \mathrm{EG}\ \neg \mathrm{sink})]\!]$$

Lemma 4.2.7 (sink-EU)**.** *Assume a transition system $\mathcal{M}$ and four* CTL *formulæ φ and ψ, such that the following holds*

$$\forall s\ .\ [\![\mathcal{M}, s \vDash \varphi]\!] \leftrightarrow [\![\mathrm{mksink}\ \mathcal{M}, \mathrm{suc}\ s \vDash \varphi']\!]$$

and

$$\forall s\ .\ [\![\mathcal{M}, s \vDash \psi]\!] \leftrightarrow [\![\mathrm{mksink}\ \mathcal{M}, \mathrm{suc}\ s \vDash \psi']\!]$$

then the proof can be lifted to the EU connective, i.e.

$$\forall s\ .\ [\![\mathcal{M}, s \vDash \mathrm{EG}\ \varphi]\!] \leftrightarrow [\![\mathrm{mksink}\ \mathcal{M}, \mathrm{suc}\ s \vDash \mathrm{EG}\ (\psi \wedge \neg \mathrm{sink})]\!]$$

The proof of Lemma 4.2.6 is a simpler case of Lemma 4.2.5. The proof of Lemma 4.2.7 is similar to the proof of Lemma 4.2.5, neither proofs are presented. The proof of Theorem 4.2.4 then follows by these lemmata.

4.3 Symbolic CTL

As presented, CTL model-checking consists of a state space indexed by the finite set $0, \ldots, n-1$. In this form it is difficult to encode realistic programs; also it is awkward to validate the models and formulæ. In contrast, a typical off-the-shelf model-checker allows for the model to be specified by a finite number of finite typed variables (e.g. enumeration sets), the transition function by a user friendly syntax, and formulæ reason about specific values of specific variables. That is, the input to the model-checker is closer to an actual program, easing the validation process. In this section, it is described how to encode finite structures into numbers, with the objective of defining symbolic CTL model-checking by lifting the definitions of the previous section.

The transition systems that are used for symbolic CTL are defined by a list of natural numbers that define the cardinalities of the types of the variables that define the state space, and associated to each state a list of natural numbers that define the cardinalities of the types of the variables that are required for a transition. An initial state and a transition function, that for a given state and a valid input for that state (sequence of variables), identifies the successor state.

Formulæ over these transition systems are as before (in Section 4.2) with the exception that the connective P a (proposition a holds in current state) is adjusted accordingly to reason about the state variables. I.e. P$[x == y]$ means variable x has value y in the current state.

The remainder of this section describes the embedding of this structure and associated formulæ into the definitions given in Section 4.2.

The basic technique used is to implicitly define a universe of finite types, and show that it is closed under multiplication and addition. See the work of Altenkirch et al. [AMM07] for more information. Furthermore, in Theorem 4.3.4, an isomorphism is defined that embeds a number of finite typed variables into a larger finite type. This embedding will define an isomorphism between the state space of symbolic CTL and CTL model-checking. Similarly, Theorem 4.3.8 defines an isomorphism between the possible values of a variable in a state and the atomic propositions of a CTL transition system.

4.3.1 Pairs

The embedding is built around the basic fact that for two finite sets A and B, their Cartesian product $A \times B$ is equinumerous to $|A| * |B|$.

Theorem 4.3.1 (fin-pair-iso)**.** *Assume* $n\ m : \mathbb{N}$*, an isomorphism exists between* Fin $n \times$ Fin m *and* Fin $(n * m)$*.*

This is proved by defining two structure preserving functions that are inverses of each other. Namely, fin-pair and fin-unpair that will pair two finite numbers, and unpair them, respectively; see module `Data.Fin.Record` in Appendix F for the full definitions.

$$\begin{aligned}&\text{fin-pair} : \forall\ \{n\ m\}\ .\ \text{Fin}\ n \times \text{Fin}\ m \to \text{Fin}\ (n * m)\\&\text{fin-pair}\ \{n\}\ \{m\}\ (x, y) = (m * x) + y\end{aligned}$$

$$\begin{aligned}&\text{fin-unpair} : \forall\ \{n\ m\}\ .\ \text{Fin}\ (n * m) \to \text{Fin}\ n \times \text{Fin}\ m\\&\text{fin-unpair}\ \{n\}\ \{m\}\ z = z\ \text{div}\ m, z\ \text{mod}\ m\end{aligned}$$

These pairing functions are shown to preserve structure:

Lemma 4.3.2 (fin-pair-1)**.**

$$\forall (n\ m : \mathbb{N})\ (x : \text{Fin}\ n)\ (y : \text{Fin}\ m)\ .\ \text{fin-unpair}\ (\text{fin-pair}\ (x, y)) \equiv (x, y)$$

Proof. Assume n, m, x and y. The proof follows by equality reasoning.

$$\begin{aligned}&\text{fin-unpair}\ (\text{fin-pair}\ (x, y)) \equiv (m * x + y)\ \text{div}\ m, (m * x + y)\ \text{mod}\ m\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textit{by definition}\\&\equiv x, y\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textit{by } y < m\end{aligned}$$

□

Lemma 4.3.3 (fin-pair-2)**.**

$$\forall (n\ m : \mathbb{N})\ (z : \text{Fin}\ (n * m))\ .\ \text{fin-pair}\ (\text{fin-unpair}\ z) \equiv z$$

Proof. Assume n, m, x and y. The proof follows by equality reasoning.

$$\begin{aligned}&\text{fin-pair}\ (\text{fin-unpair}\ z) \equiv m * (z\ \text{div}\ m) + (z\ \text{mod}\ m)\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textit{by definition}\\&\equiv z\\&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\textit{by Euclidean division}\end{aligned}$$

□

The proof of Theorem 4.3.1 follows by Lemma 4.3.2 and Lemma 4.3.3.

4.3.2 Finite Records

Finite records build upon finite pairs, such that they will be able to encode the state space and inputs; they consist of a finite number of entries where each entry is finite. Thus, they are n-ary products of finite types.

Theorem 4.3.4 (fin-record-iso)**.** *The following are isomorphic:*

$$\text{Fin } n_0 \times \text{Fin } n_1 \times \cdots \times \text{Fin } n_{i-1}$$

and

$$\text{Fin } (n_0 * n_1 * \cdots * n_{i-1})$$

This will be a corollary of Lemma 4.3.5 below, which is shown in detail since it corresponds to the proof in Agda.

The number of, and cardinalities of the entries in a record are given by a list l of natural numbers. Thus, the above types are inductively defined over l by the following two functions:

$$\text{Record} : \text{List } \mathbb{N} \to \text{Set}$$

Record maps a list $[l_0, l_1, \ldots, l_{i-1}]$ to the type $\text{Fin } l_0 \times \text{Fin } l_1 \times \cdots \times \text{Fin } l_{i-1}$, with the base case $[] \mapsto \top$. The cardinality of Record l is defined by the multiplicative product of l. That is by the function

$$\Pi : \text{List } \mathbb{N} \to \mathbb{N}$$

which computes for a list $[l_0, l_1, \ldots, l_{i-1}]$ the product, given as $l_0 \times l_1 \times \cdots \times l_{i-1}$. Notably, $\Pi\, [] = 1$.

Lemma 4.3.5 (fin-record-iso2)**.** *Assume* $(l : \text{List } \mathbb{N})$*.* Record l *and* $\text{Fin } (\Pi\, l)$ *are isomorphic.*

This is proved by defining two structure preserving functions that are inverses of each other. Namely, encode and decode that lift fin-pair and fin-unpair to n-ary products; see module `Data.Fin.Record` in Appendix F for the full definitions.

$$\begin{aligned}
&\text{encode} : (l : \text{List } \mathbb{N}) \to \text{Record } l \to \text{Fin } (\Pi\, l)\\
&\text{encode } [] \qquad r = r\\
&\text{encode } (l :: ls)\ r = \text{fin-pair } ((\pi_0\ r), \text{encode } ls\ (\pi_1\ r))
\end{aligned}$$

Here the hidden parameters of fin-pair (and fin-unpair below) are l and $\Pi\ ls$.

$$\begin{aligned}
&\text{decode} : (l : \text{List } \mathbb{N}) \to \text{Fin } (\Pi\, l) \to \text{Record } l\\
&\text{decode } [] \qquad z = z\\
&\text{decode } (l :: ls)\ z = \pi_0\ (\text{fin-unpair } z), \text{decode } ls\ (\pi_1\ (\text{fin-unpair } z))
\end{aligned}$$

Often the list indexing these functions will be written in subscript. All that remains is to show that encode and decode are structure preserving.

Lemma 4.3.6 (fin-record-1)**.**

$$\forall (l : \text{List } \mathbb{N})\ (x : \text{Record } l)\ .\ \text{decode } l\ (\text{encode } l\ x) \equiv x$$

Proof. By induction on l

case $l = []$**:** Trivial, $\text{id}\ (\text{id}\ x) \equiv x$.

case $l = (l' :: ls)$**:**

Let $z = \text{encode } l\ x = \text{fin-pair}\ ((\pi_0\ x), \text{encode } ls\ (\pi_1\ x))$.

$$\text{fin-unpair } z \equiv (\pi_0\ x, \text{encode } ls\ (\pi_1\ x)) \qquad (*)$$

by Lemma 4.3.2

$$\begin{aligned}
\text{decode } l\ (\text{encode } l\ x) &\equiv \pi_0\ (\text{fin-unpair } z), \text{decode } ls\ (\pi_1\ (\text{fin-unpair } z)) \\
&\qquad\qquad \textit{by definition} \\
&\equiv \pi_0\ x, \text{decode } ls\ (\text{encode } ls\ (\pi_1\ x)) \\
&\qquad\qquad \textit{rewrite by } (*) \\
&\equiv \pi_0\ x, \pi_1\ x \equiv x \\
&\qquad\qquad \textit{by induction hypothesis and } \eta \textit{ contraction}
\end{aligned}$$

□

Lemma 4.3.7 (fin-record-2)**.**

$$\forall (l : \text{List } \mathbb{N})\ (x : \text{Fin } (\Pi\ l))\ .\ \text{encode } l\ (\text{decode } l\ x) \equiv x$$

Proof.

case $l = []$**:** Trivial, $\text{id}\ (\text{id}\ x) \equiv x$.

case $l = (l' :: ls)$**:**

$$\begin{aligned}
&\text{encode } l\ (\text{decode } l\ x) \\
&\quad \equiv \text{fin-pair}\ (\pi_0\ (\text{fin-unpair } x), \text{encode } ls\ (\text{decode } ls\ (\pi_1\ (\text{fin-unpair } x)))) \\
&\qquad\qquad \textit{by definition} \\
&\quad \equiv \text{fin-pair}\ (\pi_0\ (\text{fin-unpair } x), (\pi_1\ (\text{fin-unpair } x)))
\end{aligned}$$

by induction hypothesis

$\equiv$ fin-pair (fin-unpair x) $\equiv x$

by η contraction and Lemma 4.3.3

□

The proof of Lemma 4.3.5 then follows by Lemmata 4.3.6 and 4.3.7. Also, the proof of Theorem 4.3.4 follows, as it is a corollary of Lemma 4.3.5.

4.3.3 State Space

We want to embed a compound state that consists of the typed variables

$$a_0 : A_0,\ a_1 : A_1,\ \ldots,\ a_{i-1} : A_{i-1}$$

where $A_0, A_1, \ldots, A_{i-1}$ are finite types. It is trivial to define an isomorphism between A_j and Fin $|A_j|$.

Thus, the compound state can be viewed as a finite product of the form:

$$\text{Fin}\,|A_0| \times \text{Fin}\,|A_1| \times \cdots \times \text{Fin}\,|A_{i-1}|$$

By Theorem 4.3.4 this is isomorphic to the finite type:

$$\text{Fin}\,(|A_0| * |A_1| * \cdots * |A_{i-1}|)$$

Atomic Propositions

In this setting, each symbolic state s contains i variables. The propositions are equality tests on these variables, i.e. $a_j == y : \text{Fin}\,|A_j|$, and are given by a pair of type $(x : \text{Fin}\,(\text{length}\ l)) \times \text{Fin}\ l_x$. Therefore, in the corresponding FSM, each variable $a_j : \text{Fin}\,|A_j|$ induces $|A_j|$ atomic propositions, one for each value that the variable can take. In total, the corresponding FSM has $\Sigma_{0 \le j < i} |A_j|$ atomic propositions.

Remark

There are redundancies on the propositions in the corresponding FSM. It is not possible that $a_j == x$ and $a_j == y$ hold in the same state. This is because when later translating a symbolic FSM into an FSM, the proposition $a_j == z$ holds *iff* a_j has value z in the original (symbolic) state.

In the following let Σ be the standard summation of natural numbers, that is $\Sigma\,[\,] = 0$ and $\Sigma\,[a_1, a_2, \cdots, a_n] = a_1 + a_2 + \cdots + a_n$.

It is now shown that this enumeration of variables and values is isomorphic to the product of a variable a_x and a value $y : A_x$ from its domain.

Theorem 4.3.8 (fin-enum)**.** *Assume a list l of $\mathbb{N}$, then* $\mathrm{Fin}\,(\Sigma\,l)$ *is isomorphic to* $(x : \mathrm{Fin}\,(\mathrm{length}\,l)) \times \mathrm{Fin}\,l_x$

This is shown by defining two inverse functions encode-Σ and decode-Σ, then proving that they define an isomorphism. See module `Data.Fin.Record` in Appendix F for the full definitions.

$$
\begin{array}{l}
\text{encode-}\Sigma : \forall l\,.\,(x : \mathrm{Fin}\,(\mathrm{length}\,l)) \times \mathrm{Fin}\,l_x \to \mathrm{Fin}\,(\Sigma\,l) \\
\text{encode-}\Sigma\;[\,]\quad\;\;(x, y)\quad\;\; = \mathrm{efq}\,x \\
\text{encode-}\Sigma\;(l_0 :: l')\quad(0, y)\quad\;\; = y \\
\text{encode-}\Sigma\;(l_0 :: l')\quad(\mathrm{suc}\,x, y) = l_0 + (\text{encode-}\Sigma\;l'\;(x, y))
\end{array}
$$

$$
\begin{array}{l}
\text{decode-}\Sigma : \forall l\,.\;\mathrm{Fin}\,(\Sigma\,l) \to (x : \mathrm{Fin}\,(\mathrm{length}\,l)) \times \mathrm{Fin}\,l_x \\
\text{decode-}\Sigma\;[\,]\quad\;\; z = \mathrm{efq}\,z \\
\text{decode-}\Sigma\;(l_0 :: l')\;z = \begin{cases} (0, z) & \textbf{if } z < l_0 \\ (\mathrm{suc} \times \mathrm{id})(\text{decode-}\Sigma\;l'\;(z - l_0)) & \textbf{otherwise} \end{cases}
\end{array}
$$

Lemma 4.3.9 (Σ iso1)**.**

$$\forall (l : \mathrm{List}\,\mathbb{N})\;(x : \mathrm{Fin}\,(\mathrm{length}\,l)) \times \mathrm{Fin}\,l_x\;.\;\text{decode-}\Sigma\;l\;(\text{encode-}\Sigma\;l\;x) \equiv x$$

Proof. By induction on l, when $l = [\,]$, proof follows from the absurd assumption. So assume $l = l_0 :: l'$. The proof follows by the following case analysis on $\pi_0\,x$:

Case $\pi_0\,x = 0$**:**

$$
\begin{array}{l}
\text{decode-}\Sigma\;l\;(\text{encode-}\Sigma\;l\;x) \\
\qquad \equiv \begin{cases} (0, \pi_1\,x) & \text{if } \pi_1\,x < l_0 \\ (\mathrm{suc} \times \mathrm{id})(\text{decode-}\Sigma\;l'\;(\pi_1\,x - l_0)) & \text{otherwise} \end{cases} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition} \\
\qquad \equiv (0, \pi_1\,x) \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by } \pi_1\,x \textit{ of type } \mathrm{Fin}\,l_0 \\
\qquad \equiv (\pi_0\,x, \pi_1\,x) \equiv x \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{rewrite by case analysis}
\end{array}
$$

Case $\pi_0\ x = \mathrm{suc}\ x'$**:** Let $j = l_0 + (\text{encode-}\Sigma\ l'\ x'\ (\pi_1\ x))$.

$$
\begin{aligned}
&\text{decode-}\Sigma\ l\ (\text{encode-}\Sigma\ l\ x) \\
&\quad \equiv \begin{cases} (0, j) & \text{if } j < l_0 \\ (\mathrm{suc} \times \mathrm{id})(\text{decode-}\Sigma\ l'\ (j - l_0)) & \text{otherwise} \end{cases} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition} \\
&\quad \equiv (\mathrm{suc} \times \mathrm{id})(\text{decode-}\Sigma\ l'\ (j - l_0)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by } j \not< l_0 \\
&\quad \equiv (\mathrm{suc} \times \mathrm{id})(\text{decode-}\Sigma\ l'\ (\text{encode-}\Sigma\ l'\ x'\ (\pi_1\ x))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by } l_0 + a - l_0 = a \\
&\quad \equiv (\mathrm{suc} \times \mathrm{id})(x', \pi_1\ x) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by induction hypothesis} \\
&\quad \equiv (\pi_0\ x, \pi_1\ x) \equiv x \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by } \eta \textit{ contraction}
\end{aligned}
$$

□

Lemma 4.3.10 (Σ iso2)**.**

$$\forall\ (l : \mathrm{List}\ \mathbb{N})\ (x : \mathrm{Fin}\ (\Sigma\ l))\ .\ \text{encode-}\Sigma\ l\ (\text{decode-}\Sigma\ l\ x) \equiv x$$

Proof. By induction on l, when $l = []$, proof follows from the absurd assumption. So assume $l = l_0 :: l'$. The proof follows by case analysis on $x < l_0$.

Case $x < l_0$**:**

$$
\begin{aligned}
&\text{encode-}\Sigma\ l\ (\text{decode-}\Sigma\ l\ x) \\
&\quad \equiv \text{encode-}\Sigma\ l \left(\begin{cases} (0, x) & \text{if } x < l_0 \\ (\mathrm{suc} \times \mathrm{id})(\text{decode-}\Sigma\ l'\ (x - l_0)) & \text{otherwise} \end{cases} \right) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition} \\
&\quad \equiv \text{encode-}\Sigma\ l\ (0, x) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by case assumption} \\
&\quad \equiv x \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition}
\end{aligned}
$$

Case $x \not< l_0$:

$$\begin{aligned}
&\text{encode-}\Sigma\ l\ (\text{decode-}\Sigma\ l\ x) \\
&\qquad \equiv \text{encode-}\Sigma\ l \left(\begin{cases} (0, x) & \text{if } x < l_0 \\ (\text{suc} \times \text{id})(\text{decode-}\Sigma\ l'\ (x - l_0)) & \text{otherwise} \end{cases} \right) \\
&\qquad\qquad \textit{by definition} \\
&\qquad \equiv \text{encode-}\Sigma\ l\ (\text{suc} \times \text{id})(\text{decode-}\Sigma\ l'\ (x - l_0)) \\
&\qquad\qquad \textit{by case assumption} \\
&\qquad \equiv l_0 + (\text{encode-}\Sigma\ l'\ (\text{decode-}\Sigma\ l'\ (x - l_0))) \\
&\qquad\qquad \textit{by definition} \\
&\qquad \equiv l_0 + x - l_0 \equiv x \\
&\qquad\qquad \textit{by induction hypothesis and cancelling}
\end{aligned}$$

□

Determining whether the x^{th} variable in a state s has value y is accomplished by defining the following function:

$$\begin{aligned}
&\text{lookup} : (l : \text{List } \mathbb{N}) \to \text{Record } l \to ((x : \text{Fin } |l|) \times \text{Fin } l_x) \to \text{Bool} \\
&\text{lookup } [] \qquad r \qquad i \qquad\qquad = \text{efq } (\pi_0\ i) \\
&\text{lookup } (l :: ls)\ (e, r)\ (0, y) \qquad = e == y \\
&\text{lookup } (l :: ls)\ (e, r)\ (\text{suc } x, y) \ = \text{lookup } ls\ r\ (x, y)
\end{aligned}$$

4.3.4 Transition System

It is now shown how the symbolic transition system is defined, and how it is mapped into the underlying transition systems of 4.2. Symbolic transition systems are defined in module `CTL.RecordSystem` of Appendix F.

```
record SymFSM : Set where
  constructor
    fsm
  field
    state : List ℕ
    arrow : Record state → List ℕ
    initial : Record state
    transition : (s : Record state) → Record (arrow s) → Record state
```

The names used here are the same as for the definition of FSM's. Where the version cannot be determined by context the names/projections are proceeded by a super-script CTL or Sym. For example, to distinguish between the projection of the field state in an FSM or SymFSM, the projections are written as: $\text{state}^{\text{CTL}}_{\mathcal{M}}$ and $\text{state}^{\text{Sym}}_{\mathcal{M}}$.

The following constructs an FSM from an SymFSM. See Figure 4.2 for an example of the translation.

$$
\begin{aligned}
&\text{toFSM} : \text{SymFSM} \to \text{FSM}\\
&\text{toFSM}\ \mathcal{M}\ =\\
&\quad \text{fsm}\ (\Pi\ \text{state}_{\mathcal{M}})\\
&\qquad (\Sigma\ \text{state}_{\mathcal{M}})\\
&\qquad (\Pi \circ \text{arrow}_{\mathcal{M}} \circ \text{decode}_{\text{state}_{\mathcal{M}}})\\
&\qquad (\text{encode}_{\text{state}_{\mathcal{M}}}\ \text{initial}_{\mathcal{M}})\\
&\qquad (\lambda\ s\ a \to \text{encode}_{\text{state}_{\mathcal{M}}}\ (\text{transition}_{\mathcal{M}}\ (\text{decode}_{\text{state}_{\mathcal{M}}}\ s)\\
&\qquad\qquad\qquad\qquad (\text{decode}_{(\text{arrow}_{\mathcal{M}}\ (\text{decode}_{\text{state}_{\mathcal{M}}}\ s))}\ a)))\\
&\qquad (\lambda\ s\ p \to \text{lookup}\ \text{state}_{\mathcal{M}}\ (\text{decode}_{\text{state}_{\mathcal{M}}}\ s)\ (\text{decode-}\Sigma_{\text{state}_{\mathcal{M}}}\ p))
\end{aligned}
$$

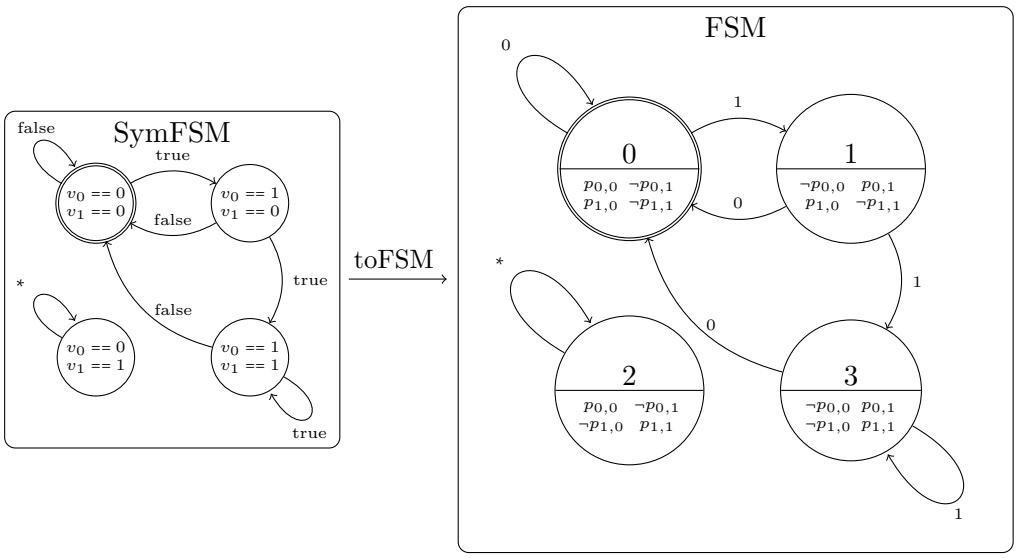


Figure 4.2: Example translation (toFSM) of a symbolic FSM (left) into an FSM (right). The symbolic FSM state consists of two Boolean variables, and the input consists of one Boolean variable. The translation clearly shows how the states and atomic propositions are related. Note that the true and false's are a representation of Fin 2 for readability.

Lemma 4.3.11 (transition-1)**.** *Assume* $\mathcal{M} : \mathrm{SymFSM}$*,* $s : \mathrm{Record}\ \mathrm{state}_{\mathcal{M}}$*, and* $a : \mathrm{Record}\ (\mathrm{arrow}_{\mathcal{M}}\ s)$*, then the following holds.*

$$\begin{aligned}&\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{transition}^{\mathrm{Sym}}_{\mathcal{M}}\ s\ a)\\ &\qquad \equiv \mathrm{transition}^{\mathrm{CTL}}_{(\mathrm{toFSM}\ \mathcal{M})}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)\ (\mathrm{encode}_{(\mathrm{arrow}_{\mathcal{M}}\ s)}\ a)\end{aligned}$$

Proof. The proof follows by equality reasoning.

$$\begin{aligned}&\mathrm{transition}^{\mathrm{CTL}}_{(\mathrm{toFSM}\ \mathcal{M})}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)\ (\mathrm{encode}_{(\mathrm{arrow}_{\mathcal{M}}\ s)}\ a)\\ &\quad \equiv \mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{transition}^{\mathrm{Sym}}_{\mathcal{M}}\ (\mathrm{decode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s))\\ &\qquad\qquad (\mathrm{decode}_{(\mathrm{arrow}_{\mathcal{M}}\ (\mathrm{decode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)))}\ (\mathrm{encode}_{(\mathrm{arrow}_{\mathcal{M}}\ s)}\ a)))\\ &\qquad\qquad\qquad \textit{by definition}\\ &\quad \equiv \mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{transition}^{\mathrm{Sym}}_{\mathcal{M}}\ s\ a)\\ &\qquad\qquad\qquad \textit{by Lemma 4.3.6}\end{aligned}$$

□

Lemma 4.3.12 (transition-2)**.** *Assume* $\mathcal{M} : \mathrm{SymFSM}$*,* $s : \mathrm{Record}\ \mathrm{state}_{\mathcal{M}}$*, and* $a : \mathrm{Fin}\ (\Pi\ (\mathrm{arrow}_{\mathcal{M}}\ s))$*, then the following holds.*

$$\begin{aligned}&\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{transition}^{\mathrm{Sym}}_{\mathcal{M}}\ s\ (\mathrm{decode}_{\mathrm{arrow}_{\mathcal{M}}}\ a))\\ &\qquad \equiv \mathrm{transition}^{\mathrm{CTL}}_{(\mathrm{toFSM}\ \mathcal{M})}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)\ a\end{aligned}$$

Proof. The proof follows by equality reasoning.

$$\begin{aligned}&\mathrm{transition}^{\mathrm{CTL}}_{(\mathrm{toFSM}\ \mathcal{M})}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)\ a\\ &\quad \equiv \mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{transition}^{\mathrm{Sym}}_{\mathcal{M}}\ (\mathrm{decode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s))\\ &\qquad\qquad (\mathrm{decode}_{(\mathrm{arrow}_{\mathcal{M}}\ (\mathrm{decode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)))}\ a))\\ &\qquad\qquad\qquad \textit{by definition}\\ &\quad \equiv \mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{transition}^{\mathrm{Sym}}_{\mathcal{M}}\ s\ (\mathrm{decode}_{\mathrm{arrow}_{\mathcal{M}}}\ a))\\ &\qquad\qquad\qquad \textit{by Lemma 4.3.6}\end{aligned}$$

□

4.3.5 Runs

It is possible to define the runs over a symbolic transition system $\mathcal{M}$ as the runs resulting from the translation in Section 4.3.4, i.e. $\mathrm{Run}_{(\mathrm{toFSM}\,\mathcal{M})}$. However, this would result in obfuscation of the semantics of symbolic model-checking (cf. Section 4.3.7), meaning that it would be harder to validate that the semantics is correct. Instead, the runs are redefined using the language of SymFSM and then used to define the semantics of symbolic model-checking. The semantics will be shown to be equivalent to CTL model-checking in Section 4.3.7.

A run over $\mathcal{M}$ is defined similarly to a run over an FSM:

$$\begin{aligned}&\mathrm{data}\ \mathrm{SymRun}_{\mathcal{M}}\ (s : \mathrm{Record}\ \mathrm{state}_{\mathcal{M}}) : \mathrm{Set}\ \mathrm{where}\\ &\quad \mathrm{next}\ :\ (a : \mathrm{Record}\ (\mathrm{arrow}_{\mathcal{M}}\ s))\\ &\qquad\qquad \to \infty\ \mathrm{SymRun}_{\mathcal{M}}\ (\mathrm{transition}_{\mathcal{M}}\ s\ a)\\ &\qquad\qquad \to \mathrm{SymRun}_{\mathcal{M}}\ s\end{aligned}$$

where ∞ prefixes a term that can potentially be unfolded infinitely many times.

A run over a symbolic transition system $\mathcal{M}$ from state s is translated into a run over a transition system by:

$$\begin{aligned}&\mathrm{toRun}_{\mathcal{M}} : \forall s\ .\ \mathrm{SymRun}_{\mathcal{M}}\ s \to \mathrm{Run}_{(\mathrm{toFSM}\,\mathcal{M})}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)\\ &\mathrm{toRun}_{\mathcal{M}}\ (\mathrm{next}^{\mathrm{Sym}}\ a\ r)\ =\ \mathrm{next}^{\mathrm{CTL}}\ (\mathrm{encode}_{(\mathrm{arrow}_{\mathcal{M}}\ s)}\ a)\\ &\qquad\qquad\qquad\qquad\qquad\qquad (\sharp\ \mathrm{toRun}_{\mathcal{M}}\ (\mathrm{transition}^{\mathrm{Sym}}_{\mathcal{M}}\ s\ a)\ (\flat\ r))\end{aligned}$$

The induction hypothesis used here requires a significant amount of equality reasoning (not presented for clarity), which follows by Lemma 4.3.6 and Lemma 4.3.11.

The inverse will also be required, and is defined as follows:

$$\begin{aligned}&\mathrm{fromRun}_{\mathcal{M}} : \forall s\ .\ \mathrm{Run}_{(\mathrm{toFSM}\,\mathcal{M})}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s) \to \mathrm{SymRun}_{\mathcal{M}}\ s\\ &\mathrm{fromRun}_{\mathcal{M}}\ (\mathrm{next}^{\mathrm{CTL}}\ a\ r)\ =\\ &\quad \mathrm{next}^{\mathrm{Sym}}\ (\mathrm{decode}_{(\mathrm{arrow}_{\mathcal{M}}\ s)}\ a)\\ &\qquad\qquad (\sharp\ \mathrm{fromRun}_{\mathcal{M}}\ (\mathrm{transition}^{\mathrm{Sym}}_{\mathcal{M}}\ s\ (\mathrm{decode}_{(\mathrm{arrow}_{\mathcal{M}}\ s)}\ a))\ (\flat\ r))\end{aligned}$$

Similarly, the induction hypothesis used here requires equality reasoning that follows by Lemma 4.3.6 and Lemma 4.3.12.

It is not required to prove that a composition of these translations yields bisimilar runs. Although this does follow from how a symbolic transition system is translated into an FSM, and by Lemma 4.3.11 and Lemma 4.3.12.

4.3.6 Formulæ

Symbolic CTL formulæ (SymCTL) are the same as CTL formulæ (in Section 4.2) with the exception that the operator P a (proposition a holds in the current state) is adjusted accordingly to reason about equality tests on state variables. That is P$[x == y]$ stands for variable x has value y in the current state.

For a given SymFSM transition system $\mathcal{M}$ system defined over the list $l = [|A_0|, |A_1|, \ldots, |A_{i-1}|]$, the set of formulæ are defined as follows:

$$
\begin{array}{l}
\text{data SymCTL } (\mathcal{M} : \text{FSM}) : \text{Set where} \\
\quad \begin{array}{ll}
\text{false} & : \text{SymCTL}_{\mathcal{M}} \\
\neg & : \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}} \\
\_\vee\_ & : \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}} \\
\_\wedge\_ & : \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}} \\
\mathbf{P[\_==\_]} & \mathbf{: (x : Fin}\ i\mathbf{) \rightarrow Fin}\ |A_x| \mathbf{\rightarrow SymCTL_{\mathcal{M}}} \\
\text{EX} & : \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}} \\
\text{EG} & : \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}} \\
\text{E[\_U\_]} & : \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}} \rightarrow \text{SymCTL}_{\mathcal{M}}
\end{array}
\end{array}
$$

Mapping $\varphi : \text{SymCTL}_{\mathcal{M}}$ into $\text{CTL}_{(\text{toFSM}\,\mathcal{M})}$ is canonical with respect to the operators. In the case of an atomic proposition $x == y$, it must be translated into the proposition which holds *iff* variable x has the value y. Let toCTL be the name of this map, given by:

$$
\begin{array}{ll}
\text{toCTL}_{\mathcal{M}} : \text{SymCTL}_{\mathcal{M}} \rightarrow \text{CTL}_{(\text{toFSM}\,\mathcal{M})} & \\
\text{toCTL}_{\mathcal{M}}\ \text{false} & = \text{false} \\
\text{toCTL}_{\mathcal{M}}\ (\neg\, \varphi) & = \neg\, (\text{toCTL}\ \varphi) \\
\text{toCTL}_{\mathcal{M}}\ (\varphi \vee \psi) & = (\text{toCTL}_{\mathcal{M}}\ \varphi) \vee (\text{toCTL}_{\mathcal{M}}\ \psi) \\
\text{toCTL}_{\mathcal{M}}\ (\varphi \wedge \psi) & = (\text{toCTL}_{\mathcal{M}}\ \varphi) \wedge (\text{toCTL}_{\mathcal{M}}\ \psi) \\
\mathbf{toCTL_{\mathcal{M}}\ (P[\ x == y\ ])} & \mathbf{= P\ (encode\text{-}\Sigma\ l\ (x,y))} \\
\text{toCTL}_{\mathcal{M}}\ (\text{EX}\ \varphi) & = \text{EX}\ (\text{toCTL}_{\mathcal{M}}\ \varphi) \\
\text{toCTL}_{\mathcal{M}}\ (\text{EG}\ \varphi) & = \text{EG}\ (\text{toCTL}_{\mathcal{M}}\ \varphi) \\
\text{toCTL}_{\mathcal{M}}\ (\text{E[}\ \varphi\ \text{U}\ \psi\ \text{]}) & = \text{E[}\ (\text{toCTL}_{\mathcal{M}}\ \varphi)\ \text{U}\ (\text{toCTL}_{\mathcal{M}}\ \psi)\ \text{]}
\end{array}
$$

4.3.7 Correctness

The semantics of symbolic CTL model-checking is similar to the semantics of CTL model-checking; however, there are two differences. First, the formula

P[x == y] is assigned the semantics that variable x has the value y in the current state; this is opposed to a given proposition holding in the current state. The second is that the runs are built over a symbolic transition system.

The semantics can be given as follows:

$$
\begin{array}{l}
[\![\_,\_\vDash\_]\!] : (\mathcal{M} : \mathrm{SymFSM}) \to (\mathrm{Record\ state}_{\mathcal{M}}) \to \mathrm{SymCTL}_{\mathcal{M}} \to \mathrm{Set} \\
[\![\ \mathcal{M}\ , s \vDash \quad \mathrm{false} \quad]\!] = \bot \\
[\![\ \mathcal{M}\ , s \vDash \quad \neg\varphi \quad]\!] = [\![\ \mathcal{M}\ , s \vDash \varphi\ ]\!] \to \bot \\
[\![\ \mathcal{M}\ , s \vDash \quad \varphi \vee \psi \quad]\!] = [\![\ \mathcal{M}\ , s \vDash \varphi\ ]\!] + [\![\ \mathcal{M}\ , s \vDash \psi\ ]\!] \\
\boldsymbol{[\![\ \mathcal{M}\ , s \vDash \mathrm{P}[\ x == y\ ]\ ]\!] = \mathrm{T}\ (\mathrm{lookup\ state}_{\mathcal{M}}\ s\ (x,y))} \\
[\![\ \mathcal{M}\ , s \vDash \quad \mathrm{EX}\ \varphi \quad]\!] = \exists\ (run : \mathrm{SymRun}_{\mathcal{M}}\ s)\ [\![\ \mathcal{M}\ , run_1 \vDash \varphi\ ]\!] \\
[\![\ \mathcal{M}\ , s \vDash \quad \mathrm{EG}\ \varphi \quad]\!] = \exists\ (run : \mathrm{SymRun}_{\mathcal{M}}\ s)\ (\forall i\ .\ [\![\ \mathcal{M}\ , run_i \vDash \varphi\ ]\!]) \\
[\![\ \mathcal{M}\ , s \vDash \ \mathrm{E}[\ \varphi\ \mathrm{U}\ \psi\ ]\ ]\!] = \exists\ (run : \mathrm{SymRun}_{\mathcal{M}}\ s)\ \exists\ (k : \mathbb{N})\ (\\
\qquad\qquad\qquad\qquad\qquad\qquad (\forall j\ .\ j < k \to [\![\ \mathcal{M}\ , run_j \vDash \varphi\ ]\!]) \\
\qquad\qquad\qquad\qquad\qquad\qquad \times [\![\ \mathcal{M}\ , run_k \vDash \psi\ ]\!])
\end{array}
$$

Below, Theorem 4.3.13 states that a given symbolic transition system $\mathcal{M}$, and a state s in $\mathcal{M}$ models a formula φ built over $\mathcal{M}$, *iff* their translations into CTL holds. Following by this theorem: it is sound to lift the ctlcheck decision procedure for plain CTL to symbolic CTL. That is, the following function is defined:

$$
\begin{array}{l}
\mathrm{symctlcheck} : (\mathcal{M} : \mathrm{SymFSM}) \to (\mathrm{Record\ state}_{\mathcal{M}}) \to \mathrm{SymCTL}_{\mathcal{M}} \to \mathrm{Bool} \\
\mathrm{symctlcheck}\ \mathcal{M}\ s\ \varphi\ =\ \mathrm{ctlcheck}\ (\mathrm{toFSM}\ \mathcal{M})\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)\ (\mathrm{toCTL}_{\mathcal{M}}\ \varphi)
\end{array}
$$

See module `CTL.RecordSystem` in Appendix F for more information. In the next chapter, this function will be overridden such that an efficient external tool will be executed in it's place. This is good because the input to the external model-checker is defined over a symbolic model, hence, better performance is obtained as more structure of the original problem is preserved. Subsequently in Section 9.4.1, the transition relation will be replaced by a Boolean valued program, this will provide optimal performance when using the external tools as all the structure is preserved.

Theorem 4.3.13 (correctness)**.**

$$
\begin{array}{l}
\forall \mathcal{M}\ s\ \varphi\ . \\
\qquad [\![\ \mathcal{M}\ ,\ s \vDash \varphi\ ]\!]^{\mathrm{Sym}} \leftrightarrow [\![\ \mathrm{toFSM}\ \mathcal{M}\ ,\ \mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s\ \vDash\ \mathrm{toCTL}_{\mathcal{M}}\ \varphi\ ]\!]^{\mathrm{CTL}}
\end{array}
$$

Proof. By induction φ. The logical connectives and constants are trivial. Regarding the CTL connectives, only the proof of the cases where $\varphi = \mathrm{P}[\ x == y\ ]$ and $\varphi = \mathrm{EX}\ \varphi'$ are carried out below. The remaining cases are left to the reader.

Case $\varphi = \mathrm{P}[\ x == y\ ]$**:**

$$\begin{aligned}
&[\![\ \mathrm{toFSM}\ \mathcal{M}\ ,\ \mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s \vDash \mathrm{toCTL}_{\mathcal{M}}\ \varphi\ ]\!]^{\mathrm{CTL}} \\
&\quad \equiv \mathrm{T}\ (\mathrm{lookup}\ \mathrm{state}_{\mathcal{M}}\ (\mathrm{decodestate}_{\mathcal{M}}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s)) \\
&\qquad\qquad (\text{decode-}\Sigma\ \mathrm{state}_{\mathcal{M}}\ (\text{encode-}\Sigma\ \mathrm{state}_{\mathcal{M}}\ (x, y)))) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition} \\
&\quad \equiv \mathrm{T}\ (\mathrm{lookup}\ \mathrm{state}_{\mathcal{M}}\ (\mathrm{decode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s))\ (x, y)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{rewrite by Lemma 4.3.9} \\
&\quad \equiv \mathrm{T}\ (\mathrm{lookup}\ \mathrm{state}_{\mathcal{M}}\ s\ (x, y)) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{rewrite by Lemma 4.3.6} \\
&\quad \equiv [\![\ \mathcal{M}\ ,\ s \vDash \varphi\ ]\!]^{\mathrm{Sym}} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition}
\end{aligned}$$

Case $\varphi = \mathrm{EX}\ \varphi'$**:** The goal is

$$\begin{aligned}
&\exists(\mathit{run} : \mathrm{SymRun}_{\mathcal{M}}\ s)\ .\ [\![\ \mathcal{M}\ , \mathit{run}_1 \vDash \varphi'\ ]\!]^{\mathrm{Sym}} \\
&\qquad\qquad \leftrightarrow \exists(\mathit{run} : \mathrm{Run}_{(\mathrm{toFSM}\ \mathcal{M})}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s))\ . \\
&\qquad\qquad\qquad\qquad [\![\ \mathrm{toFSM}\ \mathcal{M}\ ,\ \mathit{run}_1 \vDash \mathrm{toCTL}_{\mathcal{M}}\ \varphi'\ ]\!]^{\mathrm{CTL}} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition}
\end{aligned}$$

Each direction is treated separately.

"⇒" Assume $r : \mathrm{SymRun}_{\mathcal{M}}\ s$, $p : [\![\ \mathcal{M}\ , r_1 \vDash \varphi'\ ]\!]^{\mathrm{Sym}}$, and let $r' = \mathrm{toRun}_{\mathcal{M}}\ s\ r$. It is now shown that r' fulfils

$$[\![\ \mathrm{toFSM}\ \mathcal{M}\ ,\ r'_1 \vDash \mathrm{toCTL}_{\mathcal{M}}\ \varphi'\ ]\!]^{\mathrm{CTL}}$$

$$\begin{aligned}
r &\equiv \mathrm{next}^{\mathrm{Sym}}\ a\ r'' \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition} \\
r'_1 &\equiv \mathrm{transition}_{(\mathrm{toFSM}\ \mathcal{M})}\ (\mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ s) \\
&\qquad\qquad\qquad\qquad\qquad (\mathrm{encode}_{(\mathrm{arrow}_{\mathcal{M}}\ s)}\ a) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition} \\
&\equiv \mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ (\mathrm{transition}_{\mathcal{M}}\ s\ a) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by Lemma 4.3.11} \\
&\equiv \mathrm{encode}_{\mathrm{state}_{\mathcal{M}}}\ r_1 \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (*)
\end{aligned}$$

by definition

Therefore

$$\begin{aligned}&[\![\ \text{toFSM}\ \mathcal{M}\ ,\ r_1' \vDash \text{toCTL}_{\mathcal{M}}\ \varphi'\ ]\!]^{\text{CTL}}\\ &\quad\equiv [\![\ \text{toFSM}\ \mathcal{M}\ ,\ \text{encode}_{\text{state}_{\mathcal{M}}}\ r_1 \vDash \text{toCTL}_{\mathcal{M}}\ \varphi'\ ]\!]^{\text{CTL}}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{rewrite by } (*)\end{aligned}$$

This follows by induction hypothesis applied to p.

"$\Leftarrow$" Assume $r : \text{Run}_{(\text{toFSM}\ \mathcal{M})}\ s$,
$p : [\![\ \text{toFSM}\ \mathcal{M}\ , r_1 \vDash \text{toCTL}_{\mathcal{M}}\ \varphi'\ ]\!]^{\text{CTL}}$,
and let $r' = \text{fromRun}_{\mathcal{M}}\ s\ r$. It is now shown that r' fulfils

$$[\![\ \mathcal{M}\ , r_1' \vDash \varphi'\ ]\!]^{\text{Sym}}$$

$$\begin{aligned}r &\equiv \text{next}^{\text{CTL}}\ a\ r''\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition}\\ r_1 &\equiv \text{transition}^{\text{CTL}}_{(\text{toFSM}\ \mathcal{M})}\ (\text{encode}_{\text{state}_{\mathcal{M}}}\ s)\ a\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition}\\ &\equiv \text{encode}_{\text{state}_{\mathcal{M}}}\ (\text{transition}^{\text{Sym}}_{\mathcal{M}}\ s\ (\text{decode}_{(\text{arrow}_{\mathcal{M}}\ s)}\ a))\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by Lemma 4.3.12}\\ &\equiv \text{encode}_{\text{state}_{\mathcal{M}}}\ r_1' \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad (*)\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition}\end{aligned}$$

Therefore

$$\begin{aligned}&p : [\![\ \text{toFSM}\ \mathcal{M}\ , \text{encode}_{\text{state}_{\mathcal{M}}}\ r_1' \vDash \text{toCTL}_{\mathcal{M}}\ \varphi'\ ]\!]^{\text{CTL}}\\ &\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by } (*)\end{aligned}$$

The proof follows by the induction hypothesis applied to p.

□

Chapter 5

Modification made to Agda

A substantial portion of this thesis is concerned with modifying Agda to facilitate soundly (and securely) calling external tools during type-checking. Previously in Chapter 3 the underlying theory of the integration was presented, then in Chapter 4 the examples of SAT and CTL were explained. In this chapter, it is shown how to take advantage of the existing built-in mechanism in Agda to allow external tools to be executed—according to the integration in Chapter 3.

In the next chapter, an alternative integration is explored. This is when the external tool provides justifications that are then checked by Agda.

Chapter Overview. In Section 5.1, the built-in mechanism is introduced. Briefly it is a system to allow the type-checker to execute Haskell code during type-checking. Also in that section it is explained how to extend the built-in mechanism to allow external tools to be executed. In Section 5.3.1 and Section 5.3.2, the built-in mechanism is modified to achieve the SAT and CTL interfaces, respectively. The techniques learnt from these sections are generalised in Section 5.4, where a generic interface is presented. In Section 5.5 it is considered how to prevent malicious programs being specified. A technical how-to relating to these modifications can be found in Appendix D, along with selected code listings of the modifications that were required.

Orthogonally, other experimental modifications have been explored such as the extended lambda expressions in Section 5.6.2, and profiling of the reduction machinery in Section 5.6.1.

5.1 Built-In Mechanism

A common problem with implementations of type-theory + inductive constructions, is the usability of natural numbers [Bra05, AGST10]. Natural

numbers are mathematically defined inductively as the least set containing 0 and closed under successor. The elements consists of the terms of the form:

$$\mathrm{suc}\,(\mathrm{suc}\,(\ldots(\mathrm{suc}\,0)\ldots))$$

From a usability perspective it is desirable to use a base 10 representation of a natural number, and for closed terms efficient (machine) evaluation of mathematical operations such as _+_ and _*_.

To achieve this Agda uses a system of built-in terms; these are terms defined in Agda's type-checker. When one of these terms is encountered during type-checking (identified by a "BUILTIN" pragma) they are checked to fulfil a number of hard-coded constraints. For an inductive construction, its type, the number of and types of its constructors are checked; and for a function definition, its type and definition is checked against a set of axioms. Thus, it is possible[1] for the type-checker to know up to isomorphism what the built-in terms are and how they behave.

Moreover in the case of built-in inductive constructions, bijective maps toTerm and fromTerm are given which translate between an Agda term and a Haskell term. For non-dependently typed constructions such as natural numbers and lists, this is trivial as Haskell has corresponding constructions. This is also possible for dependently typed constructions, but the maps require significant attention to ensure that they are consistent. In the case of built-in functions, assuming that their types are constructed from built-in types (e.g. natural numbers, lists) it is possible to override their Agda implementation when applied to closed terms by an efficient Haskell implementation.

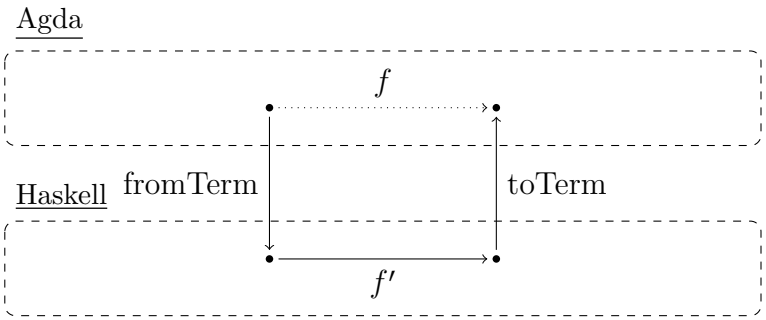


Figure 5.1: The implementation of a built-in function f can be replaced by the Haskell function f', such that the above diagram commutes.

[1] Assuming the constraints are correct.

To exemplify this, consider the addition of natural numbers. First, the set of naturals is defined as follows:

$$\begin{aligned}&\text{data } \mathbb{N} : \text{Set where}\\ &\quad \text{zero} \;:\; \mathbb{N}\\ &\quad \text{suc} \;:\; \mathbb{N} \to \mathbb{N}\end{aligned}$$

The type-checker is informed that this is the set of natural numbers by the following three pragmas.

1. {−# BUILTIN NAT ℕ #−}
   Checks that ℕ is an inductive construction of type Set that has two constructors, the types of the constructors are not checked yet.

2. {−# BUILTIN NATZERO zero #−}
   Checks that zero is a constructor of ℕ and has the type ℕ.

3. {−# BUILTIN NATSUC suc #−}
   Checks that suc is a constructor of ℕ and has the type ℕ → ℕ.

Remark

Anton Setzer and myself discovered an inconsistency in Agda's built-in mechanism for natural numbers. The inconsistency was rapidly fixed by the developers. A description is as follows: The problem was that the parser would happily read base-10 representations of natural numbers before all 3 of the built-in pragmas for natural numbers had been processed. This was exploited by defining an empty set of natural numbers with two mal-typed constructors, then only processing the first pragma, hence, the constructors do not have their types checked. When the parser encountered a base-10 number, it internally assumed that it was a natural number, and thus an element of the non-wellformed natural numbers was obtained. This was further exploited to obtain a proof of absurdity. This inconsistency was fixed by requiring the parser ensure that all three natural number pragmas had been processed successfully before parsing a base-10 representation.

The definition of addition is given as follows:

$$\begin{aligned}&\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N}\\ &\text{zero} \;+ m = m\\ &\text{suc}\; n + m = \text{suc}\; (n + m)\end{aligned}$$

It has a run-time complexity linear to the number of constructors in its first argument. However, computers have dedicated hardware that add two numbers (less than some bound, e.g. 2^{32}) efficiently. To this end the following pragma

$$\{-\#\ \text{BUILTIN NATPLUS}\ \_+\_\ \#-\}$$

notifies the type-checker that $x + y$ is to be evaluated for closed terms x and y by using Haskell's integer addition operator. More accurately it is when fromTerm does not fail on x and y that the Haskell implementation is used. In cases where fromTerm fails, the definition is β-reduced using the above definition. E.g. $2 + x$ reduces to suc (suc x) in three steps.

Agda does not make provide any assurances that the Haskell function behaves like the slow Agda function. For this reason adding new built-in functions should be done with caution, and requires a recompilation of the Agda source code. This recompilation ensures that novices do not accidentally create inconsistencies.

It should be made clear that the built-in system does not facilitate entering Haskell terms into Agda files, it only notifies the type-checker that an Agda term has been identified with a tag. As an example, above the tag is "NATPLUS". The type-checker then chooses what action to take, e.g. check the term has a given type, or bind it to a primitive implementation. In the case of natural numbers (and similarly for other literals), the Agda parser has been tweaked to allow base-10 representations, these are translated on-demand into the correct inductive construction identified by the tags: NATZERO, NATSUC.

Agda also has the notion of a primitive function; these are similar to built-in functions but do not β-reduce for open terms. A primitive function is not implemented in Agda (only its signature), and thus it behaves as a black box and is not checked.

For more information about Agda in general, please see Ulf Norell's thesis [Nor07] and the Agda wiki.

5.1.1 Theory of Built-Ins

In this section, a complete understanding of Agda is assumed, built-in and primitive functions are defined by their *observed* behaviour, then proofs about their sound and consistent use are presented.

Caveat:

In the following, proofs are carried out about Agda without a full formalisation of Agda, by instead referring to what is derivable in Agda, in the consistent fragment of Agda. Therefore, these proofs are not fully formal,

and therefore, strictly speaking the theorems are not fully mathematical theorems. A full formalisation would require a formal formalisation of Agda, and its consistent fragment. This goes beyond the scope of this thesis which is focused on practical verification of railway interlocking systems. Making it fully formal would as well obfuscate the message in technicalities. All definitions, theorems and lemmata on the following issue are subject to this caveat.

By deriving something in Agda from some code, it is meant that only Agda constructs are used, which by themselves only construct consistent Agda content. That is it is not possible to derive an element of the empty type $\bot$.

Assumptions:
In this section, we assume a complete formalisation of Agda based on Martin-Löf's logical framework, without built-in or primitive functions. Assume that the reduction rules are given by equations in Agda, and the standard reductions of the logical framework. Define the reduction relation $r \longrightarrow s$ for terms r and s, s.t. if $r \longrightarrow s$ and $r : A$, $s : A$, then $r \equiv s : A$.

We assume that the termination checker of Agda passes non-recursive definitions by pattern matching; also assume the definitions of the equality type, and $\bot$.

We assume that Agda is consistent with any reduction strategy. Without loss of generality, the reduction relation is just the call-by-value reduction, i.e. it reduces the innermost term to which a reduction applies first.

In the following, it is explained what it is meant to add a built-in, and a primitive function to some Agda code.

Definition (built-in function)**.** Assume the following Agda code:

$$\begin{aligned} & f : A \to B \\ & f\ a_1 \ = \ b_1 \\ & \ \ \vdots \quad = \ \ \vdots \\ & f\ a_n \ = \ b_n \\ & \{-\#\ \text{BUILTIN C}\ f\ \#-\} \end{aligned}$$

Here f is defined by total pattern matching on the non-overlapping cases a_i of A, and b_j are arbitrary terms of type B. Assume the (non-Agda) function

$$f^{\mathrm{C}}_{\mathrm{external}} : A \to B$$

mapping closed terms to closed terms, which corresponds to the built-in C. The effect of this code is to add to the model a new constant f, and for each $i \in \{1, \ldots, n\}$ the reduction rule

$$f\ a_i \longrightarrow b_i$$

provided a_i is an open term, and for closed terms $a : A$

$$b = f^{\mathrm{C}}_{\mathrm{external}}\ a$$

a reduction rule

$$f\ a \longrightarrow b$$

that takes precedence over the previous reductions.

Definition (primitive function)**.** Assume the following Agda code:

$$\begin{array}{c}\mathrm{primitive}\\ f : A \to B\end{array}$$

and the non-Agda function

$$f_{\mathrm{external}} : A \to B$$

mapping closed terms to closed terms, which provides the implementation of f. The effect of this code is to add to the model a new constant f, and for all closed terms $a : A$

$$b = f_{\mathrm{external}}\ a$$

a reduction rule

$$f\ a \longrightarrow b$$

N.B. Primitive functions do not reduce on open terms.

The following theorem describes a subtle case relating to overriding functions using the built-in mechanism that might lead to inconsistencies. Note that in this thesis it is assumed that in any Agda the built-in pragma immediately follows the function to be built-in.

Theorem 5.1.1 (Built-In Function Inconsistent)**.** *Assume consistent Agda code, and extend it with a built-in function $f : A \to B$ that is overridden by an implementation $f_{\mathrm{external}} : A \to B$ such that both of the following hold:*

- *If there is a closed term $a : A$ and a defining equation of f of the form $f\ a = b$, then $b = f_{\mathrm{external}}\ a$;*
- *There exists an element $a : A$, such that $f\ a$ reduces to a different result from $f_{\mathrm{external}}\ a$.*

Then the extended Agda code is inconsistent.

Remark

The first condition of the above theorem is necessary, as seen by the following: If a function $f : A \to B$ has a defining equation of the form $f\ a = b$ for some closed $a : A$, and it is overridden by f_{external}, such that $f_{\text{external}}\ a \longrightarrow b'$. Then $f\ a$ will always evaluate to b' and we do not have access to the defining equation $f\ a = b$ any more. So after overriding, f behaves as if it had the defining equation $f\ a = b'$.

As an example, consider the following consistent Agda code:

$$
\begin{aligned}
&f : \mathbb{N} \to \mathbb{N} \\
&f\ 0 \quad\ = 1 \\
&f\ (\text{suc}\ n) = \text{suc}\ (f\ n) \\
&\{-\#\ \text{BUILTIN IDENTITY}\ f\ \#-\}
\end{aligned}
$$

$$
\begin{aligned}
&f' : \mathbb{N} \to \mathbb{N} \\
&f'\ 0 \quad\ = 1 \\
&f'\ (\text{suc}\ n) = \text{suc}\ (f\ n)
\end{aligned}
$$

Here, the functions f and f' are defined by the same case distinctions, with f' intentionally referring to f on the right-hand side. The function f is overridden by the identity function, which violates the first condition.

For closed terms f is replaced by the identity function, therefore, $f\ 0$ reduces to 0. It can not be derived that $f\ 0$ was defined to reduce to 1, and hence, it can not be proved (provided the built-in pragma directly follows f) that $\forall n\ .\ f\ n \equiv f'\ n$. See the following proof for more information.

Proof. Let f' be a function of type $A \to B$, which is defined in Agda by using the same case-distinction as f, however at the right-hand side by referring to f rather than recursively calling f'. By using the same case-distinction as f for f', it follows immediately that we can prove in Agda $\forall x : A\ .\ f\ x \equiv f'\ x$ (since for an open term a, the definition of f is not overridden, and for a defining closed term a, we have that $f\ a$ and $f'\ a$ coincide).

Let $a : A$ be a closed term, such that the original value of $f\ a$ differs from $f_{\text{external}}\ a$. In the recursive evaluation of $f\ a$ there must be a term a' (which can be a itself), such that $f\ a'$ is evaluated incorrectly by f_{external}, but all recursive calls used in evaluating $f\ a'$ are evaluated correctly. Therefore $f'\ a'$ returns the result of the non-overridden function f, whereas evaluating $f\ a'$ returns a different result (because it is being overridden). We obtain a proof

that $f\ a' \not\equiv f'\ a'$, contradicting $\forall x : A\ .\ f\ x \equiv f'\ x$. □

Theorem 5.1.2 (Primitive Function Sound)**.** *Assume a primitive function $f : (a : A) \to B$, such that its implementation f_{external} is definable (in principle) in Agda by a finite case-distinction, terminates for all closed terms, and would pass the termination checker. Then if Agda was sound without f, it is sound with f.*

Remark

Should the result type of a primitive function f be empty for some arguments, then Agda becomes inconsistent. Similarly should f_{external} not terminate for some arguments, then Agda becomes inconsistent.

Proof. Assume some Agda code, extend it by defining the function f', such that f' is defined using the same case-distinctions and equations as f_{external} would be, but in Agda. Then for all closed terms f' and f_{external} reduce to the same normal-form. However for open terms f' will attempt to reduce, and f_{external} will not. Therefore the revised code proves more theorems than the original code with f. Since the revised code is sound, the original code is sound as well. □

5.2 Extending The Built-In Mechanism

It became apparent that the built-in mechanism could be used to provide an efficient implementation for decision procedures, especially for non-trivial problems. A built-in decision procedure can call an external tool and return the result to Agda in one reduction step. This involves providing the fromTerm function for the problem set and toTerm for the resulting value, a number of equations that the decision procedure must fulfil, and a Haskell function that replaces the decision procedure.

This is a significant amount of work as not only the three items listed above are required, but also that all types referenced must be declared as built-ins (requiring associated toTerm and fromTerm functions), and all functions referenced must also be declared as built-ins (requiring axioms and Haskell implementations).

In Section 5.3.1 this technique is applied to connect Agda with a SAT solver. However, it soon became apparent that, for complex decision procedures such as CTL model-checking, this technique would become unmanageable. This gave rise to the notion of pseudo built-ins.

5.2.1 Pseudo Built-Ins

The motivation behind the introduction of pseudo built-ins is clearly seen when overriding an Agda function (decision procedure) that refers to other Agda functions. Without pseudo built-ins this requires to define these functions and the functions they depend on as built-ins, because their axioms and types can only refer to other built-ins.

Defining non-trivial Agda decision procedures as built-ins that depend upon Agda functions is cumbersome for the following two reasons. The first is the amount of effort/development time involved to define a decision procedure as a built-in. The second is that dependent/large types have no canonical translation into Haskell, and require hand crafted solutions.

The first issue is that the same decision procedure (built-in function) and associated definitions are written three times, once in Agda, once Agda's source code as an axiom check (inc. type signature), and once in Haskell. For example, consider the simple Agda function:

$$\begin{aligned} &\text{test} : \mathbb{N} \to \mathbb{N} \to \text{Bool} \\ &\text{test}\ n\ m\ =\ n * n < m \wedge n \neq 0 \end{aligned}$$

To build it in, first, a Haskell implementation - which should behave equivalently - must be provided. To mitigate any issues that the Agda and Haskell implementations do not coincide, an axiom check is also specified. This axiom check amounts to implementing the function a third time. That is, it checks that the function test has the correct type, which in this case would require that $\mathbb{N}$ and Bool are built-in types. It also checks that each defining equation is defined correctly, which in this case checks that $\text{test}\ n\ m$ is definitionally equal to $n * n < m \wedge n \neq 0$. To formalise the axiom check it is required that the terms used in the definition are also built-in, which in this case includes $\_*\_$, $\_<\_$, $\_\neq\_$ and 0.

In the above example, most of the auxiliary types/functions are already built-in, however, for complicated decision procedures this is not the case. Importantly, if the function is to be replaced by an external tool, then the Haskell implementations of the auxiliary functions are never executed, resulting in a wasted effort.

The second issue relates to the difficulties of building in dependently typed functions. As an example, consider the atom function $\top : \text{Bool} \to \text{Set}$. It is trivial to define it in Agda, and with a little bit of work, it can be defined in the Agda source. However, defining it in Haskell is not canonical because

Haskell does not support large types. Instead, it can be defined directly on the internal representation of Agda terms (lists of lists); this amounts to defining the same function as Agda would have constructed automatically during type-checking, and more importantly, consistently.

Pseudo built-ins. To simplify building in decision procedures that depend upon auxiliary functions, what is needed are constants, for which we can enforce certain defining equations without the need for the implementation to be overridden. These are called pseudo built-ins, however they could have been called axiomatised constants. Pseudo built-ins behave the same as non-built-in functions, i.e. they are not overridden; but they still have axiom checks. Essentially, they are Agda functions that we ensure during type-checking fulfil a number of defining equations (axioms), and in the same way that built-in functions are associated to a tag, they are also associated. Their purpose is to allow for the axiom check of the main decision procedure to be formalised. Therefore, using pseudo built-in functions for auxiliary functions alleviates difficulties with the decision procedure using dependently typed functions, they also reduce the number of types that require toTerm translations defined. As they are not overridden, there are less points in the system that could cause inconsistencies.

For example, in Section 5.4, to ensure that a proof of the soundness of the decision procedure is defined, a term of the following type is required (which is also a pseudo built-in):

$$\forall \varphi \,.\, \mathrm{T}\,(\mathrm{check}\,\varphi) \to \forall \xi \,.\, [\![\, \varphi \,]\!]_\xi$$

To formalise its type for the axiom check, requires that the function T is built-in (i.e. referencable). For the issues discussed above, this function is awkward to build-in, therefore, a pseudo built-in is used. This is achieved by checking that T fulfils the axioms $\mathrm{T}\ \mathrm{true} = \top$ and $\mathrm{T}\ \mathrm{false} = \bot$, and then associating T with a tag that can be used to later reference it. However, T is never overridden by a Haskell implementation. In the case of T, the terms true, false, $\top$ and $\bot$ would also need to be built-ins (as they are referenced).

The pseudo built-ins allow significantly complex decision procedures to be built-in with less work than before, because, all types and functions required by a decision procedure are declared as pseudo built-ins instead of normal built-ins. However, the decision procedure remains a normal built-in as its implementation will be overridden by a Haskell function.

5.3 Specific Branches

Two branches of the Agda source were taken, one that implements a connection to a SAT solver and the other to a CTL model-checker. These branches have the necessary axioms, for the decision procedures they implement, hard coded. These branches make use of both built-in and pseudo built-in techniques.

5.3.1 SAT

The SAT branch facilitates a non-dependent version of the algorithm from Section 4.1 to be defined as a built-in. This was the first attempt at connecting Agda to an external tool, and was implemented before the concept of pseudo built-ins was formalised; thus all referenced functions and data-types must also be built-in. Importantly, this was feasible for a non-dependently typed naïve SAT algorithm due to the small number of lines of code required.

It involved defining Bool := {true, false}, along with the functions $\_\wedge\_$, $\_\vee\_$ and $\neg\_$ as built-ins. Here, it is clear that there is little (or no) advantage when defining these trivial functions as built-ins, because there is an overhead when translateing terms between Agda and Haskell. In the case of natural numbers, which are already built-in, the max function was required, and it is overridden by Haskell's integer max function. The other built-ins added relate to the decision procedure.

First, the set of Boolean formulæ are defined in Agda and Haskell as

$$\text{BooleanFormula} := \{\text{true}, \text{false}, \_\wedge\_, \_\Rightarrow\_, \_\vee\_, \neg\_, \text{var}\ \_\}$$

where the variables are indexed by $\mathbb{N}$. The functions toTerm and fromTerm are canonical maps between the constructors. Then the function

$$\text{rank} : \text{BooleanFormula} \to \mathbb{N}$$

computes the index of the largest variable in the Boolean formula, and the function

$$\text{instantiate} : \text{BooleanFormula} \to \text{Bool} \to \text{BooleanFormula}$$

which instantiates variable 0 with the second argument and shifts all other variables down by one, i.e. maps variable x to $x - 1$. Two decision procedures are defined over BooleanFormula. The first is for formulæ with no variables, however, as this is non-dependent, it is required to assume that all variables encountered are assigned true. The second recursively makes a conjunction

that instantiates variable 0 to true in one conjunct and false in the other, the recursion is bounded by an arbitrary natural.

All the functions mentioned up-to this point must be built-in, and are required to be implemented 3 times.

Finally, the top-level decision procedure is given by the decision procedure for formulæ with variables, where the bound is given by the function rank applied to the formula. This is enforced by an axiom check of the decision procedure. The Haskell implementation of the top-level decision procedure would call an external SAT solver, pass it the negated formula presented in TPTP format [Sut09], wait for a result and return this result to Agda. (See module `Boolean.TPTP` in Appendix F for the TPTP presentation of a formula.) The result was either satisfiable (false), unsatisfiable (true), or error; the error result would be presented to the user as a type-checking error.

This amounts to 9 new built-in functions, one data-type, and 7 constructors, the Bool data type was already defined as a built-in. This required the addition of approximately 626 lines of source code to a vanilla 2.2.8 Agda. Listing of the SAT solver plug-in code is in Appendix D.

5.3.2 CTL

CTL model-checking theory as defined in Section 4.2 has significantly more structure than SAT solving. Using only non-dependently typed structures would have made the decision procedure and associated correctness proofs significantly more complicated. The theory is built around finite state machines, which in turn are built over finite sets of numbers. In the following, it is demonstrated how to build-in finite numbers.

Finite Numbers

Within Agda, finite sets (or enumeration sets) are defined using dependent types as follows:

$$\begin{aligned}&\text{data Fin} : \mathbb{N} \to \text{Set where}\\ &\quad \text{zero} \;:\; \forall\{n\}\;.\;\text{Fin}\;(\text{suc}\;n)\\ &\quad \text{suc} \;:\; \forall\{n\}\;.\;\text{Fin}\;n \to \text{Fin}\;(\text{suc}\;n)\end{aligned}$$

These are represented in Haskell by a pair of numbers (x, y), where x is the value, and y is the bound/size of the set. Elements of this are given by the partial[2] function mkFinNum which is only defined when $x < y$.

[2]Partiality is by the use of the Haskell term $(\text{undefined} :: a)$, for all a. It acts like a "bomb" that crashes the program when its value is computed.

```
data FiniteNumber = Fin Nat Nat

mkFinNum :: Nat -> Nat -> FiniteNumber
mkFinNum n s | (unNat n) < (unNat s) = Fin n s
             | otherwise = undefined
```

Assuming that the elements of FiniteNumber are only constructed by mkFinNum, then it is a simple matter defining finite numbers as a built-in data-type. The fromTerm and toTerm functions destruct/construct the inductive structure and are linear to the size of the number being translated.

Three built-ins were added, one for the type Fin, and one for each of its constructors.

A key issue here is that these finite numbers are efficient to work with as they are represented by machine integers, providing that a significant computation is carried out by a built-in function. This is because Agda does not support finite numbers in the internal syntax as literals in the same way that it supports natural numbers. For example, consider the addition of two finite numbers $a\ b : \mathrm{Fin}\ n$, there would be no advantage as these numbers would have to translated into machine integers from a number of applications of suc to zero, added together by hardware then translated back into an Agda representation by applying suc constructor $a + b$ times to zero. These are the same steps that would be carried out for addition when directly implemented as an Agda function, in fact, slightly worse as b would be destroyed and reconstructed instead of copied.

Moreover, this method of embedding a dependent type into Haskell is not mechanisable. In general, there are better methods that do scale, such as the approach identified by McBride in [McB02]. Recent versions of GHC with generalised algebraic and promoted data-types allow for a limited, native encoding of dependent products.

CTL Decision Procedure

The data-types for finite state machines and CTL formulæ were built-in. The CTL formulæ required one built-in for the type and one for each of the 8 constructors. The FSM's required one built-in for the type, one for the constructor and 6 pseudo built-ins for projections from the FSM. These projections were required when constructing the types of subsequent built-ins. None of the built-in functions return a FSM structure; thus it was not required to implement toTerm on FSM's. This meant that the process was simplified as the Haskell representation of a FSM is an over approximation due to dependent typed definitions in the Agda representation.

The two functions from Section 4.2.2 that added a sink state to the FSM

and adjusted the formulæ were pseudo built-ins, one further pseudo built-in was also required that ensured a proof of Theorem 4.2.4 (SinkCTL) was given.

The remaining 9 built-ins implement the CTL decision procedure; all except the top-level decision procedure are pseudo built-ins. These include basic functions such as the Boolean operators $\wedge$, $\vee$ and $\neg$; as well as more complicated functions that check (finite) runs of the transition system for properties. The implementation of these functions is hinted at in Section 4.2.

The final built-in is the top-level decision procedure. It is replaced by a Haskell function which executes NuSMV [CCG+02]. This is implemented in such a way that it first translates the CTL problem using the functions mkSink and liftCTL. Then this problem is canonically translated into a format that NuSMV understands by explicitly constructing the transition relation, and the result is passed to NuSMV. The result of NuSMV is handled with the same method as the SAT solver.

In total 32 built-ins were added (including 3 for the finite numbers), 3 types, 11 constructors, 11 pseudo and 1 traditional. This required 1,130 lines of code added to a vanilla Agda 2.2.8, and the plug-in/CTL library required 1,300 lines of Agda code.

CTL Example

As an example of whole the process, from entering a CTL problem in Agda, through to executing NuSMV, consider the transition system in Figure 5.2. Suppose EX (P 0) is the property to be checked to hold in the initial state 1.

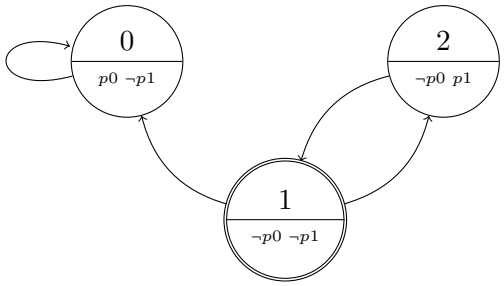


Figure 5.2: Simple Transition System

The problem is entered into Agda by creating an FSM and a CTL formula

that represents these structures as follows:

$$
\begin{array}{l}
\mathcal{M} : \mathrm{FSM} \\
\mathcal{M} = \mathrm{record}\ \{ \\
\qquad \begin{array}{lcll}
\mathrm{state} & = & 3 & ; \\
\mathrm{atom} & = & 2 & ; \\
\mathrm{arrow} & = & \{\ 1 \mapsto 2\ ;\ \_ \mapsto 1\ \} & ; \\
\mathrm{initial} & = & 1 & ; \\
\mathrm{transition} & = & \left\{ \begin{array}{l} 0\ 0 \mapsto 0\ ;\ 1\ 0 \mapsto 0\ ; \\ 1\ 1 \mapsto 2\ ;\ 2\ 0 \mapsto 1 \end{array} \right\} & ; \\
\mathrm{label} & = & \left\{ \begin{array}{l} 0\ 0 \mapsto \mathrm{true}\ ;\ 0\ 1 \mapsto \mathrm{false}\ ; \\ 1\ \_ \mapsto \mathrm{false}\ ;\ 2\ 0 \mapsto \mathrm{false}\ ; \\ 2\ 1 \mapsto \mathrm{true} \end{array} \right\} & \}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{problem} : \mathrm{CTLProblem} \\
\mathrm{problem} = \mathcal{M}, 1 \vDash \mathrm{EX}\ (\mathrm{P}\ 0)
\end{array}
$$

Here, CTLProblem is syntax that wraps up a compatible model, state and formula. When these definitions are applied to the decision procedure, they are implicitly given a sink state (state 0), and a new proposition 0 that only holds in the sink state. That is, the model is applied to the function mkSink, and the following model is obtained:

$$
\begin{array}{l}
\mathrm{mkSink}\ \mathcal{M} \equiv \mathrm{record}\ \{ \\
\qquad \begin{array}{lcll}
\mathrm{state} & = & 4 & ; \\
\mathrm{atom} & = & 3 & ; \\
\mathrm{arrow} & = & \{\ 0 \mapsto 1\ ;\ 1 \mapsto 2\ ;\ 2 \mapsto 3\ ;\ 3 \mapsto 2\ \} & ; \\
\mathrm{initial} & = & 2 & ; \\
\mathrm{transition} & = & \left\{ \begin{array}{l} \_\ 0 \mapsto 0\ ;\ 1\ 1 \mapsto 1\ ;\ 2\ 1 \mapsto 1\ ; \\ 2\ 2 \mapsto 3\ ;\ 3\ 1 \mapsto 2 \end{array} \right\} & ; \\
\mathrm{label} & = & \left\{ \begin{array}{l} 0\ 0 \mapsto \mathrm{true}\ ;\ 0\ 1 \mapsto \mathrm{false}\ ;\ 0\ 2 \mapsto \mathrm{false}\ ; \\ 1\ 0 \mapsto \mathrm{false}\ ;\ 1\ 1 \mapsto \mathrm{true}\ ;\ 1\ 2 \mapsto \mathrm{false}\ ; \\ 2\ \_ \mapsto \mathrm{false}\ ; \\ 3\ 0 \mapsto \mathrm{false}\ ;\ 3\ 1 \mapsto \mathrm{false}\ ;\ 3\ 2 \mapsto \mathrm{true} \end{array} \right\} & \}
\end{array}
\end{array}
$$

The above model and CTL formula are then automatically output in NuSMV format, note that for technical reasons the non-referencable atomic proposition -1 is required:

```
MODULE main
IVAR
  input : {0,1,2};
DEFINE
```

```
  labels := [{ -1,0},{ -1,1},{ -1},{ -1,2}];
VAR
  state : {0,1,2,3};
INIT
  state = 2;
TRANS
  next(state) =
    case
      state = 0 & input = 0 : 0;
      state = 1 & input = 0 : 0;
      state = 1 & input = 1 : 1;
      state = 2 & input = 0 : 0;
      state = 2 & input = 1 : 1;
      state = 2 & input = 2 : 3;
      state = 3 & input = 0 : 0;
      state = 3 & input = 1 : 2;
      TRUE : 0;
    esac;
SPEC
  EX ((1 in labels[state]) & (EG (!(0 in labels[state]))));
```

It is seen from this output that an arrow (input) and state have been added, the initial state is 2, and the transition (next) function is total. Each state has a set of atomic propositions (labels) attached to it, for technical reasons the set of labels for a state cannot be empty thus each set contains the label '-1'. This label is non-referenceable in Agda, and hence by the generated output.

The above output has a fixed number of outgoing arrows from each state, but the Agda representation allows for different states to have different numbers of outgoing arrows. Using the sink state this is simulated: any arrow taken that has not been defined in Agda will transition into the sink state (via the catch-all clause '`TRUE : 0`'). As the formula has been adjusted not to accept runs that pass-through the sink state this effectively simulates the Agda representation.

5.4 Generic Interface

The insights from implementing SAT and CTL as plug-ins for Agda as described in Section 5.3.1 and Section 5.3.2 highlighted a number of shortcomings. The most notable was that the decision procedures, in order to preserve soundness, required an axiom check which was equivalent to writing the decision procedure twice; once in Agda and once in Agda's source code. Without the pseudo built-ins, a third implementation would also be provided, namely

a Haskell function implementing the built-in function; the pseudo built-ins and motivation for them are discussed in Section 5.2.1.

This was thought to be an inhibiting factor of creating new plug-ins for different theories because an intimate knowledge of Agda's internals is required, and the ability/will to re-compile Agda. For this reason, a generic interface was explored and implemented. It does not require any per plug-in modifications to the Agda source code. The interface is implemented by using pseudo built-ins to reference the necessary Agda terms.

The generic interface does not perform an axiom check of the decision procedure, but instead requires that a proof of correctness is provided. There is also no possibility to determine whether the correct external tool is executed.

Remark

In Chapter 6 a sounder method is introduced. It is based on a different method than this generic interface, where axiom checks are not required, and the possibility of executing the wrong external tool is mitigated. However, the method presented here is faster for practical applications.

In total 6 pseudo built-ins and one standard built-in were added. The standard built-in executes the external tool indicated by one of the pseudo built-ins, and returns the result to Agda. Typically when defining a decision procedure, the problem set is defined first, then the decision procedure is given over this definition. This is modelled by the following two built-ins:

```
Problem : Set
{-# BUILTIN ATPPROBLEM Problem #-}

DecProc : Problem → Bool
{-# BUILTIN ATPDECPROC DecProc #-}
```

The second built-in depends upon the first, if omitted, then the second built-in will not type-check. In the case of SAT the problem set (Problem) is a propositional formula, and in the case of CTL the problem set is a 3-ary dependent pair of the model, state in the model and a CTL formula (that depends on the model).

Evaluating DecProc executes the external tool. This requires that Agda knows which external tool to execute, and how to translate Problem into a

format that the external tool understands. This is achieved by the following two built-ins:

```
Input : Problem → String
{-# BUILTIN ATPINPUT Input #-}

Tool : String
{-# BUILTIN ATPTOOL Tool   #-}
```

For completeness the sets Bool and String are built-in by the standard library.

In the interest of preventing malicious programs being specified, the tool is only specified by an identifier. The Agda configuration has been extended to allow for a selection of external tools to be specified by pairs of names and paths, see Section 5.5 for more information.

This provides Agda with enough information to execute the external tool, and wait for its response, but this is potentially unsound for the following reasons:

- The tool might be inconsistent with decision procedure, or
- the translation (Input) might not be correct.

To mitigate these issues, we recommend that the translation of the problem set into a string is canonical, and can be verified by a human. Any non-trivial transformations such as adding a sink state in CTL (Section 4.2.2) should be Agda functions, and should be proved to preserve correctness, although this is not required. This is different from many other ATP/ITP integrations that perform non-trivial translations outside the logic of the ITP, which is a primary motivation for an external tool providing a justification that can be mechanically checked. To increase trust that the input to the tool is correct, it can be inspected and manually checked on-demand for a given problem, this is because the translation is an Agda function.

If the tool is inconsistent with the decision procedure, this would be either accidental or malicious. If malicious, this technique provides no support[3]; moreover, any oracle based ATP/ITP integration is susceptible to malicious attacks. In order to mitigate accidentally entering the wrong tool, i.e. a SAT solver in-place of an SMT solver, there is also little support, but it is hoped that the user will notice strange behaviours. However if the user constructs the decision procedure incorrectly, then the interface does provide support.

[3]See the Linux command `false`, that always fails.

The user provides the semantics of the decision procedure (probably by reading from a book and entering it directly using Agda's Unicode support), then they must also provide a proof of correctness to enable use of the external tool.

The proof of correctness is not required for the evaluation of the decision procedure, but the intention is that it will assist the user to validate the plug-in before using it. This means that if the user has worked hard in proving that their decision procedure correctly implements their chosen theory, with respect to the semantics which they also provide. Then there is significantly less chance they would enter the wrong tool. It is recommended in the interest of soundness that a certified tool is used, but a widely used and trusted tool would also be admissible. The following three built-ins are now added:

```
Semantics : Problem → Set
{-# BUILTIN ATPSEMANTICS Semantics #-}

Sound : (γ : Problem) → T (DecProc γ)
                      → Semantics γ
{-# BUILTIN ATPSOUND Sound #-}

Complete : (γ : Problem) → Semantics γ
                         → T (DecProc γ)
{-# BUILTIN ATPCOMPLETE Complete #-}
```

Here, it is also required that the atom function (T) mapping Bool into Set is a pseudo built-in that fulfils the necessary axioms, which in turn requires that the unit type ⊤ and empty type ⊥ are built-in. To simplify what must be built-in, the correctness proof is split into two functions Sound and Complete. Otherwise, the product type would also have to be built-in. For a concrete example, see the built-in pragmas in modules `Boolean.CommonBinding` and `Boolean.SatSolver` of Appendix F.

When the function DecProc is reduced on an element of the problem set γ, the type-checker will first ensure that Sound and Complete have been provided and that the environment variable `AGDA_EXECUTE_PERMISSION` is set. Then it will execute the external tool named by Tool. It first applies the function Input to γ that yields a string in the tool's input language.

The Boolean valued result is computed by examining the return value of the tool. Should the tool return 0, a true value is used, should the tool return 1 then false is used; any other value results in a type-checking error being raised and the tools output dumped into the log. These values were chosen

to be in accordance with POSIX standards where 0 indicates success and 1 indicates failure. Typically it is required to write a simple wrapper for the external tool that parses the output and sets the return values appropriately. The implementation of DecProc is by the following Haskell function, albeit with the technical details omitted and function names changed for clarity. See Appendix D for the full code.

```
primitiveDecProc :: Term -> TCM Bool
primitiveDecProc t = do
  primitiveSound
  primitiveComplete
  input    <- primitiveInput t
  tool     <- primitiveTool
  path     <- lookupToolPath tool
  exitcode <- executeCommand path input
  case exitcode of
    1 -> return False
    0 -> return True
    _ -> throwError
```

Here, the primitive functions reference the corresponding Agda term that was tagged by the built-in declaration.

The generic interface has been tested with the SAT algorithm Section 4.1, notably the dependently typed version was used which also simplified the correctness proofs when compared to the non-dependently typed version. It was also used to implement CTL model-checking, symbolic CTL model-checking, and CTL model-checking of ladder logic programs, see Section 4.2, Section 4.3 and Section 9.4.1, respectively. In the case of model-checking a ladder, the function Input was replaced with an efficient translation that preserved the original structure of the ladder, see Figure 9.7.

The implementation of this generic interface required adding 164 lines to the Agda source code, circa development version 2.3.1.

Chapter 6 builds upon the generic interface presented here to facilitate proof reconstruction, this yields a very high-level of soundness as it mitigates issues where the wrong tool is executed.

5.4.1 SAT Evaluation

A number of experiments have been undertaken using the SAT solver plug-in, some toy problems and an industrial test case. The toy problems were used for testing purposes. Each toy problem consists of two data sets. The first is for generating the formulæ, executing the external tool and exploring the proof-object. The second set evaluates the same code as the first set but does not execute the external tool. This triggers the inefficient decision

procedure and typically will require exponential resources. Each of these data sets consists of a number of plots, these are:

Total CPU time taken to type-check the file, this includes garbage collection (GC) and waiting for the external tool to finish executing. CPU time was chosen over the actual time taken (from the users perspective) as the actual time is affected by a number of issues. These include caching, page swaps, multiple cores[4] and other processes competing for resources. The remaining plots breakdown the total time into is constituent components.

Mutator is the CPU time that was spent executing Agda's Haskell code. The mutator plot does not include the time that Agda waited for the external tool to complete execution.

GC is the CPU time that was spent performing garbage collection while running Agda, this statistic was gathered using the GHC RTS.

Tool is the approximate wall time[5] that the external tool took to decide the problem set. This include the time that wrapper scripts took to process the output of the tool to determine satisfiability/unsatisfiability.

Two toy problems were used to explore the integration. The problems were chosen to test different aspects of the integration. Problem set 1 is the law of excluded middle, and problem set 2 are the first unsatisfiable instances of the pigeonhole principle (PHP_n^{n+1}). Results relating to industrial problem sets can be found in Section 10.5 and a full application in Chapter 11.

Problem set 1 explores the limit of the number of variables that a formulæ can contain, such that the resulting proof-objects are feasible to explore. In a nut-shell, the formulæ have the form

$$x_{n-1} \vee \cdots \vee x_1 \vee x_0 \vee \neg(x_{n-1} \vee \cdots \vee x_1 \vee x_0)$$

where n is the size of the test. Let φ_n be the above formula. The majority of SAT solvers can trivially solve instances of this formula with many hundred variables as they uncover the underlying structure, namely the law of excluded middle. However, the naïve Agda

[4] Although Agda only uses one core, when executing external tools it is possible that the operating system executes part of Agda (mainly GC) while waiting for the tool to finish. This can result in > 100% core usage.

[5] The wall time is the total time taken for the external tool from the users perspective, for example it includes paging and IO waiting.

Remark

In early tests it was discovered that up to 95% of the total time was spent performing garbage collection (GC), this was mitigated by rewriting the tests/functions so they do not cause this undesirable behaviour from the GC. For example, the following function induces a lot of GC:

$$\begin{aligned} &\mathrm{f} : (n : \mathbb{N}) \to \mathrm{BooleanFormula}\ n \\ &\mathrm{f}\ 0 \quad\ = \ \mathrm{false} \\ &\mathrm{f}\ (\mathrm{suc}\ n) \ = \ \mathrm{var}\ 0 \vee \mathrm{mapsuc}\ (\mathrm{f}\ n) \end{aligned}$$

Here, mapsuc applies suc to each variable in the formula. The problem is that the function continually creates and destructs Boolean formulæ which is detrimental to the GC algorithm. This is because all Haskell terms are immutable when in memory, so for each update the whole structure is copied (in memory) before it is updated. The solution is to rework the above function as follows to mitigate this issue:

$$\begin{aligned} &\mathrm{f}' : (n\ m : \mathbb{N}) \to (g : \mathrm{Fin}\ n \to \mathrm{Fin}\ m) \to \mathrm{BooleanFormula}\ m \\ &\mathrm{f}'\ 0 \quad\ \_\ g \ = \ \mathrm{false} \\ &\mathrm{f}'\ (\mathrm{suc}\ n)\ \_\ g \ = \ \mathrm{var}\ (g\ 0) \vee \mathrm{f}'\ n\ \_\ (g \circ \mathrm{suc}) \end{aligned}$$

implementation of a SAT solver used in this work has the lower-bound complexity $\Omega(2^n)$.

A test of size n proceeds as follows,

- Type-check that

$$\text{tautology-sound}\ n\ \varphi_n\ \mathbf{tt}\ \xi$$

  has the type $[\![\varphi_n]\!]_\xi$, where ξ is the map $\{0 \mapsto \text{true}\ ;\ \_ \mapsto \text{false}\}$. The function tautology-sound is obtained from the correctness proof of the SAT solver, see Theorem 4.1.2. The decision procedure is triggered when type-checking:

$$\mathbf{tt} : \mathrm{T}\ (\text{tautology}\ n\ \varphi_n)$$

  If the decision procedure has been overridden by an external tool, then the tool is executed here, otherwise the naïve decision procedure is executed.

- The proof-object is computed by tautology-sound, and the instantiate lemma (Lemma 4.1.1). However, as all the variables have concrete values assigned by ξ, there is no exponential blowup, but a tower of n applications of Lemma 4.1.1 is constructed. This has a quadratic complexity. That is the following execution sequence, for clarity some of the arguments are omitted as they are large.

$$\begin{array}{l}
\text{taut-sound } n\ \varphi_n\ \text{tt}\ \xi \\
\downarrow \\
\text{lem-inst } n\ (\text{taut-sound } (n-1)\ (\text{inst } \varphi_n\ (\xi\ 0))\ \text{tt}\ (\xi \circ \text{suc})) \\
\downarrow \\
\text{lem-inst } n\ (\text{lem-inst } (n-1)\ (\text{taut-sound } (n-2) \\
\qquad (\text{inst } (\text{inst } \varphi_n\ (\xi\ 0))\ (\xi\ 1))\ \text{tt}\ (\xi \circ \text{suc} \circ \text{suc}) \\
\downarrow * \\
\text{lem-inst } n\ (\ldots\ (\text{lem-inst } 1\ (\text{taut-sound } 0 \\
\qquad (\text{inst } 1\ \ldots\ (\text{inst } \varphi_n\ (\xi\ 0))\ \ldots\ (\xi\ (n-1)))\ \text{tt}\ (\xi \circ \text{suc}^n)))\ldots)
\end{array}$$

  This is n applications of lem-inst to the Curry-Howard correspondence. Each application of lem-inst performs induction over the formula φ_n (or a partially instantiated instance).

- The resulting proof-object is explored to identify which variable has been assigned true. The proof-object consists of a number of applications of the sum type, so the exploration proceeds by repetitively breaking open the proof-object into the left or right until the variable which has been assigned true has been identified.

The tests are lazy, so if instead of setting variable 0 to be true, variable $n-1$ is set to true. This variable is the first to be inspected while exploring the result, so the result is computed much faster.

See Figure 5.3 for a cubic plot when the naïve SAT solver is overridden by Z3 [dMB08]; and see Figure 5.4 for a logarithmic plot when no external tools are used. In both of these plots, the mutator plot is the most significant as it is the actual time the type-checker required.

These plots show that connecting Agda to external SAT solvers yields at least a 10 fold increase in the number of variables that an instance of the excluded middle problem set can contain, whilst still being solvable in the same (or less) time. Execution without the external tool is exponential—it evaluates the naïve decision procedure. When using the external tool Z3 the complexity is reduced to $\approx n^{3.15}$. It is not clear why the complexity is worse than quadratic; possibly this is because of a lack of sharing in the type-checker, or general inefficiencies in the Agda source code. It is clear that as the size of the problem set grows, then so does the percent of time spent performing garbage collection;

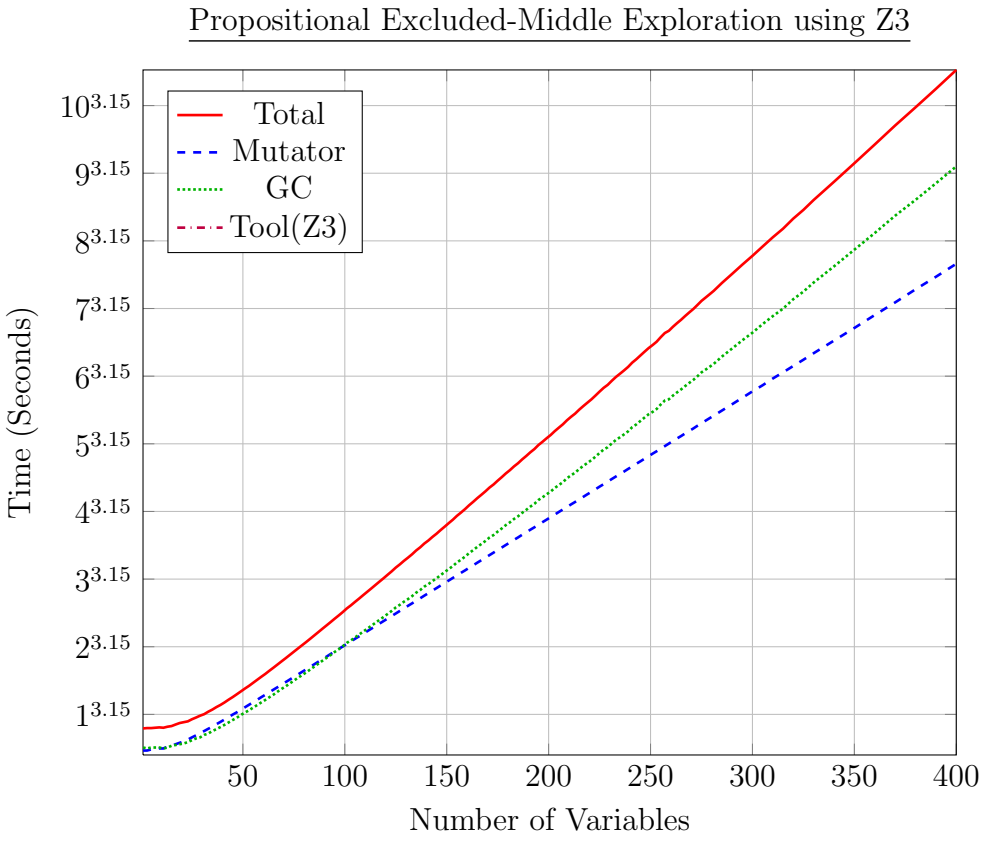


Figure 5.3: This plot shows the type-checker executing Z3, time spent executing Z3 was a at most 0.03 seconds and is not plotted. Greatest total time is 1660.69 seconds. 158 samples were taken, between 1-280 they were alternate, between 280-330 every 5, and between 330-400 every 10.

this is because as the memory becomes full the garbage collection cycles take longer[6]. For instance, the test with less than 100 variables spent roughly equal time executing the type-checker and performing garbage collection, but towards the end of the tests (circa. test 400) the garbage collection required almost 50% more time than the type-checker, i.e. 610 seconds vs 1049 seconds.

Problem set 2 is the Pigeonhole principle. The pigeonhole is widely used as a benchmark problem set for theorem provers, including SAT solvers, due to its hard complexity. The pigeonhole formulæ are denoted by

[6]This is due to the computer memory (RAM) filling-up, and the garbage collection algorithm reacting by compacting parts of the memory, see the GHC documentation for more information.

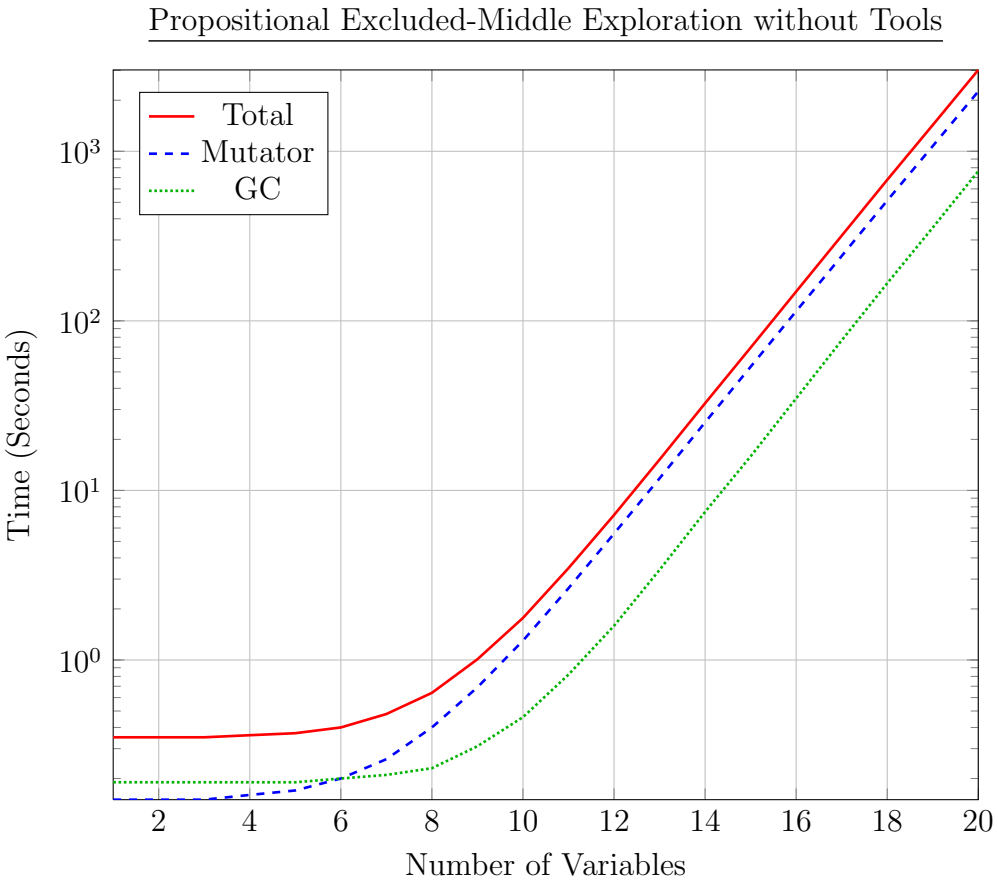


Figure 5.4: This plot shows the type-checker executing the naïve SAT solver. The lower bound is 2^n, and upper bound is slightly worse. The largest value for total time is 3,018.48 seconds.

PHP^n_m, which represents placing n pigeons into m holes. Each instance has $n * m$ propositions, $p_{i,j}$, that represent pigeon i is in hole j. The pigeonhole formulæ are defined as follows:

$$\text{PHP}^n_m = \left(\bigwedge_{i<n}\left(\bigvee_{j<m} p_{i,j}\right)\right) \Rightarrow \left(\bigvee_{i<n}\left(\bigvee_{j<m} p_{i,j} \wedge \left(\bigvee_{i'<n \wedge i \neq i'} p_{i',j}\right)\right)\right)$$

This formula intuitively has the semantics that if n pigeons are placed into m holes, then these exists a pigeon that shares its hole (with another pigeon). Clearly when $n > m$, instances of PHP^n_m are universally true. The instances used are PHP^{n+1}_n. These are the first unsatisfiable instances, and also have the greatest complexity to prove.

The purpose of these tests is not to explore how large the problem is,

but how the complexity of the problem affects computing a result. The tests proceed in a similar fashion as the previous tests of problem set 1 did, however, instead of the test searching for which variable is assigned true, it searches for which hole is occupied by more than one pigeon. First, for a test of size n, the following is type-checked

$$\text{tautology-sound } n \ \text{PHP}_n^{n+1} \ \mathbf{tt} \ \xi : [\![\text{PHP}_n^{n+1}]\!]_\xi$$

where ξ is the map $\{p_{i,0} \mapsto \text{true} \ ; \ \_ \mapsto \text{false}\}$, i.e. it puts all the pigeons in hole 0. Hole 0 is the last hole to be inspected below, thus this results in the worst case performance. The function tautology-sound is obtained from the correctness proof of the SAT solver, see Theorem 4.1.2. The decision procedure is triggered when type-checking:

$$\mathbf{tt} : \top \ (\text{tautology } n \ \text{PHP}_n^{n+1})$$

If the decision procedure has been overridden by an external tool, then the tool is executed here, otherwise the naïve decision procedure is executed. The same comments as said for problem set 1 about the computation hold here as well, i.e. that a tower of $n * m$ applications of instantiate are evaluated before a proof-object is obtained.

Inspecting the proof-objects of PHP_n^{n+1}, yields a function that produces a proof-object for any combination of $n+1$ pigeons being placed into n holes. Further inspecting this second proof-object reveals a hole that is shared by two pigeons. This inspection proceeds by repeatedly breaking open the sum type into a left and right, until it has identified a $p_{i,j}$ that shares its hole. However, it does not further explore the proof-object to determine which pigeon shares its hole.

See Figure 5.5 for a logarithmic plot when the naïve SAT solver is replaced by Z3. See Figure 5.6 for a logarithmic plot when no external tools are used.

These plots also show a significant speedup when using an external tool to solve the problem set. When no external tool was used it was not possible to explore instances that had more than 4 holes due to lack of resources. For this reason, it was not possible to analyse the results in full or construct a meaningful plot in Figure 5.6; it is however presumed to be exponential. In Figure 5.5, for $n \geq 9$, the majority of the time is used by the external tool. It is clear that solving instances of PHP require exponential time for Z3, inspecting the proof-objects also required exponential time, although of a lesser magnitude.

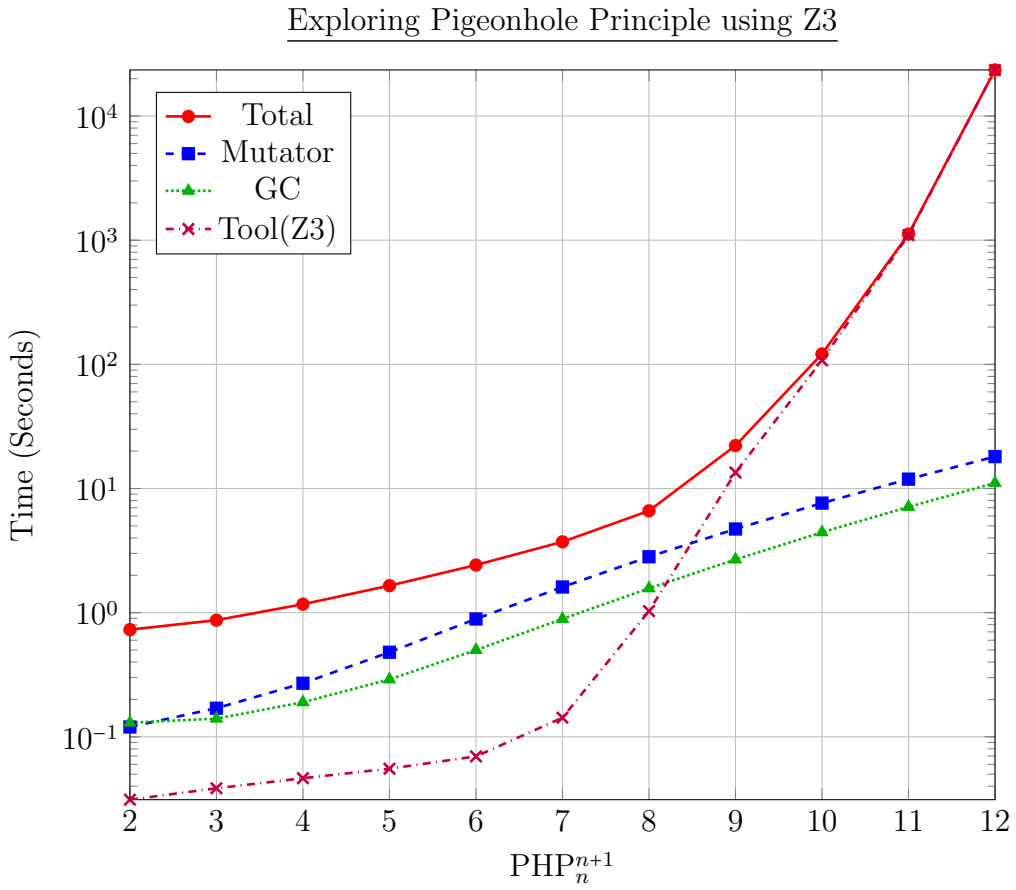


Figure 5.5: This plot shows the type-checker executing the external tool to solve instances of the PHP problem set. The majority of the time spent was for the external tool to solve the problem. The largest value for total time is 23,527.56 seconds.

Test Platform

All tests were carried out on a x86_64 GNU/Linux (Ubuntu Lucid) that had a x86_64 GHC 7.0.4. Agda 2.3.1 dated "Thu Feb 2 09:24:47 GMT 2012" was used for the tests, it was patched using the techniques described in this thesis. The GHC run-time system was provided with the option `-M7G` which specifies the maximum heap that can be allocated, this maintained efficient computations during type-checking by preventing use of the swap space. When the maximum heap is specified the GHC GC algorithm changes its behaviour, once > 30% of the heap is used. These changes result in an increased time performing GC, but less heap usage. More information can be

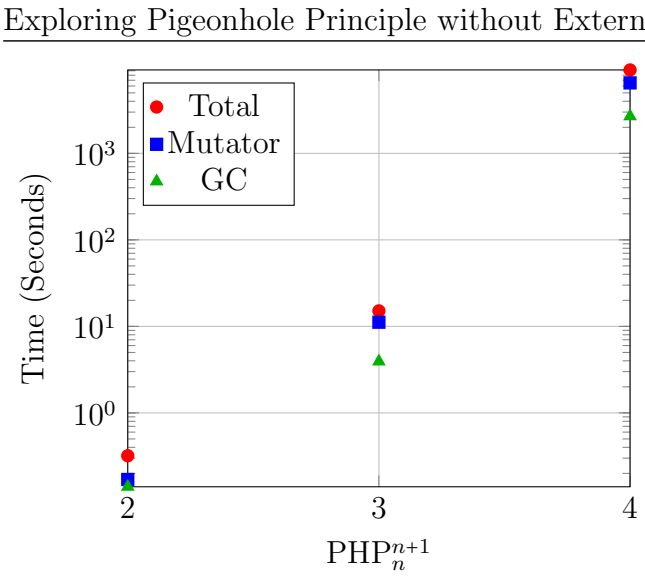


Figure 5.6: This plot shows the type-checker solving instances of the pigeonhole problem. The largest value for total time is 9,795.36 seconds.

Remark

In Figure 10.3, the time statistics of applying the verification framework considered throughout this thesis to an industrial problem are not competitive. In part, this is because the formula representing the problem set is generated by a function from a large data-type, and thus it must be normalised twice, once for the type, and once for the soundness proof/SAT solver.

found in the GHC documentation[7] under "compacting garbage collection".

More technical notes about the hardware: dual core CPU at 3GHz, 4MB cache; 8GB of dual channel RAM at 800MHz.

5.4.2 CTL Evaluation

A number of experiments were made with toy problems, such as the Pelicon crossing. Although even with a state space of size 3, Agda was unable to inspect the proof-object obtained. It was possible to execute the model-checker (NuSMV) successfully on the Pelicon crossing, but Agda was unable

[7] `http://www.haskell.org/ghc/docs/7.0.4`

to inspect any proof-objects within the given time limit. This is due to the highly inefficient decision procedures that were used, and the fact that the decision procedure is written mutually recursively. So it will execute parts of the naïve decision procedure while inspecting the proof-object. This is orthogonal to SAT where the decision procedure is not mutually defined, so each recursive call is replaced by a call to the SAT solver, resulting in significant performance increases.

Moreover, the various CTL interfaces used were never able to verify a substantial problem set of the industrial test case of an interlocking system written in ladder logic (cf. Section 9.2). The problem was translated into the tools input language by preserving as much of the structure of the original program as possible; that is a Boolean variable in the program mapped to a Boolean variable in the model, and assignments to these variables formed part of the transition function in the model. This failure to verify an actual interlocking system was due to the chosen model-checker (NuSMV) failing to terminate successfully within a generous amount of time. It was left running for ≈ 6 weeks, after which time execution was halted due to a segmentation fault. A number of different command line options were given to NuSMV, but they were unsatisfactory.

In conclusion, it became apparent that the CTL interfaces in their current form are unusable. For small toy problems, inspecting the proof-object is not feasible; and for larger problems, executing the external model-checker was not possible. In the next chapter, where certificates produced by external tools are checked, it is not clear what the resulting certificates for CTL model-checking would be, and whether Agda would have had enough resources to check them.

5.5 Security

The issue of securing against malicious code execution has not been discussed yet, with the exception of a reference to the environment variable `AGDA_EXECUTE_PERMISSION`.

By security, it is meant that it is not possible while type-checking an Agda module that a malicious program is executed. This is significant because type-checking a function, could require that other functions are evaluated. If the function being checked references the built-in decision procedure described previously, then it will execute the external tool. It is undesirable to download an Agda module, which, while type-checking, executes some malicious program, such as wiping the users home directory.

To prevent malicious programs from being executed consideration needs

to be payed to where the path to the executable is stored. In the specific branches, the program path is hard-coded into the Agda source code, and would require recompilation to change it. Securing a path by recompilation is secure enough for the average user, while still allowing the power-user to customise the program path.

The generic branch has the goal of not requiring to be recompiled for different solvers. Thus, the program path cannot be stored in the Agda binaries. Instead, two layers of security are applied the first disables the ability to execute external tools without having first explicitly enabled the functionality, and the second abstracts the path of the external tool out of the Agda modules.

The first layer disables/enables the ATP interface; it prevents accidental use of the interface without the user having explicitly enabled it. It is implemented by using the environment variable `AGDA_EXECUTE_PERMISSION`, it could equally have been implemented by adding a command line flag. If this variable is undefined, then type-checking any expression that requires the external tool will result in a descriptive type-checking error. The value that this variable is set to does not matter, although, during development, the value 1 was always used. When using Emacs mode, the variable can easily be set using the following command sequence

```
M-x setenv RET AGDA_EXECUTE_PERMISSION RET 1 RET
```

and then reloading Agda.

The second layer abstracts the path of the tool from the name of the tool; it is intended to prevent malicious programs being specified in the Agda modules. It works by configuring Agda with a list of paths and names, then the built-in string `ATPTOOL` references these names. The point is that the configuration of these names and paths is disjoint from the Agda modules. So in an Agda module, only the name of a tool is specified, and not the path. This means that the existing pragma system in Agda is not sufficient as the pragmas are specified in the modules. Also, using command line arguments is not sufficient as the Emacs mode does not use them. It is required that the configuration of the paths cannot be changed during an execution of Agda as this could result in inconsistencies. So it was decided to implement this layer by a further environment variable `AGDA_EXTERNAL_TOOLS`, that is formatted as follows:

```
tool1=path1;tool2=path2;...;tooln=pathn
```

This means that there is no possibility of updating the list of paths without modifying the variable and restarting Agda. As this variable must be formatted, as described above, the Emacs mode was extended to configure the

variable. Then, before GHCi/Agda is executed, it sets the variable accordingly. The variable is configured in Emacs mode by

```
M-x customize-group RET agda2 RET
```

then scrolling down to `Agda2 External Tools`, and expand it to edit. For example after configuring Agda to call Z3, the configuration might look like:

```
Agda2 External Tools: [Hide Value]
[INS] [DEL] Cons-cell:
            Tool Name: z3
            Executable Path: /opt/bin/z3
[INS]
   [State]: SAVED and set.
   Names and paths of external tools.
```

New tools are added by using the `[INS]` button, and old tools are removed by the `[DEL]` button.

5.6 Experimental Modifications

Understandably, a number of different modifications were thought-out, and some were implemented.

5.6.1 Basic Profiling by Counting

During the early phase of the project, there were a number of occasions where it was not clear why type-checking certain definitions, particularly in relation to decision procedures and the proofs of correctness was very time consuming. In many of the cases, the definitions could be checked by hand in a fraction of the time Agda took, due to knowing when and where to normalise a term. This is a well-known issue with Agda, and there is no clear solution. Currently the recommended solution is to place the troublesome definitions into an abstract block to inhibit the unfolding.

To help diagnose these problems and pinpoint the definition that was being unnecessarily normalised, a profiling feature for the type-checker was thought-out and implemented. This entailed adding a pragma to the Agda language which would match a regular expression against each function as it was unfolded. If the function name is accepted by the regular expression, then a counter is incremented. Each function has its own counter[8]. After

[8]The counters were provided by adding a hash-table to the type-checking monad that mapped Agda terms (functions) to integers.

type-checking, normalising a term, or inferring the type of a term, a tabular printout was placed into the log window that showed the matched functions and number of times they were unfolded. The pragma is of the form

{−# OPTIONS --regex-profile *REGEX* #−}

where *REGEX* is a regular expression. The pragma is placed in the preamble of the file to be type-checked, or given at the command line in the same way that the other pragmas are.

This profiling provided a limited method of exploring a computation in Agda, it was particularly adept at identifying functions that were applied many times. However, after this profiling had been applied a number of times it became apparent that it was simpler and faster to identify these terms by hand.

5.6.2 AIM − XIII

During Agda's Implementers Meeting (AIM) XIII, I participated in two small projects, both of which were accepted into Agda's source code. The first was to refactor the built-in mechanism as it was now being used in ways that it was never intended, and the second was to implement anonymous case-distinctions by extending lambda expressions.

Refactoring Built-Ins

The other participants were Nicolas Pouillard and Simon Foster who where both also interested in exploring an expansion of built-ins. The problem was that the machinery for the built-in system was originally intended to provide support for machine integers and strings, but soon grew (as did Agda) to support a number of advanced concepts such as propositional equality and co-algebraic data-types[9]. Although these features worked, they were in many cases hacked into Agda one-by-one resulting in code that was hard to understand, spread through many files and hard for normal users to modify for their own experiments.

We collated details about all the official built-ins that were in the Agda source code, along with our own views of uses of built-ins such as pseudo built-ins. Using this information 5 categories of built-ins were defined.

Data Types This includes the types of natural numbers and Booleans, given by a type (typically Set) and a list of constructor names. The

[9]These features are only fully supported by the type-checker once the correct definitions are given and declared as built-ins.

term tagged by one of these built-ins is then checked that its type matches the given type and that it has the correct number of constructors.

Constructors Almost self explanatory, this includes the constructors of data-types; each constructor is given by a type. The term tagged by one of these built-ins is then checked that its type matches the given type and that it is a constructor.

Primitive These are built-in functions that have a primitive (Haskell) implementation that replaces them for closed terms. These are given by their primitive implementation (from which the type is derived) and an axiom check, that if fails results in a type-checking error.

Postulates In Agda, there are a small number of types that are provided as black boxes such as Strings and floating point numbers. These are only used for communicating with the underlying system, i.e. when performing IO in a compiled program. It is possible to compute with these types, but only using primitive functions. These are defined as postulated built-ins and are only given by a type.

Other Anything built-in that does not fit into a previous category is put into the unknown category. This category has an optional type and an optional axiom check. The pseudo built-ins are implemented using this category by giving a type but omitting the axiom check. Moreover, this category can be used just to tag an arbitrary term in Agda by omitting the type and axiom check.

This facilitated one large list of all built-ins (except co-inductive) in Agda. More importantly it has simplified the process of adding custom built-ins—without intimate knowledge of Agda's internals—which is of benefit to the whole community.

These changes resulted in removing 57 lines from the Agda source code. The only noticeable change was that the IO type now had to be declared as a built-in, as it had previously been hacked into Agda. The work of refactoring the built-ins took two days.

Extended Lambda Expressions

The other participants were Fredrik Forsberg and Noam Zeilberger. Previously Agda only allowed simple lambda expressions, that in all but the absurd cases did not allow for case-distinction. That is,

$$\lambda\, x_0\ \ldots\ x_{n-1} \to t$$

and

$$\lambda\, x_0\ \ldots\ x_{m-1}\ ()$$

were valid expressions. This had the limitation that a non-absurd case-distinction on x_i was not possible. Moreover, the only method of performing a case-distinction in Agda was as part of a function definition, and their was no provision to include case-distinction within a term. Thus, we introduced new syntax that allowed for anonymous functions/in-line case-distinctions within terms. These had a very natural form of

$$\lambda\, \{\ \text{true} \to t_1\ ;\ \text{false} \to t_2\ \}$$

in the case of Booleans and in the case of natural numbers:

$$\lambda\, \{\ \text{zero} \to t_1\ ;\ (\text{suc}\ x) \to t_2\ \}$$

Before explaining how these were implemented, a brief outline of Agda's internal structure is required. There are three layers of syntax, first is a concrete representation of the file being type-checked. The second is a pre-processed syntax that aims to simplify the type-checking by normalising the input. The final layer is the result of type-checking the preprocessed input. It is also necessary to provide reverse translations (although not necessarily consistent) when presenting information/errors to the user. These layers can be seen in Figure 5.7.

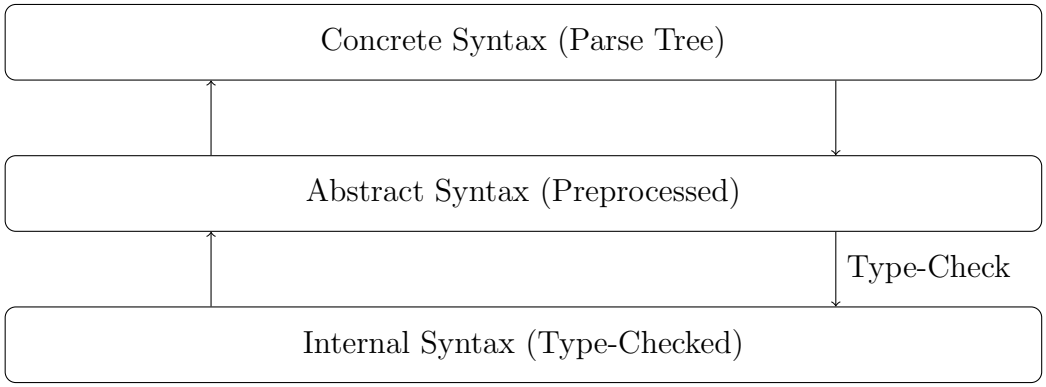


Figure 5.7: Agda's syntax architecture, based on notes taken from Andraes Abel (an Agda developer) at AIM XIII.

To implement the extended lambda expressions, the concrete and abstract syntaxes where extended with the extended lambda syntax. When translating from concrete to abstract syntax, a fresh function is generated

that realises the extended lambda, including its case-distinction. When type-checking the abstract syntax, the abstract lambda terms behave as function call sites.

There were still two issues remaining. The first is the reverse direction used for pretty printing, this required identifying which functions had been implicitly generated, and replacing their pretty print functions by an extended lambda pretty print. A second issue was the Emacs mode. It required modifications to enable support for interactive case-distinctions of extended lambdas. As the extended lambda's are already treated as functions inside Agda, the simplest option was to take advantage of this, and tweak the output (by string manipulations) in the Emacs mode. That is, the case-distinction procedure is called as normal, but before updating the Agda file with the output, for each case, the function names are dropped, the '=' are replaced by '→', and the '\n' are replaced by ';'.

5.7 Concluding Remarks

In this chapter, a number of modifications to Agda have been presented. These include (with respect to integrating external tools) two specific branches of SAT and CTL interfaces that offer high-levels of soundness due to the axiom check that is performed, and a generic interface that trades some soundness for usability. In the case of the SAT interface, a numerical analysis has been presented.

There were also experimental modifications that include the addition of extended lambda expressions, re-factoring of the built-in machinery and basic profiling of the head normal-form reductions.

5.7.1 Built-In Algebraic Data

If possible, adding built-in data-types should be avoided for two reasons: first if not done carefully, they can result in lower performance (with the exception of a few literals that Agda supports in the internal syntax) due to needless toTerm/fromTerm calls. Secondly, mapping dependently typed data to Haskell data (and back) must be done carefully to ensure consistent computations. See the previous discussion of declaring finite numbers as built-ins in Section 5.3.2.

5.7.2 Future Work

On the technical side, there are two more areas that would have been beneficial to explore. First, creating a modularised plug-in interface, and secondly using the quote goal feature to discover formula codes.

Externalised Plug-In Interface

As future work, it remains to implement a plug-in mechanism for external ATP tools. The idea being that all built-in data-types and functions are defined in pseudo-Agda files. These files are implicitly loaded and type-checked when a built-in pragma is found. This type-checking would enforce the axiom and inductive checks on functions and data-types, respectively.

This file would also define the corresponding Haskell and Agda terms, and any dependencies between built-ins.

For instance the natural numbers are characterised by their induction principle.

$$(A : \mathbb{N} \to \mathrm{Set}) \to A\ 0 \to (\forall n\ .\ A\ n \to A\ (\mathrm{suc}\ n)) \to \forall n\ .\ A\ n$$

This would be proved (trivially) for a standard definition of the natural numbers, and contained in the pseudo-Agda file as follows:

$$\begin{aligned} &\mathrm{f} : (A : \mathbb{N} \to \mathrm{Set}) \to A\ 0 \to (\forall n\ .\ A\ n \to A\ (\mathrm{suc}\ n)) \to \forall n\ .\ A\ n \\ &\mathrm{f}\ A\ p\ q\ 0 \quad\ = p \\ &\mathrm{f}\ A\ p\ q\ (\mathrm{suc}\ n) = q\ n\ (\mathrm{f}\ A\ p\ q\ n) \end{aligned}$$

When type-checking the built-in pragma for natural numbers, the above definition (and proof of) would be type-checked. The only significant issue here is a mapping between the names (0, suc, $\mathbb{N}$) used by the definition being built-in, and those used in the pseudo-Agda file.

Furthermore, in the case of addition on natural numbers, there are two defining equations. Therefore, the two formulæ contained in the pseudo-Agda file that must be type-checked are:

$$(A : \mathbb{N} \to \mathrm{Set}) \to \forall\ n\ .\ A\ (n + 0) \to A\ n$$

and

$$(A : \mathbb{N} \to \mathrm{Set}) \to \forall\ n\ m\ .\ A\ (n + (\mathrm{suc}\ m)) \to A\ (\mathrm{suc}\ (n + m))$$

Provided $\_+\_$ is defined correctly, then the proofs of these formulæ are trivial. For instance the following two functions are defined:

$$\begin{aligned} &\mathrm{g} : (A : \mathbb{N} \to \mathrm{Set}) \to \forall\ n\ .\ A\ (n + 0) \to A\ n \\ &\mathrm{g}\ A\ n\ p\ =\ p \end{aligned}$$

$$\mathrm{h} : (A : \mathbb{N} \to \mathrm{Set}) \to \forall\ n\ m\ .\ A\ (n + (\mathrm{suc}\ m)) \to A\ (\mathrm{suc}\ (n + m))$$
$$\mathrm{h}\ A\ n\ m\ p\ =\ p$$

Here, there is as well an issue of mapping the $\_+\_$ from the function being built-in, to the pseudo-Agda file. There is also the added complexity that addition depends upon the natural numbers being built-in first, thus (0, suc, $\mathbb{N}$) also need to be mapped to the relevant terms.

Quote Goal

Another area that could be explored is to use the *quote goal* feature of Agda. It is a language construct that provides access to an internal representation of the current goal. This representation is given by a built-in data-type of terms, such as literals, applications and variables.

It was suggested by Makoto Takeyama (an Agda developer) that the quote goal mechanism could be used to analyse the given goal, and generate a code of the formula representing the original problem, which could be passed to an external tool to determine the validity. That is, given the problem of proving $\mathrm{T}\ (p \vee \neg p)$ always holds for some $p : \mathrm{Bool}$, the quote goal would give the underlying term, such that is possible to deduce that T is applied to $p \vee \neg p$, $\vee$ is applied to p and $\neg p$, and that $\neg$ is applied to p. Thus, it is possible to generate a propositional formula representing this goal, namely $p \vee \neg p$. Currently, when these codes are needed they are manually entered into the soundness proof.

Quote goal should be compatible with our technique. It is already used for the purpose of producing formulæ for solvers, e.g. the ring-solver in the standard library.

Chapter 6

Reconstructing Justifications

The ATP integration approach (Oracle + Reflection), which has been described so far, is limited in that it assumes that the external tool operates correctly. However, there are occasions where an exceedingly high-level of assurance is required, and assuming that the tool operated correctly is insufficient. For these cases, the (Oracle + Justification) approach was implemented: that is the justifications produced by the external tools are checked to be correct and converted into proofs. The implementation is based on the work of Armand et al. in [AGST10], where Coq [The04] was extended by a function that would call the SAT solver zChaff [MMZ+01], and then type-check the resulting refutation proof. However, our implementation is more generic as it allows for the justifications to give a direct proof (i.e. natural deduction), as well as a proof by refutations.

In this chapter, techniques are built-up that allow this interface to be defined in type theory, and then an example is presented of integrating the propositional fragment of the first-order theorem prover eProver [Sch02, Sch04]. eProver was used to generate a propositional calculus *proof* for a propositional Agda formula. By proof it is meant a data structure representing a derivation in a natural deduction system [Pel99, Gen35]. The proof is then type-checked to obtain a proof-object. There are many SAT solvers that emit refutation traces, but few that include in the derivation, details of how the formula was translated into a conjunctive normal form (CNF). These low-level justifications were the motivation for selecting eProver. This is because it mitigates having to write an Agda function to translate formulæ into CNF efficiently and in a way that preserves enough structure of the original problem to help the solver.

Chapter Overview. First, in Section 6.1 an Agda model of an abstract inference rule system is provided. In Section 6.2 the details of how to soundly connect an instance of this rule system to an external tool are discussed.

Finally in Section 6.3, the specific case of successfully integrating eProver is discussed and results are presented.

6.1 Inference Rule System

The proof reconstruction interface has been designed to be generic, such that it is possible to embed an arbitrary logic based on inference rules, provided the logic is provable in Agda. The rule systems specified here form a natural deduction system, and the proofs in this system are represented by a list of derivations. To simplify the process of interpreting the obtained justifications, the definition of a derivation rule is in two parts: a syntactical correctness check, and a soundness proof. The correctness and soundness of the rules are then lifted to the derivation lists. Such that from a proof of the correctness of a derivation list applied to the soundness proof an Agda proof-object is obtained for the theorem proved by the derivation.

Remark

The following has all been formalised in Agda, see module `Proof.List` in Appendix F.

First the logic that the external tool is built over (e.g. propositional logic) requires formalising. The logics are defined by a set of formulæ Φ, a set of environments Ξ, and a semantic relation $[\![\_\vDash\_]\!]$ between the environments and formulæ. In the following it is assumed that the formulæ in Ψ are sequents, however, this is not made explicit until later sections. This is a similar definition to that described in Chapter 4.

For a given logic, a rule system Δ is defined. They are defined by a set of inference rules. These inference rules are formalised by a set of rule identifiers Δ_{ids}, and the semantics of the rules are given by the following two functions. The first assigns an arity to each rule:

$$\text{arity} : \Delta_{ids} \to \mathbb{N}$$

The second defines a decidable correctness relation on inferences of the rules. That is, from the correct number of correct premises, the result is correct.

$$\text{correct-rule} : (\delta : \Delta_{ids}) \to \text{Vec}_{\Phi}\ (\text{arity}\ \delta) \to \Phi \to \text{Bool}$$

This function typically checks that an inference of the rule is syntactically correct, and must be decidable by the type-checker (hence the result is Bool

and not Set), or the proof reconstruction will fail. Determining the correctness of the rule's application by syntactical methods requires that Φ has decidable propositional equality, i.e.

$$\_==_{\Phi}\_ : \Phi \to \Phi \to \text{Bool}$$

Remark

The way that the correctness check has been specified above means that the bookkeeping of assumptions must be internalised into Φ. This is because each rule is considered in isolation and does not know the context. In most cases, this will result in Φ being defined as a sequent. There are situations where assumptions are not required, for example, resolution proofs in SAT.

Finally, the soundness of the rule system must be proved. The soundness will reify from a proof that the rule has been correctly used, a function that takes the correct number of semantic arguments (premises) and yields a semantic object (conclusion). The soundness of a rule system Δ is given as follows:

$$\begin{aligned}
\text{sound-rule} \ : \ & (\delta : \Delta_{ids}) \\
& \to (\mathit{premises} : \text{Vec}_{\Phi}\ (\text{arity}\ \delta)) \\
& \to (\mathit{conclusion} : \Phi) \\
& \to \top\ (\text{correct-rule}\ \delta\ \mathit{premises}\ \mathit{conclusion}) \\
& \to \text{Vec}^{*}_{\lambda\varphi \to \forall \xi\ .\ [\![\ \xi \vDash \varphi\ ]\!]}\ \mathit{premises} \\
& \to \forall \xi\ .\ [\![\ \xi \vDash \mathit{conclusion}\ ]\!]
\end{aligned}$$

Here, $\text{Vec}^{*}_{f} v$ is a dependently typed vector whose length is determined by the vector v, the types of the elements in $\text{Vec}^{*}_{f} v$ are defined by the family f which is indexed by elements in v. For instance, the i^{th} element in $\text{Vec}^{*}_{f} v$ has the type of the i^{th} element in v applied to f. It is defined as follows:

$$\begin{aligned}
&\text{data}\ \text{Vec}^{*}\ \forall\{A\}\ .\ (F : A \to \text{Set}) : \forall\{n\}\ .\ \text{Vec}_A\ n\ \text{where} \\
&\quad [] \ \ : \ \text{Vec}^{*}_{F}\ [] \\
&\quad _::_ \ : \ \forall\{n\ a\}\ \{v : \text{Vec}_A\ n\}\ .\ (x : Fa) \to (xs : \text{Vec}^{*}_{F}\ v) \to \text{Vec}^{*}_{F}\ (a :: v)
\end{aligned}$$

Derivations in Δ are then defined as simply typed syntactic objects, represented by a list of proof nodes. Each proof node in the derivation represents an application of a rule to a list of numeric references to previous points in

Remark

In the following, the use of simple types was initially because of a technical restriction when interfacing with Haskell code (via the built-in mechanism). Subsequent improvements meant that this restriction vanished, but it was observed that preserving the simple typed definitions resulted in better performance.

the derivation list, which form the premises; and to a formula that is the conclusion of the rule. Thus, the proof nodes only depend on the sets Δ_{ids} and Φ and do not enforce correctness.

$$\begin{array}{l}\text{record ProofNode}_{\Phi}^{\Delta_{ids}} : \text{Set where}\\ \quad \text{constructor}\\ \quad\quad \text{proofnode}\\ \quad \text{fields}\\ \quad\quad \text{rule} : \Delta_{ids}\\ \quad\quad \text{formula} : \Phi\\ \quad\quad \text{premise} : \text{Vec}_{\mathbb{N}}\ (\text{arity rule})\end{array}$$

A (possibly incorrect) derivation is then canonically given as a list of nodes. That is

$$\text{ProofList}_{\Phi}^{\Delta_{ids}} = \text{List ProofNode}_{\Phi}^{\Delta_{ids}}$$

where the head of the list is the first derivation, and the last node in the list is the conclusion.

It is then possible to define the correctness of a derivation with respect to the correctness defined in the rule system.

$$\text{correct-list} : \text{ProofList}_{\Phi}^{\Delta_{ids}} \to \text{Bool}$$

This proceeds by structural induction over the list, checking that each node references valid premises that occurred previously in the derivation, and that the premises and conclusion have the correct syntactical structure. The intuition is that this correctness check can be executed implicitly by the type-checker to determine if the external tool's output is a correct derivation.

Assume a theorem $\varphi : \Phi$, a derivation of φ is a correct proof list l, such that φ is the conclusion of root of the list.

$$\begin{array}{l}\text{derivation} : \Phi \to \text{ProofList}_{\Phi}^{\Delta_{ids}} \to \text{Bool}\\ \text{derivation } \varphi\ [\,] \;=\; \text{false}\\ \text{derivation } \varphi\ l \;=\; \varphi ==_{\Phi} \text{formula}_{(\text{last } l)} \wedge \text{correct-list } l\end{array}$$

From a proof that the list is correct it is possible by using the soundness proof of each rule to reconstruct a proof-object of the desired theorem. This proceeds in the same method as checking the correctness, i.e. by structural induction over the list. By Theorem 6.1.1, a function of the following signature is obtained.

$$\begin{aligned}\text{sound-list} : (\varphi : \Phi) \to (l : \text{ProofList}_{\Phi}^{\Delta_{ids}}) \\ \to \mathrm{T}\,(\text{derivation}\;\varphi\;l) \to \forall \xi\;.\;[\![\;\xi \vDash \varphi\;]\!]\end{aligned}$$

Theorem 6.1.1. *Assume theorem φ, let l be a derivation of φ in a sound rule system such that* derivation $\varphi\;l$ *holds, then*

$$\forall \xi\;.\;[\![\;\xi \vDash \varphi\;]\!]$$

holds.

Proof is by simple induction on l; it then follows by the proof of correctness and soundness for each rule.

6.1.1 Classical Propositional Logic

To illustrate the use of the rule system, consider the standard example of classical propositional logic. A full formalisation of this section is in module `Proof.PropLogic` of Appendix F. Although canonical, for completeness the inference rules of classical propositional logic are repeated in Figure 6.1. The propositional formulæ used are a simply typed variant of the propositional formula defined in Section 4.1, where the variables are indexed by natural numbers instead of finite numbers. As the inference rules require that assumptions are made, the set of formulæ are defined in *sequent-style*. That is the set of formulæ Φ is given by pairs of contexts (given by a list of formulæ) and a formulæ. To aid readability a sequent is defined using the following data-type:

$$\begin{aligned}&\text{data}\;[\_\Rightarrow\_]\;(A : \text{Set})\;(B : \text{Set}) : \text{Set}\;\text{where} \\ &\quad \_\Rightarrow\_\;:\;\text{List}\;A \to B \to [A \Rightarrow B]\end{aligned}$$

The set of formulæ Φ is defined as follows:

$$\Phi = [\text{BooleanFormula} \Rightarrow \text{BooleanFormula}]$$

The environments are functions mapping natural numbers to Boolean values.

$$\Xi = \mathbb{N} \to \text{Bool}$$

The semantic relation is canonical with respect to propositional logic, Curry-Howard isomorphism, and the environment.

$$\frac{\Gamma_1 \Rightarrow \varphi \quad \Gamma_2 \Rightarrow \psi}{\Gamma_1 \cup \Gamma_2 \Rightarrow \varphi \wedge \psi}\wedge^+ \qquad \frac{\Gamma \Rightarrow \varphi \wedge \psi}{\Gamma \Rightarrow \varphi}\wedge_l^- \qquad \frac{\Gamma \Rightarrow \varphi \wedge \psi}{\Gamma \Rightarrow \psi}\wedge_r^-$$

$$\frac{\Gamma \Rightarrow \varphi}{\Gamma \Rightarrow \varphi \vee \psi}\vee_l^+ \qquad \frac{\Gamma \Rightarrow \psi}{\Gamma \Rightarrow \varphi \vee \psi}\vee_r^+ \qquad \frac{\Gamma_1 \Rightarrow \varphi \vee \psi \quad \Gamma_2 \Rightarrow \rho \quad \Gamma_3 \Rightarrow \rho}{\Gamma_1 \cup (\Gamma_2 \setminus \{\varphi\}) \cup (\Gamma_3 \setminus \{\psi\}) \Rightarrow \rho}\vee^-$$

$$\frac{\Gamma \Rightarrow \psi}{\Gamma \setminus \{\varphi\} \Rightarrow \varphi \to \psi}\to^+ \qquad \frac{\Gamma_1 \Rightarrow \varphi \to \psi \quad \Gamma_2 \Rightarrow \varphi}{\Gamma_1 \cup \Gamma_2 \Rightarrow \psi}\to^-$$

$$\frac{\Gamma \Rightarrow \bot}{\Gamma \Rightarrow \varphi}\text{efq} \qquad \frac{\Gamma \Rightarrow \bot}{\Gamma \setminus \{\neg\varphi\} \Rightarrow \varphi}\text{raa} \qquad \frac{}{\Gamma \cup \{\varphi\} \Rightarrow \varphi}\text{ax}$$

Figure 6.1: Classical Propositional Inference Rules

These 11 rules represent an instance of the rule system. Each rule has an identifier and an arity, which are canonically definable from Figure 6.1, i.e.

$$\Delta_{ids} = \{\wedge^+, \wedge_l^-, \wedge_l^-, \cdots, \text{raa}, \text{ax}\}$$

and

$$\text{arity} = \{\wedge^+ \mapsto 2\ ;\ \wedge_l^- \mapsto 1\ ;\ \wedge_l^- \mapsto 1\ ;\ \cdots\ ;\ \text{raa} \mapsto 1\ ;\ \text{ax} \mapsto 0\}$$

The formation of these rules define the syntactical correctness of the rule system. Restricting the contexts in the conclusion of a rule instead of explicitly weakening the context of the respective antecedents was chosen to simplify the Agda implementation. The correctness is formulated as follows:

$$\begin{aligned}
&\text{correct}\ \wedge^+ \begin{bmatrix}\Gamma_1 \Rightarrow \varphi_1,\\ \Gamma_2 \Rightarrow \varphi_2\end{bmatrix} (\Gamma_3 \Rightarrow \varphi_3) = (\varphi_3 \equiv (\varphi_1 \wedge \varphi_2)) \ \wedge\ (\Gamma_1 +\!\!+ \Gamma_2 \sqsubseteq \Gamma_3)\\
&\text{correct}\ \ \vdots \qquad \vdots \qquad\quad \vdots \qquad = \qquad\quad \vdots\\
&\text{correct raa}\ [\Gamma_1 \Rightarrow \varphi_1]\ (\Gamma_2 \Rightarrow \varphi_2) = (\varphi_1 \equiv \text{false}) \ \wedge\ (\Gamma_1 \setminus (\neg\varphi_2) \sqsubseteq \Gamma_2)
\end{aligned}$$

where

$$[\varphi_1, \varphi_2, \ldots, \varphi_n] \sqsubseteq [\psi_1, \psi_2, \ldots, \psi_m] = \bigwedge_{i=1}^{n}\left(\bigvee_{j=1}^{m} \varphi_i \equiv \psi_j\right)$$

and

$$[\varphi_1, \varphi_2, \ldots, \psi, \ldots, \varphi_n] \setminus \psi = [\varphi_1, \varphi_2, \ldots, \varphi_n]$$

Note that the omitted cases of the correctness follow trivially from Figure 6.1 by appropriately reusing/modifying the above definitions.

All that remains is to show the soundness of each rule. As Agda is based on intuitionalistic type-theory, all of these rules except *reductio ad absurdum* hold in general and follow by the Curry-Howard isomorphism.

In the case of reductio ad absurdum (raa), it is required to show that the law of excluded middle holds for propositional formulæ intuitionalistically. It trivially holds for atomic propositional formula (by case-distinction), and is shown to hold for the propositional connectives, if it holds for their operands. Therefore by induction it holds for arbitrary propositional formula, and is used to prove soundness of the raa rule.

Thus from a derivation for the theorem $[] \Rightarrow \varphi$ that is proved to be syntactically correct, a proof-object of the form $\forall \xi \,.\, [\![\, \xi \vDash \varphi \,]\!]$ is obtained.

It should be noted that SAT solvers do not typically emit derivations using this set of (low-level) rules; instead their derivations are high-level functions that compute normal-forms and introduce intermediate variables that are equivalent to sub-formulæ. This is the case of eProver and is discussed in Section 6.3. However, first (in the next section) it is discussed how to connect Agda in a sound way to external tools to obtain a ProofList.

6.2 Primitive Implementation

This section explains how to implement the (Oracle + Justification) approach described in Section 2.3.1. Note that this is orthogonal to the (Oracle + Reflection) approach presented in previous chapters.

The implementation presented here is based upon the work of Armand et al. [AGST10] where they used reflexive methods to type-check SAT solver traces in Coq. This entailed writing a Coq program that had as inputs a clause set and a refutation trace produced by zChaff, and then determined whether the refutation trace was correct with respect to the clause set. This program was then executed by Coq after the SAT solver had been executed to determine the correctness of the result. See Section 2.3.1 for more information on the general technique. The paper also presents promising results that compete with the current *gold standard* of proof reconstruction in ITP tools.

In this work, the implementation is abstracted from the concrete ATP theories. This has the advantage that a simple bidirectional interface is provided to connect Agda to ATP tools, and more generally, any external tool. This simplicity is at the expense of being able to craft individual and efficient data-types that Agda uses to communicate with the ATP tool; that is no data-types dependent on the rule system, or instances of the rule system are built-in. Instead, we rely on Agda building the input string for the tool,

and parsing the output string from the tool. In fact, Agda's internal parser is used, and thus it is possible to parse Agda terms directly, as if they had been manually typed into the Agda code.

In the following, two primitive functions are introduced that allow for the execution of arbitrary external programs in a safe way. The first of these functions is a parser and illustrates the techniques used by the second primitive function, which, in addition executes an external tool. These primitive functions are independent of the rule systems discussed previously.

Recall that a primitive function behaves as if it had been postulated, except that it reduces on closed terms, for which it reduces in one step. Following by Theorem 5.1.2, a primitive function is sound to use provided for all possible inputs, the result type is never empty, and the implementation of the function is compatible with Agda's logic. Therefore, following by the first constraint that the result type is never empty: it is safe to define the following primitive function that takes as input a string, and returns a maybe result of some type.

$$\begin{array}{l}\text{primitive}\\ \quad \text{primParser} : (A : \text{Set}) \to \text{String} \to \text{Maybe}\ A\end{array}$$

Of course, this function could always return nothing, but consider the scenario when it attempts to parse the string using Agda's parser, and then type-checking that the resulting term has type A. If it does have type A, then the function returns the result, however, if parsing fails, or type-checking fails, then nothing is returned. That is the following Haskell function provides the implementation, albeit with the technical details omitted and function names changes to make the definition clear.

```
primParser :: Type -> String -> TCM (Maybe Term)
primParser ty s = catchError
                  (do t <- parse s
                      t <- typeCheck t ty
                      return $ Just t)
                  (return Nothing)
```

It is easy to see that this function will always return a result, hence by Theorem 5.1.2 provided the parser it implements can be defined in Agda, at least in principle, then it is a conservative extension. Naïvely thinking, parsers can be implemented in Agda directly so the function should be definable in Agda directly; however, this is not the case because the Agda parser depends upon the context/scope from which it is called. This is because as new definitions (functions, data-types, etc.) are made, the Agda parser is extended with their names, for instance the following code sequence is

inconsistent:

```
A : primParser ℕ "f 0" ≡ nothing
A = refl

f : ℕ → ℕ
f = suc

B : primParser ℕ "f 0" ≢ nothing
B = λ ()

C : ⊥
C = B A
```

The problem here is that the language that the parser recognises is extended between type-checking A and B, from which an inconsistency is derived.

To repair this problem the language that primParser recognises is fixed when the primitive function is defined. That is when the code

```
primitive
  primParser : (A : Set) → String → Maybe A
```

is type-checked, the current scope (e.g. function definitions, data-types, imported modules) is saved. Then when primParser is reduced this saved scope is recalled, and the term is parsed in the original scope. When importing a module hierarchy that has defined primParser, the current module cannot re-define it, nor can it import a second module that has also defined primParser. This effectively fixes the language recognised by primParser. As the parser is written in Haskell, it is reasonable to assume that a specialisation of the parser fixed to this language could have been written directly in Agda.

This concludes the first primitive function; however, it does not form part of the interface to the external tool and is only presented to illustrate the functionality of the parser. The second primitive function is an extension of the first, such that instead of parsing a string provided by the user, it parses the string obtained from the output of executing an external tool. There is a *caveat* relating to the external command, namely that the command must be functional in nature. That is, for two equal inputs the outputs should be equal. When using the above function to execute an external decision procedure this is not a problem, provided the obtained term $a : A$ does not

contain any superfluous information, such as time/date that the tool was executed.

The second function is parametrised by two strings. The first string is the name of the command to execute, see Section 5.5 for information about the name of the tool and how it is translated into a path. The second string is fed to the external tool as input. The function has the following signature:

$$\begin{array}{l}\text{primitive}\\ \quad \text{primExternal} : (A : \text{Set}) \to (\mathit{tool}\ \mathit{input} : \text{String}) \to \text{Maybe}\ A\end{array}$$

In such a situation the primitive function executes the command indicated by *tool*, passes it the string *input* and parses the output of the tool using the Agda parser and type-checker, as before. That is the following Haskell implementation is obtained, as usual the technical details are omitted and function names changed to aid readability.

```
primExternal :: Type -> String -> String -> TCM (Maybe Term)
primExternal ty tool input = do
  path   <- lookupToolPath tool
  output <- executeCommand path input
  scope  <- lookupParseScope
  catchError
    (do t <- parseInScope scope output
        t <- typeCheck t ty
        return $ Just t)
    (return Nothing)
```

Thus, primExternal represents an external command in Agda. As was discussed previously, the parsing functionality is safe to use provided the scope is fixed and the external command is referentially transparent.

This concludes the second primitive function, and the underlying mechanism to call external tools. This second function provides a powerful mechanism to connect Agda to various different tools. It is emphasised that not only theorem provers in the traditional sense are compatible, but in general, it is possible to delegate large computations to custom, unverified programs. Then these programs compute the result and produce a certificate that can easily be checked by an Agda program. This method of using unverified tools that produce checkable certificates was identified by de Bruijn in [dB70], where he gave the example of the magic square problem. In fact, it is possible for the external tool to provide semantic proofs directly, but this option was not explored.

6.2.1 Type-Checking Derivations

In this subsection (and the next) assume a rule system Δ that has been defined for the external theorem prover X, such that the set of formulæ are defined in a sequent-like manner. That is, let Φ' be the set of formulæ X is defined over, then

$$\Delta_\Phi = [\Phi' \Rightarrow \Phi']$$

and the following function is defined:

$$\text{toString}_X : \Phi' \to \text{String}$$

It transforms a formula into a string in the tool's input language.

Define the function which calls the external tool, and then attempts to reconstruct a derivation from the result. However, as the formulæ are defined by sequents and in this work, we are interested in theorems, so the left-hand side of the sequent is omitted, i.e.

$$\begin{aligned}&\text{createList} : \Phi' \to \text{Maybe ProofList}^{\Delta_{ids}}_{\Delta_\Phi}\\&\text{createList } \varphi \;=\; \text{primExternal “id of } X\text{”} \; (\text{toString}_X \; \varphi)\end{aligned}$$

where “id of X” is the name under which the tool X was registered in Agda.

To check that a derivation obtained from createList is correct requires lifting derivation_Δ (which is obtained from the rule system) to a maybe type. This is achieved by the following function:

$$\begin{aligned}&\text{derivation-maybe}_\Delta : \Phi' \to \text{Maybe ProofList}^{\Delta_{ids}}_{\Delta_\Phi} \to \text{Bool}\\&\text{derivation-maybe}_\Delta \; \varphi \; \text{nothing} \;=\; \text{false}\\&\text{derivation-maybe}_\Delta \; \varphi \; (\text{just } l) \;=\; \text{derivation}_\Delta \; ([] \Rightarrow \varphi) \; l\end{aligned}$$

As usual the result is of type Bool as it helps Agda to infer the proofs while type-checking. Note that here the formula φ is explicitly made into a theorem by ensuring that the left-hand of the sequent is empty.

Soundness

Much of the discussion in Section 3.2.1 relating to (Oracle + Justification) is relevant here. In addition, it is emphasised that using Agda to type-check the derivations made by the external tool reduces the number of points in the system that must be trusted when compared with our other method of integrating tools. This is because the other method assumes that the external tools are consistent with the chosen ATP theory, should this not be the case, then Agda would become inconsistent, see Theorem 5.1.1.

This method of using a primitive function to call the SAT solver means that there is no Agda implementation of the function, so whatever result is returned by the primitive function will not yield in an inconsistency, provided the external tool fulfils the functional property. An important caveat to this is that the return type is non-empty, i.e. maybe a derivation. If it was sometimes empty, then it could be possible to derive an inconsistency, see Theorem 5.1.2. The obtained derivations are then checked to be correct by Agda functions. Therefore, no trust is placed in the implementation of createList.

However, it should be noted that absolutely no assurances are made, or even needed about the completeness of this method. It is entirely possible that for some reason (such as a parser error while pre-processing the tools output) a theorem does not produce correct a derivation.

Implementing this interface required adding 179 lines of code to the Agda source code, circa development version 2.3.1.

6.2.2 Efficient Reconstruction

The final step is to reconstruct proof-objects from the derivations efficiently. By efficient, it is meant that it is efficient when compared to the method previously used by (Oracle + Reflection). In this subsection, let Δ and Φ' be as in the previous section (see page 135).

After accounting for the maybe result of a derivation, the reconstruction follows by Theorem 6.1.1. That is

$$\begin{aligned}
&\text{reconstruct-maybe}_\Delta : (\varphi : \Delta_\Phi) \to (l : \text{Maybe ProofList}^{\Delta_{ids}}_{\Delta_\Phi}) \\
&\qquad\qquad \to \top\ (\text{derivation-maybe}_\Delta\ \varphi\ l) \\
&\qquad\qquad\quad \to \forall \xi\ .\ [\![\ \xi \vDash \varphi\ ]\!] \\
&\text{reconstruct-maybe}_\Delta\ \varphi\ \text{nothing}\ p\ =\ \text{efq}\ p \\
&\text{reconstruct-maybe}_\Delta\ \varphi\ (\text{just}\ l)\ \ p\ =\ \text{sound-list}_\Delta\ \varphi\ l\ p
\end{aligned}$$

It is then trivial to assemble all the parts to provide a high-level interface to the reconstruction mechanism as follows:

$$\begin{aligned}
&\text{reconstruct}_\Delta : (\varphi : \Phi') \to \top\ (\text{derivation-maybe}_\Delta\ \varphi\ (\text{createList}\ \varphi)) \\
&\qquad\qquad\qquad \to \forall \xi\ .\ [\![\ \xi \vDash \varphi\ ]\!] \\
&\text{reconstruct}_\Delta\ \varphi\ p\ =\ \text{reconstruct-maybe}_\Delta\ \varphi\ (\text{createList}\ \varphi)\ p
\end{aligned}$$

This definition, when applied to a theorem φ will trigger the external tool to execute, then type-check the resulting derivation. Should the derivation not be correct, then type-checking will fail as there is no element in $\bot$. Should the resulting proof-object be inspected, then it is constructed from the proof

of correctness of the derivation. This reconstruction is linear to the length of the derivation.

In tests, this method of reconstructing the proof-objects from justifications is significantly faster than evaluating the naïve, Agda implementation. This is because many of the choices that the decision procedure would have to take are recorded in the justifications, this results in an essentially linear complexity (depending on the inference rules used) to the number of nodes in the derivation. Results of the SAT reconstruction are presented in Section 6.3.2.

6.2.3 Aside about using Built-Ins

During the implementation of the (Oracle + Justification) interface, a variety of different techniques were explored. One of these techniques—which is worth noting—was to implement in full the derivations as a built-in datatype. This meant that instead of using the Agda parser to interpret the result of executing the external tool, the tool would output (in binary) a representation of a derivation, which Agda would decode into a Haskell datatype. As these data-types are built-in, they can simply be translated into an internal Agda term. The issue here is that the data-structures are created once by the external tool, and then transferred into Agda by a serialisation process, mitigating the need to trigger the parser.

Unsurprisingly using serialisation instead of parsing resulted a better performance, but required an amount of hackery to allow the rule identifiers (from an arbitrary rule system) to be built-in. A limitation of this approach was that the set of formulæ used in the derivation is fixed to be first-order sequents; this is so there is a common language to interface Agda with Haskell (via. the built-in mechanism). First-order formula were chosen because, at present, the majority of automated theorem provers are equivalent to first-order, or a fragment of it. This fixing of the formula resulted in derivations of the form:

$$\mathrm{ProofList}^{\Delta_{ids}}_{[\mathrm{FOF}\Rightarrow\mathrm{FOF}]}$$

So part of the correctness check of the derivation would determine whether the derivation is convertible to a derivation of the form

$$\mathrm{ProofList}^{\Delta_{ids}}_{\Delta_{\Phi}}$$

which the correctness of the rule system is defined over.

Moreover, this method significantly restricts the choice of tool, by requiring it to communicate with Agda in a special binary format. For this reason, it is deprecated in-favour of the presented method that uses primitive functions and the Agda parser. That is flexibility was chosen over efficiency.

6.3 eProver – The Propositional Fragment

The first-order theorem prover, eProver [Sch02], was used for producing propositional justifications of a propositional theorem. eProver is based upon the superposition calculus [BG94, Sch02], although, in this work, only the propositional logic fragment is considered. This tool was selected due to its comprehensive justifications presented clearly in the widely supported TSTP format [Sut09]. Many of the other theorem provers considered (iProver [Kor08], MiniSat [ES03], Vampire [RV02], SPASS [WBH+02], Paradox [CS03], ZCHAFF [MMZ+01]) would require that either the input was transformed into conjunctive normal-form (CNF), or that the output derivation steps for translating the input formula into CNF were omitted. Due to the micro-level proof reconstruction of our technique, these large and complex derivation steps were problematic. Conversely, techniques that perform macro-level proof reconstruction, such as the Sledgehammer sub-system in Isabelle [BN10], do not encounter these problems as the steps are fed into an internal, verified ATP tool, e.g. Sledgehammer uses Metis. The internal ATP reconstructs the proof-object directly as an internal data structure for the ITP tool. Vampire was a close contender to eProver, but due to its lack of an open-source licence (hence source code availability), its use was deprecated.

6.3.1 Rules

It was discovered by examining the output justifications and browsing the source code, that the rules in Figure 6.2 were used by eProver (v1.4). See module `Proof.EProver` in Appendix F for more information. The rules presented have been slightly modified to account for the post-processing of the justifications, specifically this modification allows for fresh variables to be defined in a way that is easy to use for the correctness proof in Agda.

The following functions are used in Figure 6.2:

a[b/c] Syntactical substitution, replace all occurrences of b for c in a. This function is defined in module `Boolean.PL-Formula.Substitute` of Appendix F.

$\neg\varphi$ Negation. It was chosen to remove $\neg$ from the constructors of the Boolean formulæ. This was to simplify equivalence tests between formulæ. Thus, the negation symbol is the map $\varphi \mapsto \varphi \Rightarrow \text{false}$.

mknnf Transforms a formula into negated normal-form, although there are no explicit negations there are absurd implications. The mknnf function pushes these absurd implications to the literals (without building

$$\frac{\Gamma \Rightarrow \varphi}{\Gamma \Rightarrow \text{const-rm } (\text{mknnf } \varphi)}\,\text{fof\_nnf} \qquad \frac{\Gamma \Rightarrow \varphi}{\Gamma \Rightarrow \text{const-rm } \varphi}\,\text{fof\_simplification}$$

$$\frac{\Gamma \Rightarrow \varphi_1 \wedge \varphi_2 \wedge \cdots \wedge \varphi_n}{\Gamma \Rightarrow \varphi_i}\,\underset{i\in\{1\ldots n\}}{\text{split\_conjunct}} \qquad \frac{\Gamma \Rightarrow \varphi}{\Gamma \Rightarrow \text{mkdist } \varphi}\,\text{distribute}$$

$$\frac{\Gamma \Rightarrow \varphi}{\Gamma \Rightarrow \text{const-rm } \varphi}\,\text{cn} \qquad \frac{}{\Gamma \cup \{\varphi\} \Rightarrow \varphi}\,\text{axiom} \qquad \frac{\Gamma \Rightarrow \bot}{\Gamma \setminus \{\neg\varphi\} \Rightarrow \varphi}\,\text{unsat}$$

$$\frac{\Gamma_1 \Rightarrow \varphi_1 \quad \Gamma_2 \Rightarrow \varphi_2}{\Gamma_1 \cup \Gamma_2 \Rightarrow \varphi_1[\neg\varphi_2/\bot]}\,\text{rw}_1 \qquad \frac{\Gamma_1 \Rightarrow \varphi_1 \quad \Gamma_2 \Rightarrow \neg\varphi_2}{\Gamma_1 \cup \Gamma_2 \Rightarrow \varphi_1[\varphi_2/\bot]}\,\text{rw}_2$$

$$\frac{\Gamma_1 \Rightarrow \varphi_1 \quad \Gamma_2 \Rightarrow \varphi_2}{\Gamma_1 \cup \Gamma_2 \Rightarrow \text{const-rm } (\varphi_1[\neg\varphi_2/\bot])}\,\text{sr}_1 \qquad \frac{\Gamma_1 \Rightarrow \varphi_1 \quad \Gamma_2 \Rightarrow \neg\varphi_2}{\Gamma_1 \cup \Gamma_2 \Rightarrow \text{const-rm } (\varphi_1[\varphi_2/\bot])}\,\text{sr}_2$$

$$\frac{\Gamma_1 \Rightarrow \varphi_1 \quad \Gamma_2 \Rightarrow \varphi_2 a \leftrightarrow \varphi_2 b}{\Gamma_1 \cup \Gamma_2 \Rightarrow \varphi_1[\varphi_2 b/\varphi_2 a]}\,\text{apply\_def} \qquad \frac{\Gamma \cup \{x_i \leftrightarrow \psi\} \Rightarrow \varphi}{\Gamma \Rightarrow \varphi}\,\text{fresh}$$

Figure 6.2: eProver derivations, note that unsat is the classical reductio ad absurdum law. The rw and sr rules form standard propositional refutation. The fresh rule introduces a new variable x_i and has the side conditions $x_i \notin FV(\varphi \wedge \psi) \cup FV(\Gamma)$.

chains of absurd implications), and pre-existing implications are replaced by material implications. A side-effect of this procedure is that a number of constant values are introduced into the formula which must be removed; they are removed by const-rm. The translation is achieved by defining two mutually recursive, symmetric functions that keep track of whether the formula has been negated. See module `Proof.EProver.NNF` in Appendix F for the complete definitions.

$$\begin{aligned}
&\text{mknnf} : \text{BooleanFormula} \to \text{BooleanFormula}\\
&\text{mknnf } (\varphi \wedge \psi) &&= \text{mknnf } \varphi \wedge \text{mknnf } \psi\\
&\text{mknnf } (\varphi \vee \psi) &&= \text{mknnf } \varphi \vee \text{mknnf } \psi\\
&\text{mknnf } (\varphi \Rightarrow \psi) &&= \neg\text{mknnf } \varphi \vee \text{mknnf } \psi\\
&\text{mknnf } \varphi &&= \varphi
\end{aligned}$$

and

$$
\begin{array}{lcl}
\neg\text{mknnf} : \text{BooleanFormula} \to \text{BooleanFormula} & & \\
\neg\text{mknnf}\ (\varphi \wedge \psi) & = & \neg\text{mknnf}\ \varphi \vee \neg\text{mknnf}\ \psi \\
\neg\text{mknnf}\ (\varphi \vee \psi) & = & \neg\text{mknnf}\ \varphi \wedge \neg\text{mknnf}\ \psi \\
\neg\text{mknnf}\ (\varphi \Rightarrow \psi) & = & \text{mknnf}\ \varphi \wedge \neg\text{mknnf}\ \psi \\
\neg\text{mknnf}\ (\text{var}\ x) & = & \neg(\text{var}\ x) \\
\neg\text{mknnf}\ \text{true} & = & \text{false} \\
\neg\text{mknnf}\ \text{false} & = & \text{true}
\end{array}
$$

const-rm Removes the constants (true and false) from a formula. Removing constants from a propositional formula, amounts to evaluating/simplifying the formula, in some cases the whole formula will collapse down into a constant and will not be able to be simplified any further. This is achieved by starting at the root of the formula and replacing each connective node ($\wedge, \vee, \Rightarrow$) by a function that will attempt to evaluate that node, after its operands have been evaluated. If evaluation succeeds, then these functions will return a Boolean value, and otherwise they will reconstruct the original node. For example, in the case of the $\vee$ node, it is replaced by the function:

$$
\begin{array}{lcl}
\text{map-or} : \text{BooleanFormula} \to \text{BooleanFormula} \to \text{BooleanFormula} & & \\
\text{map-or}\ \text{true}\ \psi & = & \text{true} \\
\text{map-or}\ \text{false}\ \psi & = & \psi \\
\text{map-or}\ \varphi\ \text{true} & = & \text{true} \\
\text{map-or}\ \varphi\ \text{false} & = & \varphi \\
\text{map-or}\ \varphi\ \psi & = & \varphi \vee \psi
\end{array}
$$

The remaining two cases are similar, and are formalised in module `Boolean.PL-Formula.RemoveConstants` of Appendix F.

mkdist Constructs a CNF formula by distributing the disjunctions over conjunctions. It assumes that the formula is in negated normal-form (i.e. implication only occurs as absurd implication on the literals), notably it treats implications as literals—because the only implications that remain in a negated normal-form formula represent negations of the literals. It is implemented by the three functions below, and the full code is in module `Boolean.PL-Formula.Distribute` of Appendix F. Note that the underlined types are only for intuitive understanding, and they are synonymous with BooleanFormula to mitigate having to translate the result back into a BooleanFormula. We note that if Agda

supported sub-types/refinement types, then their types could have been given explicitly.

$$
\begin{aligned}
&\text{dist-clause} : \underline{\text{Clause}} \to \underline{\text{CNF}} \to \underline{\text{CNF}} \\
&\text{dist-clause } cl\ (\varphi \wedge \psi) \;=\; (\text{dist-clause } cl\ \varphi) \wedge (\text{dist-clause } cl\ \psi) \\
&\text{dist-clause } cl\ \varphi \qquad\quad =\; cl \vee \varphi \\
&\text{dist} : \underline{\text{CNF}} \to \underline{\text{CNF}} \to \underline{\text{CNF}} \\
&\text{dist}\ (\varphi_1 \wedge \varphi_2)\ \psi \;=\; (\text{dist}\ \varphi_1\ \psi) \wedge (\text{dist}\ \varphi_2\ \psi) \\
&\text{dist}\ \varphi \qquad\quad \psi \;=\; \text{dist-clause}\ \varphi\ \psi \\
&\text{mkdist} : \text{BooleanFormula} \to \underline{\text{CNF}} \\
&\text{mkdist}\ (\varphi \vee \psi) \;=\; \text{dist}\ (\text{mkdist}\ \varphi)\ (\text{mkdist}\ \psi) \\
&\text{mkdist}\ (\varphi \wedge \psi) \;=\; (\text{mkdist}\ \varphi) \wedge (\text{mkdist}\ \psi) \\
&\text{mkdist}\ \varphi \qquad\quad =\; \varphi
\end{aligned}
$$

Remark

The negated and conjunctive normal-form functions do not construct true normal-forms. In the case of negated normal-form it leaves *junk* in the formula that must be removed by const-rm, and conjunctive normal-form does not consider the associativity of conjunction and disjunction. The latter issue does not matter as in the following the formulæ are flattened.

Checking that the derivations obtained from eProver are definitionally correct with respect to Figure 6.2 was troublesome to achieve. In part, this is because eProver was not internally verified by Agda, so the implementation of the rules provided in Agda produced syntactically different (but semantically equivalent) formulæ. For example a sequence of nested conjuncts/disjuncts could change its order, thus definitional equality in Agda would fail to hold. Although a lot of work was put into attempting to achieve definitionally equivalent results from eProver, it was decided to sacrifice some efficiency by defining an equivalence between propositional formulæ.

The equivalence works by first flattening the syntax of the formulæ. The flattening (greedily) transforms formulæ built over binary operators into formulæ built over semantically equivalent n-ary operators. The flattened formula are defined in module `Boolean.PL-Formula.Equivalence` of Ap-

pendix F as follows:

$$
\begin{array}{l}
\text{data Flattened} : \text{Set where} \\
\quad \text{true false} \;:\; \text{Flattened} \\
\quad \wedge\_\ \vee\_ \;:\; \text{List}' \text{ Flattened} \to \text{Flattened} \\
\quad \_\Rightarrow\_ \;:\; \text{Flattened} \to \text{Flattened} \\
\quad \text{var} \;:\; \mathbb{N} \to \text{Flattened}
\end{array}
$$

Here, List′ is a non-empty list, i.e. the nil constructor [] is replaced by $[\_] : A \to \text{List}'\ A$.

The equivalence relation is defined between two flattened formulæ, intuitively it means that the two formulæ have the same semantical structure. The equivalence on flattened formulæ is defined as follows:

$$
\begin{array}{lll}
\text{equiv} : \text{Flattened} \to \text{Flattened} \to \text{Bool} & & \\
\text{equiv}\ (\wedge\ \Gamma) & (\vee\ \Delta) & = (\text{all-equiv}\ \Delta\ (\wedge\ \Gamma)) \vee (\text{all-equiv}\ \Gamma\ (\vee\ \Delta)) \\
\text{equiv}\ (\vee\ \Gamma) & (\wedge\ \Delta) & = (\text{all-equiv}\ \Delta\ (\vee\ \Gamma)) \vee (\text{all-equiv}\ \Gamma\ (\wedge\ \Delta)) \\
\text{equiv}\ (\vee\ \Gamma) & (\vee\ \Delta) & = (\text{equiv-operands}\ \Gamma\ \Delta) \\
 & & \quad \vee (\text{all-equiv}\ \Gamma\ (\vee\ \Delta)) \vee (\text{all-equiv}\ \Delta\ (\vee\ \Gamma)) \\
\text{equiv}\ (\vee\ \Gamma) & \psi & = \text{all-equiv}\ \Gamma\ \psi \\
\text{equiv}\ (\wedge\ \Gamma) & (\wedge\ \Delta) & = (\text{equiv-operands}\ \Gamma\ \Delta) \\
 & & \quad \vee (\text{all-equiv}\ \Gamma\ (\wedge\ \Delta)) \vee (\text{all-equiv}\ \Delta\ (\wedge\ \Gamma)) \\
\text{equiv}\ (\wedge\ \Gamma) & \psi & = \text{all-equiv}\ \Gamma\ \psi \\
\text{equiv}\ \varphi & (\vee\ \Delta) & = \text{all-equiv}\ \Gamma\ \varphi \\
\text{equiv}\ \varphi & (\wedge\ \Delta) & = \text{all-equiv}\ \Gamma\ \varphi \\
\text{equiv}\ (\varphi_1 \Rightarrow \psi_1) & (\varphi_2 \Rightarrow \psi_2) & = (\text{equiv}\ \varphi_1\ \varphi_2) \wedge (\text{equiv}\ \psi_1\ \psi_2) \\
\text{equiv}\ \varphi & \psi & = \varphi \equiv \psi
\end{array}
$$

where

$$\text{all-equiv}\ [\varphi_1, \varphi_2, \ldots, \varphi_n]\ \psi = \bigwedge_{i=1}^{n} \text{equiv}\ \varphi_i\ \psi$$

and

$$\text{equiv-operands}\ [\varphi_1, \varphi_2, \ldots, \varphi_n]\ [\psi_1, \psi_2, \ldots, \psi_m] = (\bigwedge_{i=1}^{n}(\bigvee_{j=1}^{m} \text{equiv}\ \varphi_i\ \psi_j)) \wedge (\bigwedge_{j=1}^{m}(\bigvee_{i=1}^{n} \text{equiv}\ \psi_j\ \varphi_i))$$

This relation was proved to be an equivalence, i.e. reflexive, symmetric and transitive. Deciding the relation results in an exponential algorithm. Assume two flattened formula φ and ψ, an upper bound on the complexity of equiv φ ψ is computed as follows. Let n be the greatest width of φ and

ψ, and b_k be the upper bound on the complexity for a formula of height at most k, such that $b_0 = 1$, then

$$\begin{aligned} b_k &= 2n^2 b_{k-1} + 2n b_{k-1} \\ &= 2n(n b_{k-1} + b_{k-1}) \\ &= 2n(n+1) b_{k-1} \\ &= (2n(n+1))^k \end{aligned}$$

Therefore, let h be the maximum height of φ and ψ, the complexity is given as $(2n^2)^h$.

Figure 6.3 presents an example of the flattening and the equivalence on the flattened formulæ.

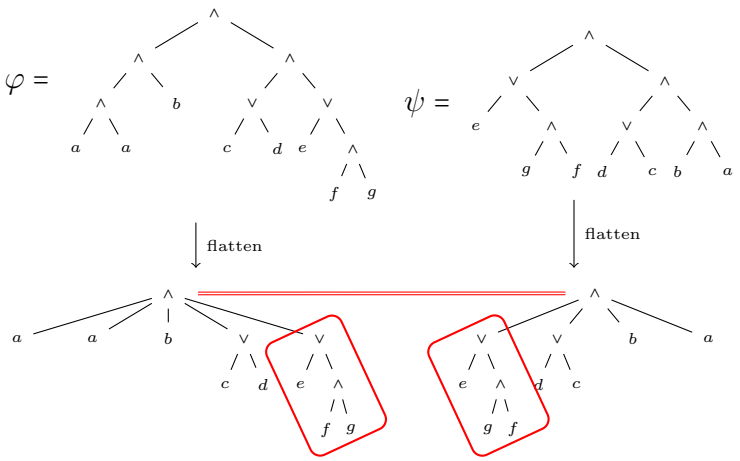


Figure 6.3: Determining equivalence between two formulæ φ and ψ, is done by collapsing the binary propositional operators into semantically equivalent n-ary operators. Then checking that the roots of the flattened formulæ are equal, and their operands are equivalent. This equivalence allows for reordering and duplication of the operands. After φ and ψ are flattened, the roots ($\wedge$) are checked to be definitionally equal and that each of the operands that one structure has, so does the other. Consider the boxed sub-formulæ, these are equivalent operands of the roots.

Implementing proof reconstruction for eProver required 1,607 lines of Agda code. The majority of this related to proving soundness of the rules. A third of the code (566 lines) is for the equivalence relation.

Wrapper Program

A small wrapper program was required to pre-process the output of eProver. The output is formatted using TSTP [Sut09]. The wrapper program then translates it into a compatibly formatted derivation (i.e. ProofList). The principal feature provided by this pre-processing is that the assumptions used by each rule are made explicit into a context; this saves the Agda program that checks whether the derivation is correct, from performing a number of look-ups in a (potentially) large list. As a general heuristic, it is faster to perform such operations outside Agda. The process of the wrapper script is given by:

- Parse the output,
- Construct a ProofList
- Pretty print the ProofList in Agda format.

Each of these steps is considered in more detail below.

Parsing. The wrapper made use of the logic-TPTP parser, available from Hackage. However, due to the diverse nature of TSTP files the program needs fine-tuning for different ATP tools. This is mainly due to there not being a fixed set of rules that are used to represent the derivations, and that different tools annotate the rules in different ways.

During the early phase of the project, it was the intention for Agda to support the full parse tree of logic-TPTP, but it soon became clear that this level of detail was not necessary, and it was not clear whether processing this low-level representation in Agda would be efficient.

For example, when the wrapper program looks-up a rule application, it is required to search a list, resulting in a large number of equality tests. To perform this same operation in Agda would be significantly slower; also performing these operations outside of Agda allows for the use of library code. For example, the wrapper program uses the Haskell Data.Generics library to extract parts of the parse tree without having to perform full case-distinctions on it, which would have to be done in Agda.

Construction. The output of eProver, once parsed from TSTP format is given as a list. The head of the list is the first derivation, and the last node in the list is the conclusion of the derivations. The list has the type

$$\text{List}\ (\mathbb{N} \times \text{BooleanFormula} \times \text{Rule} \times \text{List}\ \mathbb{N})$$

where the first natural number is the identifier of the derivation, the formula is the conclusion of the derivation, Rule is the derivation identifier, and the list of numbers are references (pointers in the list) to the premises. It is noted that this list typically forms a partial map, where the identifiers form the domain. This is because eProver will directly omit steps of the derivation which are not useful for proof reconstruction.

To recall, a proof list essentially has the type:

$$\text{List}\ (\text{List}\ \text{BooleanFormula} \times \text{BooleanFormula} \times \text{Rule} \times \text{List}\ \mathbb{N})$$

Instead of each rule explicitly stating its identifier, its position in the list determines its identifier. Also, a list of formulæ are added that forms the context, i.e. contains the assumptions. Finally, the list of premises are adjusted accordingly as the identifiers of the derivations have changed.

This is translated by induction on the list, building up a list of proof nodes, and a mapping between the old identifiers and the new identifiers. When an axiom is found, it is added to the context. In general, the contexts are constructed according to Figure 6.2. At the end of the derivation, it might not be the case that the context is empty because free variables were introduced during the derivation using the TSTP "introduced" annotation. These annotations are replaced by the "axiom" (assumption) rule, which are then discharged at the end of the derivation by applying the "fresh" rule. See Figure 6.2 for the formalisation of "fresh", and module `Boolean.PL-Formula.DropEquivalence` in Appendix F for the technical details.

There are also a number of smaller translations undertaken, such as removing superfluous derivations (e.g. TSTP answers), and unfolding compound applications of the same derivation, as the proof nodes do not support compound applications. Negation is replaced by absurd implications. The introduced variables previously mentioned are identified by strings and not natural numbers, so there is also an added step of scanning the derivation, finding how many variables it contains, then mapping these introduced variables to fresh, natural number indices, as required by the constraints of the "fresh" rule.

For more information regarding the use of this wrapper program, example inputs and outputs, see Appendix E.

Printing. The output is printed direct to the standard output stream (stdout). Printing is a fairly trivial process; a Haskell function traverses the produced derivation, and prints it in the same ASCII format that can be fed directly into Agda, and type-checked.

6.3.2 Evaluation

The eProver proof reconstruction presented has been used to explore various problem sets, especially the excluded-middle and pigeonhole principle problem sets, see Figure 6.4 and Figure 6.7, respectively for comparative plots to those in Section 5.4.1. The propositional formulæ, functions used to explore the proof-objects and the meaning of Total, GC, Mutator, and Tool are the same as defined in Section 5.4.1. The difference in these plots is that instead of obtaining proof-objects by Theorem 4.1.2, they are instead obtained by the external tool and Theorem 6.1.1.

The first plot (Figure 6.4) is of the excluded-middle problem set. The plot shows that the number of variables the formula can containe while being feasible to type-check, is greater than 10 fold than using the (Oracle + Reflection) approach, cf. Figure 5.3. This is because eProver was able to elucidate the hidden structure of the problem. Hence the obtained derivations were of a constant number of steps, whereas in (Oracle + Reflection) the proof-objects were reconstructed using the soundness proof of the naïve tautology checker, which is bounded by the number of variables in the formula. There are two steps to the process: first is to obtain and type-check a derivation, and the second is to explore the resulting proof-object. In Figure 6.4 the composition of both steps is plotted, and in Figure 6.5 only the first step is plotted. The first plot had an observed run-time complexity of $n^{2.25}$, whereas the latter plot's complexity (which includes deciding the equivalence relation) is $n^{1.25}$. It is not entirely clear why there is such a large difference (see below), in part this is due to a lack of sharing in Agda and over normalisation. The construction of the proof-object by Theorem 6.1.1 performs induction on the derivation list and applies the soundness proof for each rule, culminating in a proof of the theorem. This proof has the form:

$$\forall \xi \,.\, [\![\, x_{n-1} \vee \cdots \vee x_1 \vee x_0 \vee \neg(x_{n-1} \vee \cdots \vee x_1 \vee x_0) \,]\!]_\xi$$

This is applied to the environment

$$\{0 \mapsto \mathrm{true}\,;\, \_ \mapsto \mathrm{false}\}$$

to obtain a proof-object of the form:

$$\underbrace{\mathrm{inj}_2\ (\mathrm{inj}_2\ (\cdots}_{n-1\ \text{times}}(\mathrm{inj}_1\ tt)\cdots))$$

This object is obviously linear to explore.

The second problem set is the pigeonhole principle. It has a worse performance than using no external tools at all, compare Figure 5.6 with Figure 6.7.

Note that for $n = 3$ in Figure 5.6 the time of the mutator is 15 seconds, where in Figure 6.7 it is 369 seconds. The plot in Figure 6.6 is the time required to type-check that a derivation for the pigeonhole principle is correct. When using no external tools, Agda could explore a larger instance, namely when $n = 4$, than was possible when reconstructing the proof-object. This is because the complexity of the derivations grows faster than exponential (see number of steps in the table below) as the problem set grows (i.e. as n grows larger). Instead, when using no external tools the naïve decision procedure is exponential on the number of variables (brute force), so the complexity of the pigeonhole principle is not relevant. For instance, the derivations generated by the wrapper program and piped to Agda had the sizes, and numbers of steps:

	$n = 1$	$n = 2$	$n = 3$	$n = 4$
Size	14KB	73KB	464KB	3.5MB
Number of steps	30	81	279	1257

To put these numbers into perspective, at the time of writing, all the code in the standard library is 948KB; whereas the sizes above relate to a single Agda term (ProofList). These inputs are then parsed by Agda, and type-checked. It is noted that type-checking the derivation for $n = 4$ was not possible, the type-checker was left running for 20 hours after which time it started to use swap space, so it was terminated. From this table and Figure 6.7, it is clear that as the derivations grow in complexity (number of nodes), so to does the type-checking.

It was also tried on an industrial problem set (cf. Section 10.5), but the testbed had insufficient resources (RAM) to type-check any derivations. For instance, the initial case resulted in a derivation that was approximately 20MB, Agda quickly consumed all the systems resources and was then terminated. The inductive case produced a derivation that was approximately 118MB, which is far beyond the current capabilities Agda. It should be noted that the output from eProver for the same problem was approximately 80KB. One reason for this blow-up is that the final rules to be applied to the derivation were “raa” and “fresh”, both of these add information to the context, which is propagated throughout the derivation (contexts). This is by design as it helped to reduce the amount of work done in Agda (i.e. not having to look-up axioms), however, for large problem sets it appears to have a detrimental effect, although it only creates a blow-up linear to the length of the derivation.

6.4 Remarks

Enabling proof reconstruction using this method allows for in most cases larger and more complicated proof-objects to be created and explored than was possible before, provided that the derivation for the theorem to be proved is not too large. Therefore, there is a trade-off between this approach, and the previous approach (Oracle + Reflection) with respect to the number of variables and number of nodes in the derivation.

However, there are a number of issues that have become apparent with the implementation. First, the equivalence relation defined on flattened formulæ is exponential. This relation was required to simplify the process of integrating eProver, i.e. not requiring definitionally equivalent results. However, it should be possible to formalise in Agda versions of mknnf and mkdist that coincide with the versions defined inside eProver, possibly by copying their definitions from the eProver source code. This would mean that Agda and eProver would give definitionally equivalent results, hence the equivalence relation would not be required. Alternatively, it might be possible to pre-process the output into a normal-form, particularly with an ordering on the variables before it is passed to Agda, this would reduce the workload of deciding whether two formulæ are equivalent in Agda.

A second issue is that assumptions are explicitly carried around inside the derivation. This was to simplify the correctness check inside Agda. However, it should be possible to replace axioms (assumptions) by numerals, similar to what is done in natural deduction. For example, the style of natural deduction pioneered by Supps in [Sup99] would be particularly suited.

We remark that while integrating eProver with Agda, it became apparent that eProver contained a bug in the outputting of the justifications. The bug did not affect the soundness of eProver, but it was enough to prevent Agda from being able to type-check the derivations. Specifically when introducing a new variable, instead of using a provable equivalence to define the value of the introduced variable, only an implication in the wrong direction was output. Essentially the following rule was used

$$\frac{p \quad \varphi \to p}{\varphi}$$

where $p \notin FV(\varphi)$. This resulted in breaking the proof reconstruction in Agda, the error was brought to the attention of Stephan Schulz (developer of eProver) who quickly fixed it. This shows the advantage of performing micro-step proof reconstruction (i.e. each proof step is used directly), because eProver has been used for macro-step proof reconstruction in a number of other interactive theorem provers without the bug being detected.

Macro-step reconstruction is where the proof steps are fed into a second (verified) theorem prover that attempts to make sense of them, and then translates them into a proof-object. For an example of macro-step reconstruction see [Hur05], where the automated theorem prover Metis is used to interface between tools such as eProver and Isabelle.

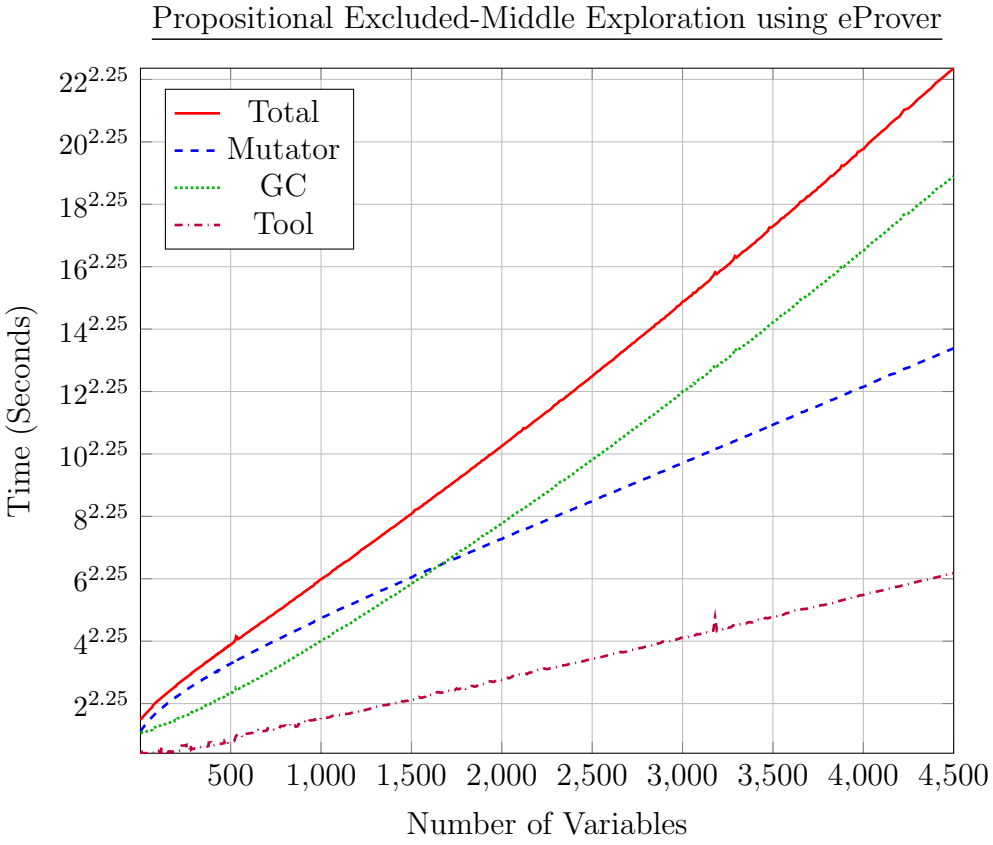


Figure 6.4: This plot shows the reconstruction of instances of the excluded middle problem set. The largest value for total time is 1,087.45 seconds. The tests were not limited by space, and it would have been possible to continue to larger magnitudes; however, this would not have been enlightening as the observed complexity is clearly visible above. 671 samples were taken at varying distances, at the beginning, samples were taken at a spacing of 10, towards the end, samples were taken at a spacing of 50.

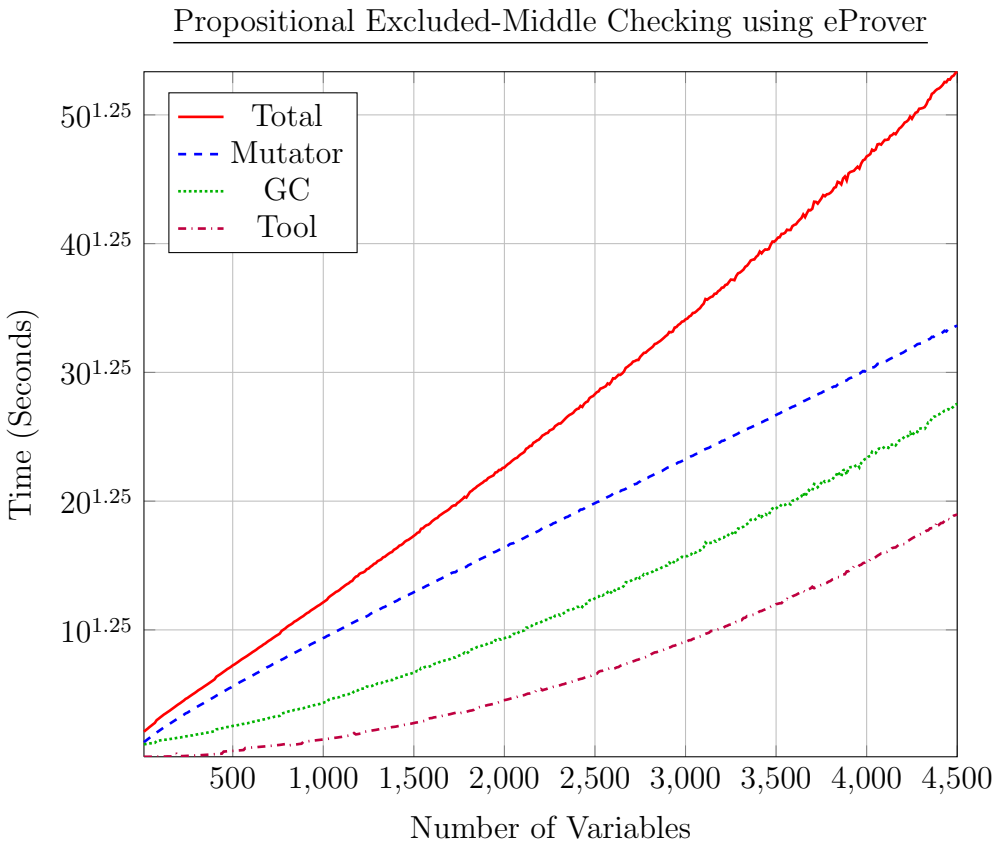


Figure 6.5: This plot shows only the stats for checking whether an instance of the excluded middle problem produced a correct derivation. The greatest time is 144.2 seconds. 449 samples were taken at a spacing of 10.

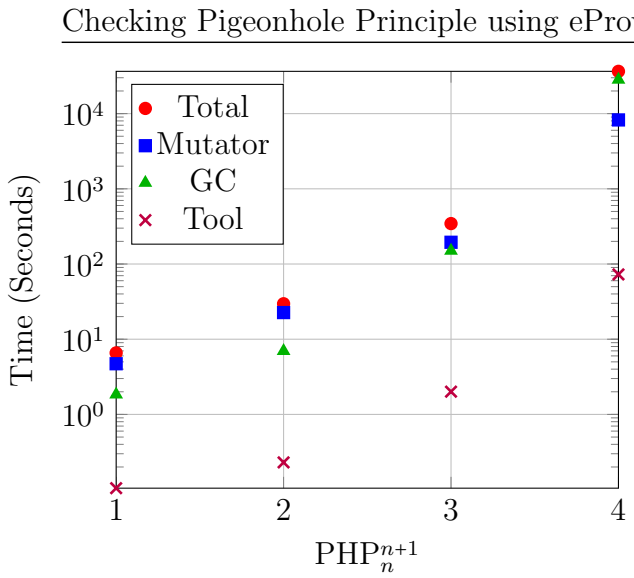


Figure 6.6: This plot shows the time required to check that an obtained derivation for an instance of the pigeonhole principle was correct. The value for the largest total time is 36,489.55 seconds, it was not possible to run larger tests due to a lack of resources.

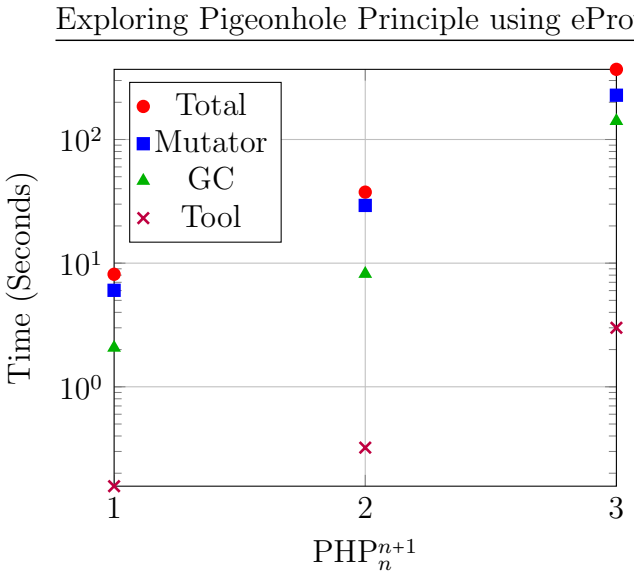


Figure 6.7: This plot shows the reconstruction of instances of the pigeonhole problem. The value for the largest total time is 369.3 seconds, compared to 344.9 seconds above for the same size, it was not possible due to lack of resources to run larger tests.

Chapter 7

Summary

In this part, various methods of integrating Agda with external tools have been explored. Specifically two techniques have been explored. First, a new approach of combining an oracle with reflection was explored (Oracle + Reflection). Using this technique, significantly larger problem sets were explored than was possible without the technique. The technique entailed defining a decision procedure in Agda for a logic and proving that is correct, then overriding the implementation of the decision procedure for closed terms by a call to an external tool. Two different techniques were applied to override the implementation of the decision procedure. The first technique required that the Agda source code was modified for each new decision procedure, such that it would only execute the external tool if the decision procedure (in Agda) fulfilled a number of axioms. The second technique traded soundness for usability by not requiring modifications to Agda sources for each different decision procedure. However, it is possible for the standard user to derive inconsistencies by specifying the wrong external tool to execute.

Using this approach the decision procedures for SAT solving, CTL model-checking, and symbolic model-checking were overridden by a call to an external tool. A number of plots were presented that clearly showed that connecting Agda to external tools using this approach allowed significantly larger problem sets to be verified, and the resulting proof-object to be explored.

Subsequently a second approach of integrating Agda with external tools was implemented, this was the (Oracle + Justification) technique in Chapter 6. In this approach, the decision procedure for the logic is not defined in Agda. Instead, the logic that the external theorem prover operates within is formalised, notably the inference rules are formalised. Then a primitive function is provided that will execute the external tool, and obtain a derivation from it. These derivations are translated into Agda terms and type-checked, thus reducing the kernel of trust to the Agda type-checker itself. This technique was demonstrated by implementing the propositional frag-

ment of eProver's paramodulation calculus.

The use of eProver caused a number of problems with trying to obtain definitionally equal formulæ. This resulted in having to design an equivalence between propositional formula; however, this equivalence requires exponential time. The plots show that when the proofs are not complicated, such as the excluded-middle problem sets, this approach is more efficient than our previous approach. However, as the complexities of the derivation grow the former method is more efficient.

7.1 Future Work

As future work, the oracle and reflection approach could be extended into a plug-in mechanism. These plug-ins would be defined in an *almost* Agda file format. This format would specify for each built-in data-type, its inductive elimination principle and corresponding Haskell data-types. It would also specify for each function definition a number of axioms (which are given by Agda formulæ), and a Haskell implementation of the function. Then during type-checking when a built-in declaration is encountered, the corresponding plug-in file is also type-checked. This type-checking would ensure that if it is a data-type that is to be built-in, then it fulfils its inductive elimination principle; or if it is a function to be built-in, then it is of the correct type and fulfils the required axioms. Of course, these plug-ins would be dependent upon each other, e.g. decision procedures are dependent upon the definition of Booleans. The fundamental issue here is that these plug-in files should not be something that the standard Agda user touches, thus plug-ins are trusted to be implemented correctly. One possibility is that the plug-ins are required during Agda's compilation, the intention being that the standard user does not compile Agda themselves. Implementing such a system would yield a high-level of soundness assurances, provided the implementations of the plug-ins are correct.

It has been identified that the proof reconstruction interface is limited by two factors, firstly an equivalence relation that requires exponential time to decide, and secondly, by the explicit representation of the assumptions. As future work, it would be beneficial to represent assumptions in the derivation by numerals, rather than explicitly, i.e. how it is normally done for natural deduction. This would require that the formulation of a rule system and the correctness check would need to be amended to reference previous points in the derivation. The benefit of performing this transformation would be that less space is required to represent the derivations and that when checking they are correct, the checks on the contexts would be vastly simplified. That

is instead of checking whether two contexts are equivalent by checking they contain the same formulæ (possibly in different orders), one would only have to check they contain the same numerals.

The exponential equivalence relation should be removed in-favour of either computing a normal-form outside of Agda for the formulæ in the derivation during pre-processing, or perhaps more interestingly by internally verifying eProver. By internally verifying eProver, it is meant that the algorithms used by eProver should be formalised in Agda, and proved correct. Then instead of using approximations of these functions, as is currently the case, requiring an equivalence, they could be directly computed, resulting in definitionally equivalent formulæ. While internally verifying eProver, it might be the case that bugs are identified in its internals.

7.2 Automated Provers Explored

Throughout the project, a selection of different automated theorem provers were explored. Although this project is concerned with formulating a framework in Agda for developing verified control systems, the ATP's that were focused on were selected not only for speed but also by usability. For instance there are a number of acclaimed SAT solvers, such as zChaff [MMZ+01], that only take as input conjunctive normal-form (CNF) formula. These tools would then have required an external translation into CNF, further increasing the amount of trust required. The tools are described below.

Z3 v3.2 [dMB08] Possibly the best theorem prover used in this work. It is fast, versatile and has a user-friendly input/output. Z3 supports the use of many different file formats, one of which is Thousand Problems for Theorem Provers (TPTP), the format used in this work. However, it was never discovered how to get proof reconstruction working for it. It appears that Z3 supports dumping the unsatisfiable core of a theorem, but after a number of attempts it was not identified how to dump the core for propositional problems. The documentation only refers to the core dump under SMT theorem proving. A second limitation is that Z3 is free for research purposes, but not for industrial applications.

incremental-sat-solver v0.1.7 [Fis12] During the early phases of the project, this SAT solver was integrated into Agda. It was selected because it is freely available in Hackage, the same package manager used by Agda, and thus it could be install automatically with Agda (via a dependency). As the SAT solver was imported into the Haskell program as a function of the type Formula → Bool, it did not require

being wrapped up in an IO monad, thus simplifying the connection. In fact, the built-in decision procedure in Agda was bound to the actual imported library function. It was soon discovered that incremental-sat-solver had efficiency issues (time and space) for our industrial test cases, so its use was deprecated.

Paradox / minisat [CS03, ES03] The Paradox and Equinox tool set was explored after the incremental-sat-solver proved itself inefficient. These tools are also written in Haskell and make use of minisat. Although fast, and interfaced using TPTP, occasionally Paradox was unstable. This instability was because of the use of Haskell's foreign function interface. Paradox was executing minisat (C code) library functions that would conflict with GHC's run-time memory allocations, and this was manifested by an unpredictable segmentation fault.

iProver v0.8.1 / eProver [Kor08] Subsequently the iProver tool was used. It interfaces using TPTP. Internally iProver operates on CNF formulæ, and the input is translated into CNF using eProver's clauseifier. The use of iProver was largely successful. It was not the most efficient tool, but it worked; however when considering proof reconstruction, iProver was unable to provide justifications for its proofs.

eProver v1.4 [Sch02] Following on from iProver, eProver was explored. It is a nice tool-set that could interface using TPTP/TSTP formats and provides many features. Most importantly, with regards to this thesis was a tool called eproof that would extract from an execution of eProver a sequence of justifications for an unsatisfiable theorem. During this project, a bug was identified in how one of these justifications was presented (circa v1.2). After notifying the developer, the bug was rapidly fixed. The bug related to an implication in the wrong direction after a fresh variable was assigned a value. The eProver tool was used for both the (Oracle + Reflection) and (Oracle + Justification) approaches. However, Z3 performs faster for the (Oracle + Reflection) SAT interface (cf. Section 5.4.1 and Section 6.3.2).

SPASS and Vampire [WBH+02, RV02] Although these well-known and respected tools were considered for proof reconstruction, they were rejected. SPASS was rejected as it did not support outputting in TSTP format; instead it used its own format. Apparently a script does exist to translate into TSTP format, but the author was not able to locate it after much searching.

The Vampire prover was rejected for being closed-source. This is because when performing proof reconstruction, in essence an approximation of the tools internals is verified. If access to these internals is not possible, then it is harder to define (and prove) the reconstruction in Agda when compared to eProver which is open-source.

NuSMV v2.5.2 [CCG+02] The model-checker NuSMV is used for the CTL interface. However, it was never able to operate on the industrial test case. Various command line options were tried, but either it would consume all available memory then start swapping or would take a lot of time, and then terminate with a segmentation fault. It appears as though (at least at the time of writing) NuSMV is inherently unstable because there have been numerous discussions on the nusmv-users mailing list about segmentation faults/crashes on a variety of operating systems. It was surprising that NuSMV was unable to verify the test case as it is presented by its creators as an industrial model-checker.

Apart from the automated theorem provers above, the Haskell package `logic-TPTP-0.2.0.7` was used. This package provides a parser that was used to interpret the results of the provers and construct a parse-tree. The package was particularly useful.

Finally, a number of different automated theorem provers are available on-line at `http://www.tptp.org` . These provers are executed on the server and take TPTP format files. This allows for quickly testing different theorem provers on a problem set, it was used on a number of occasions during the project to identify which tools would be suitable for proof reconstruction.

Remark

As future work, it would be appealing to connect Agda to the on-line tools, and allow the user to select which tool they would like to execute. This is the same behaviour as the use of Sledgehammer in Isabelle.

Part II

Railways

Excerpt from Armagh Rail Disaster Report: The effects of the collision were most disastrous, the rear three vehicles of the excursion train being completely destroyed, their débris being thrown principally to the right of the direction in which they had been running, down the slope of an embankment (about $46\frac{1}{2}$ high) on which the railway is here carried. On the collision occurring, the engine of the ordinary train—which consisted of engine and tender, horse box, brake van, three carriages, and third class brake van, fitted with the non-automatic vacuum brake—was separated from its tender and thrown over to the right upon its right side (left wheels uppermost) on to the top of the slope of the embankment. The rear five vehicles broke away from the horse box, ran back down the incline, and were stopped by the guard in the rear brake van applying his hand brake after they had run back about a quarter of a mile. The tender and horse box also ran back and were stopped by the driver—who, upon the crash occurring had turned round and was holding on by the tender coal plate—applying the tender hand brake (which had remained in working order) a few carriage lengths short of the rear five vehicles.

– Major Gen. C.S. Hutchinson
Assistant Secretary
Board of Trade (Railway Dept.)
1889

Chapter 8

Railway Specification

One goal of this work is to use Agda to aid in the development of verified railway critical systems. The remainder of this thesis will focus on the railway domain, and using Agda to verify properties in this domain.

Grand Challenge Dines Bjørner proposed a grand challenge of computer science is to develop formally, and verify railway systems [Bjø04, Hoa03]. This is not limited to critical systems. Part of the challenge includes the specification of these systems. The railway domain was proposed to be a grand challenge due to the number of different systems operating with different types of data, see the bullet points below. Over recent years, there have been many successful attempts to specify different parts of the railway domain, some of which are presented in this chapter. The parts of the grand challenge relevant to this research include:

- The topology of the network, how all the train tracks and signals are connected.
- The operational behaviour of the topology, when a signal can show a proceed aspect.
- Interlocking systems[1] that control the topology.

The grand challenge also encompasses other issues (not considered here) such as:

- Fare management i.e. consistency between stations,
- Train timetables, and
- Staff rostering.

[1] An interlocking system is a critical system that is responsible for safety on the railway.

Chapter Overview. This chapter discusses this project's contribution to modelling the topological (Section 8.1) and operational aspects (Section 8.2) of railways. The models presented here have been used to model railyards in Agda (and form the basis of the verification), thus they are minimal with respect to the information they contain. It should be an easy matter to extend them to facilitate the verification of a wider range of properties.

From Section 8.3 onwards a framework is identified that facilitates proving theorems in the railway domain, namely that the domain safety requirements follow by a selection of signalling principles (lemmata). In this case, domain safety is that *trains do not collide or derail*. This framework facilitates exploring which signalling principles are required to guarantee domain safety. The signalling principles, domain safety, and models should be validated by domain experts. The advantage of performing this meta-verification is that it elucidates the required verification conditions that a given control system must satisfy in-order to guarantee domain safety. Although these sections are specifically concerned with proving theorems in railway domain, the approach can be applied to other domains, in such a situation, the signalling principles are synonymous with safety principles.

Specifically the sections from Section 8.3 onwards are as follows: in Section 8.3 an abstract model of the railway domain is developed. In Section 8.4 signalling principles and domain safety are introduced. Then in Section 8.4.1 using this abstract model a number of signalling principles are shown to imply domain safety.

Modelling and verifying the implementation of a train control system with respect to the railway domain models presented here is deferred until Chapter 9 and Chapter 10. There is a case study of the verification in Chapter 11.

8.1 Physical Layout

Initially, when designing a railway, it is required to fix a physical layout (topology) of the involved hardware, i.e. track segments, sets of points, signals, platforms, emergency systems, balises, etc. In practice, the physical layout is specified by a document called a "track plan", see Figure 8.3 for an example. The track plan uses a graphical notation for precisely determining what each piece of equipment is, and in some cases how the pieces are connected together.

Each class of hardware previously indicated can be broken down into subclasses that determine their attributes. For example, signals are sub-classed into main, distant, call-on or shunt; each of these sub-classes has a different

purpose, track plan notation and attributes. The main signals and distant signals are further categorised by the aspects they can show.

Remark

The railway domain has a well-developed terminology that was created and has been evolved over the past 200 years by engineers. Many of these terms are not-intuitive as they were first used to describe the behaviour of, at the time, present day technology. One such example: the terms normal and reverse which today are mostly used to describe the position of a set of points, but historically referred to the position of a lever (in a signal box) that controlled the set of points, reverse being the case where the lever was pulled from the normal position. For a listing of the relevant terminologies please see Appendix A.

A fuller description of railway signalling can be found in [KR01, Noc02, Lea03], a series of books published by the Institution of Railway Signal Engineers.

Many techniques have been used to define formal models the physical layout of a railway network [P07], some of the successful techniques include algebraic specification [BGP95, Bjø06], graph theory [Han98, Mon92] and predicate logic [Kan08, Eri97a]. Common between all the approaches is that the layout contains all identifiers of the components, what these components are, their relationships to other components, and their attributes. From a mathematical perspective, the layout forms a signature for subsequent definitions. Notably these signatures (and a logic) form the language which defines the interlocking system. In the interest of feasibility, a railway network consists of a number of composable layouts, typically one (or more) per railyard, e.g. station or junction.

As a running example in this chapter, consider the basic topology map in Figure 8.3 that depicts a set of points and a number of signals. Figure 8.1 and Figure 8.2 describe the components of the diagram.

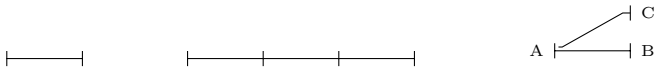


Figure 8.1: Basic track segments, a hatch mark (vertical line) delimits the segment. Left: a singular track segment. Centre: 3 connected segments. Right: a set of points, a singular track segment consisting of two units. The normal and reverse configurations of the set of points is depicted by the break in the line near A, e.g. normal is when travel between A and B is possible, and reverse is when travel between A and C is possible.

Figure 8.2: Selection of different signals. Left: 1 aspect signal, only shows danger aspect. Centre left: 2 aspect signal, shows danger or proceed aspects. Centre right: 3 aspect signal, shows danger, caution or proceed aspects. Right: signal is only visible when travelling from A to B, also known as facing direction.

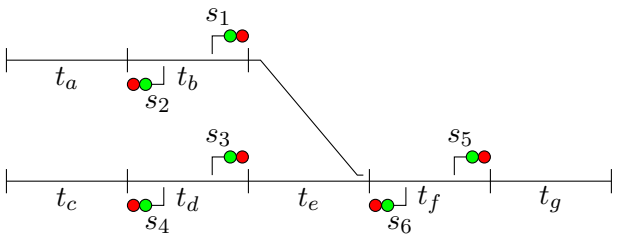


Figure 8.3: Topology of a set of points t_e and adjacent track segments; t_e consists of two units, the first unit from s_3 to s_5 forms the normal position and the second from s_1 to s_5 forms the reverse position. There are 7 track segments $(t_a, t_b, \ldots)$ and 6 signals $(s_1, s_2, \ldots)$.

Although, the layouts essentially consist of a number of sets and relations on these sets, choosing a definition for the layout in Agda that would help and not hinder the proofs or verifications required a significant amount of exploration. A number of different definitions were explored in an attempt to find a suitable trade-off between specifying and proving properties about the layouts. The specification was required to be in a human readable format that is close enough to the actual railway documents to ease validation. In this thesis, a simplified layout is used where only main signals and track segments are considered. The layout consists of the connections between

track segments, and locations of the signals (given by a pair of connected track segments). Although simple, these layouts are expressive enough to model a wide variety of topologies; no information is provided about the functionality of a track segment, e.g. set of points, cross-over or turntable. In essence, these models form a graph where the nodes are track segments and the directed edges are directed connections; an edge can be annotated by signals. These layouts are specified in module `RDM.RailYard` of Appendix F as follows:

$$
\begin{array}{l}
\text{record PhysicalLayout} : \text{Set}_1 \text{ where} \\
\quad \text{field} \\
\quad\quad \text{Segment} \quad : \text{Set} \\
\quad\quad \text{Signal} \quad : \text{Set} \\
\quad\quad \text{connections} : \text{Segment} \to \text{List Segment} \\
\\
\quad \text{Connected} : \text{Segment} \to \text{Segment} \to \text{Set} \\
\quad \text{Connected } ts_1\ ts_2 = ts_2 \text{ isin connections } ts_1 \\
\\
\quad \text{field} \\
\quad\quad \text{signalLocation} : \text{Signal} \to \text{SignalLocation}_{\text{Segment,Connected}}
\end{array}
$$

Here Segment and Signal are sets of identifiers for track segments and signals, respectively. The field connections is a function mapping a track segment onto a list of its adjacent neighbours. The relation $\text{Connected } s\ t$ is inhabited *iff* segment s is directly connected to segment t; it is realised by the use of the relation isin, which is defined as follows:

$$
\begin{array}{l}
\_\text{isin}\_ : \forall\ \{A\}\ .\ A \to \text{List } A \to \text{Set} \\
a \text{ isin } [\,] \quad\quad\quad = \bot \\
a \text{ isin } (a' :: as) = a \equiv a' \vee a \text{ isin } as
\end{array}
$$

The field signalLocation maps a signal to a SignalLocation. This is a structure that wraps up two adjacent track segments; adjacency is enforced with the use of the relation Connected. SignalLocation is defined as follows:

$$
\begin{array}{l}
\text{record SignalLocation } (S : \text{Set})\ (R : S \to S \to \text{Set}) : \text{Set where} \\
\quad \text{field} \\
\quad\quad \text{facing} \quad : S \\
\quad\quad \text{trailing} \quad : S \\
\quad\quad \text{connected} : R \text{ facing trailing}
\end{array}
$$

The topology shown in Figure 8.3 can be coded as a physical layout by defining the sets as $\text{Segment}_{\mathit{fig8.3}} = \{t_a, \ldots, t_g\}$ and $\text{Signal}_{\mathit{fig8.3}} = \{s_1, \ldots, s_6\}$.

Definition of $\text{connections}_{fig8.3}$ is canonical from Figure 8.3,

$$connections_{fig8.3}\ t_a = [t_b]\ ,\ \ldots\ ,\ \text{connections}_{fig8.3}\ t_e = [t_f, t_d, t_b]$$

Depending on the direction of travel a signal s might not be visible, this is encoded by the naming of the fields in the SignalLocation record, i.e. "s_2 is observable when travelling from t_b to t_a" is encoded as:

$$\begin{aligned}&\text{signalLocation } s_2 = \text{record } \{\\ &\qquad\qquad \text{facing } = t_b\ ;\ \text{trailing } = t_a\ ;\ \text{connected } = \text{inj}_1\ \text{refl}\ \}\end{aligned}$$

There is no attempt to model the state of these topologies here, but the operational behaviour is described by the control tables (see next section). Instead, theses layouts are only meant as a signature for subsequent definitions. Later in Section 8.3 abstract layouts are introduced, and shown to follow from these layouts. It is these abstract layouts which will have their states modelled.

8.1.1 Track Segments

These models are intended to be extensible with respect to track segments. In early attempts, explicit provisions were made for sets of points, but this became cumbersome and raised questions about which types of track segments should be supported. Therefore, it was decided to allow connections between segments to be non-symmetric, and allow an individual segment *ts* to maintain a list of which other segments it is possible to travel to, from *ts*. These possible moves are because track segments can change their state. For example, a set of points can be in the normal or reverse position.

It might be preferable when modelling track segments generically to split up the list of connected segments into two disjoint lists (left and right). This would allow for removal of impossible moves over track segments by requiring that a train does not both enter and leave a track segment from the same list; however, complicated track segments, such as a turn table, would not be formalisable. For example in Figure 8.3, regardless of the configuration of the set of points, t_e, it should not be possible to travel from t_b to t_e, and then directly to t_d. The splitting of the connected segments here would ensure that a train entering from the left would leave from the right (or vice versa). A detailed discussion of generic track segment modelling can be found in [BGP95].

In these abstract specifications and models, it is not required to know the underlying state of a track segment. This is because much of this thesis is

concerned with safety verification, in which, it is crucial the track segment does not change its state while in use, but which state it is in, is not as important. Therefore, the track segment needs to be *locked* while in use; this lock will be explained later.

The control tables enrich the information here; they identify which track segments have what functionality. For example, sets of points are identified in the control table but not in the physical layout.

8.2 Control Table

The physical layouts presented do not describe the operational requirements of the topology, e.g. when a signal can show a proceed aspect. Following the long-standing practices in the railway domain, the operational requirements are defined by control tables. These consist of a number of (tableau) documents detailing when a piece of hardware can be controlled to do something. Moreover, the control tables form a specification of the control system, and are signed-off by domain experts before the development of the control system begins. For this reason, the safety of the railway is delegated to control tables.

The control tables could become difficult to understand, particularly when there is a complex arrangement of signals and sets of points. Signal engineers simplify the definition of a control table by the abstract concept of a route. Informally, routes start at signals and consist of a sequence of contiguous track segments up-to the next visible signal. From the perspective of railway signalling, routes form the smallest unit of the topology that a train is authorised to occupy. Routes are abstract as they do not correspond to any physical entity in the layout. In this work, due to the simplicity of the physical layouts, the control tables only define the conditions under when a route can be set[2], this includes which track segments must be unoccupied, and the required states of selected track segments.

These conditions determine when it is permissible to set a route. They form an abstract specification for a portion of the railway network, and are relations between physical objects in the layout and abstract concepts such as routes, see Figure 8.4 for an archetypal control table. One use of these tables is during the development of interlocking software; in such cases, control tables and layouts are used as a specification by developers, and testers derive test cases from them.

[2]The route must also be requested, in addition to these constraints holding, for it to be set.

Route	Signal		Tracks	Points		Facing
	Start	End		Normal	Reverse	
R_1	s_1	s_5	t_e t_f	–	t_e	–
R_2	s_3	s_5	t_e t_f	t_e	–	–
R_3	s_6	s_2	t_e t_b	–	t_e	t_e
R_4	s_6	s_4	t_e t_d	t_e	–	t_e

Figure 8.4: Control table for topology in Figure 8.3. R_1 can be set when track segments t_e and t_f are unoccupied and set of points t_e is in the reverse position. The minimal information depicted here is typical of the railway domain, in practice there is more information pertaining to speed limits, emergency events and specific signalling schemes.

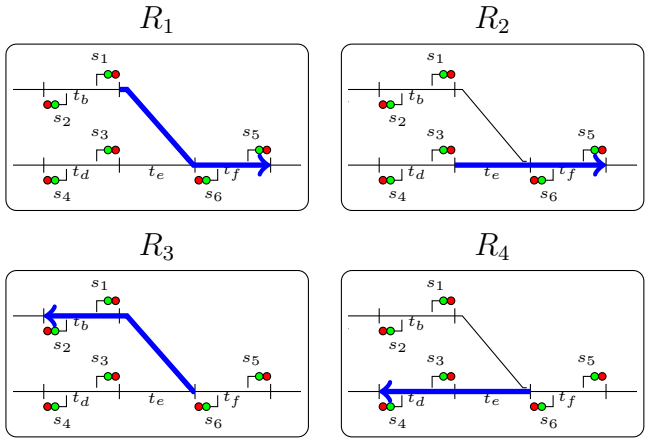


Figure 8.5: Routes from control table in Figure 8.4 depicted.

Automatically deriving verification conditions from the control tables is possible. This is because the semantics of the control tables are well defined. Thus, if an interlocking system fulfils these conditions, then it correctly refines the control table. Therefore, assuming the verification system is certified, this would considerably reduce the amount of effort/money required to test the system is correct. This problem is not inherently interesting and has not been explored in depth, in this work; however, a basic verification condition generator is outlined in Section 10.1.

From an abstract mathematical perspective, control tables are sentences built over a physical layout. In this work, the control tables are defined to be a list of relations built over a set of route identifiers and components of the physical layout. The only relation considered here formalises the conditions under when a route can be set, which corresponds to an entry in the table

of Figure 8.4. Moreover, each entry of the control table corresponds to a route. It would be an easy matter to extend the relations considered, but it was not necessary for the verifications performed during this work. Let p be a physical layout, Segment_p and Signal_p are the set of segments and signals projected from p. An entry of the control table is defined as follows:

$$
\begin{array}{l}
\text{record ControlTableEntry}_p : \text{Set where} \\
\quad \text{field} \\
\qquad \begin{array}{ll}
\text{start} & : \text{Signal}_p \\
\text{segments} & : \text{List Segment}_p \\
\text{normalpoints} & : \text{List Segment}_p \\
\text{reversepoints} & : \text{List Segment}_p \\
\text{facing} & : \text{List Segment}_p
\end{array}
\end{array}
$$

This is a syntactical object, correctness is enforced by the definition of the control table. An instance

$$
\begin{array}{l}
\text{record } \{ \\
\quad \begin{array}{llllll}
\text{start} & = x & ; \text{segments} & = & \overrightarrow{ts} & ; \\
\text{normalpoints} & = \overrightarrow{np} & ; \text{reversepoints} & = & \overrightarrow{rp} & ; \\
\text{facing} & = \overrightarrow{f} & & & & \}
\end{array}
\end{array}
$$

expresses that an unnamed route starts at signal s, uses track segments ts_i and requires points np_j/rp_k to be in the normal/reverse position, respectively. $\vec{f}$ identifies all segments (typically sets of points) in the route that are traversed in the facing direction. Although it is possible to compute the list of normal/reverse points and facing segments from the physical layout, they have been added to simplify subsequent definitions (and keep accordance with practices in the railway domain). In this representation, the end signal is omitted as it would require many parallel routes when a route terminated at a track segment that contained multiple signals; this is because the next route would have been defined by the end signal of the current route, hence, there would need to be a number of parallel routes that only differ by the end signal. The control table maps routes to control table entries, and provides proofs about the well-formedness of the table. The first part of the table is defined as follows:

$$
\begin{array}{l}
\text{record ControlTable}_p : \text{Set where} \\
\quad \text{field} \\
\qquad \begin{array}{ll}
\text{Route} & : \text{Set} \\
\text{RouteEq} & : (r_1\ r_2 : \text{Route}) \to r_1 \equiv r_2 \vee r_1 \not\equiv r_2 \\
\text{entries} & : \text{Route} \to \text{ControlTableEntry}_p \\
\text{connections} & : \text{Route} \to \text{List Route}
\end{array}
\end{array}
$$

Decidable propositional equality will be required on the set of routes for subsequent proofs.

Before introducing the second part of the table that formalises the well-formedness of the first part of the table, an improved notation is introduced. For a given route rt in control table c, its fields can be projected out from its entry in the table by omitting the entry. That is, instead of writing

$$\text{segments}_{\text{entries}_c\ rt}$$

to obtain a list of segments in route rt. When ambiguities do not arise, we omit entries_c. The above is simply written as follows:

$$\text{segments}_{rt}$$

To express the well-formedness of a control table, we define the following relations over the first part of the table, and they form part of the definition of the ControlTable record.

$$\begin{aligned}
&\text{Connected} : \text{Route} \to \text{Route} \to \text{Set}\\
&\text{Connected}\ rt_1\ rt_2 = rt_2\ \text{isin}\ (\text{connections}\ rt_1)
\end{aligned}$$

$$\begin{aligned}
&\text{SegInRoute} : \text{Segment}_p \to \text{Route} \to \text{Set}\\
&\text{SegInRoute}\ ts\ rt = ts\ \text{isin}\ \text{segments}_{rt}
\end{aligned}$$

$$\begin{aligned}
&\text{FacingInRoute} : \text{Segment}_p \to \text{Route} \to \text{Set}\\
&\text{FacingInRoute}\ ts\ rt = ts\ \text{isin}\ \text{facing}_{rt}
\end{aligned}$$

$$\begin{aligned}
&\text{NormalInRoute} : \text{Segment}_p \to \text{Route} \to \text{Set}\\
&\text{NormalInRoute}\ ts\ rt = ts\ \text{isin}\ \text{normalpoints}_{rt}
\end{aligned}$$

$$\begin{aligned}
&\text{ReverseInRoute} : \text{Segment}_p \to \text{Route} \to \text{Set}\\
&\text{ReverseInRoute}\ ts\ rt = ts\ \text{isin}\ \text{reversepoints}_{rt}
\end{aligned}$$

The well-formedness of the table is then given by a number of axioms that guarantee properties such as all routes must contain at least one track segment, or that all pairs of connected routes have their first/last track segments connected with a signal. A conjunction of the following 7 axioms defines the well-formedness. These axioms form the second part of the table where each axiom becomes a field.

$$\forall rt\ \exists ts\ .\ \text{SegInRoute}\ ts\ rt \qquad (\mathit{NonEmptyRoutes})$$

$$\forall rt_1\ rt_2\ rt_3\ .\ \text{Connected}\ rt_1\ rt_2 \rightarrow \text{Connected}\ rt_3\ rt_2 \rightarrow (\exists ts\ .\ \text{SegInRoute}\ ts\ rt_1 \wedge \text{SegInRoute}\ ts\ rt_3) \quad (\textit{WellFormed})$$

$$\forall rt_1\ rt_2\ .\ \text{Connected}\ rt_1\ rt_2 \rightarrow \text{facing}_{\text{signalLocation}_p\ (\text{start}_{rt_2})} \equiv \text{segments}_{rt_1}[\textit{last}] \wedge\ \text{trailing}_{\text{signalLocation}_p\ (\text{start}_{rt_2})}\ \text{segments}_{rt_2}[0] \quad (\textit{RoutesConnected})$$

$$\forall rt\ i\ .\ 0 \leq i < (\text{length}\ (\text{segments}_{rt} - 1)) \rightarrow \text{Connected}_p\ \text{segments}_{rt}[i]\ \text{segments}_{rt}[i+1] \quad (\textit{Continuous})$$

$$\forall rt\ ts\ .\ \text{FacingInRoute}\ ts\ rt \rightarrow \text{SegInRoute}\ ts\ rt \quad (\textit{InRoute1})$$

$$\forall rt\ ts\ .\ \text{NormalInRoute}\ ts\ rt \rightarrow \text{SegInRoute}\ ts\ rt \wedge \neg(\text{ReverseInRoute}\ ts\ rt) \quad (\textit{InRoute2})$$

$$\forall rt\ ts\ .\ \text{ReverseInRoute}\ ts\ rt \rightarrow \text{SegInRoute}\ ts\ rt \wedge \neg(\text{NormalInRoute}\ ts\ rt) \quad (\textit{InRoute3})$$

Informally, for a given route rt these axioms ensure that segments_{rt} is not empty and that the segments are contiguous. The segments identified by normalpoints_{rt}, $\text{reversepoints}_{rt}$ and facing_{rt} are all contained in segments_{rt}. For two connected routes rt_1 and rt_2, i.e. travel from rt_1 to rt_2 is possible, then the last segment of rt_1 and first segment of rt_2 are connected, and that there is a signal facing in the correct direction between these two segments. For two routes rt_1 and rt_3 that are both connected to rt_2, then rt_1 and rt_3 share a common track segment. This common track segment (usually the berth segment of rt_2) is required for subsequent proofs, it will help to show that at most one train enters rt_3 at a time.

The formal definition of a control table is in module `RDM.RailYard` of Appendix F.

Remark

To prove the theorems in Section 8.4.1, it is not required for the routes to not contain other signals. However, in practice to mitigate sources of confusion for the drivers, routes do not contain facing signals.

8.3 Abstract Layout

The physical layouts and control tables that have been presented are low-level representations corresponding to actual documents found in the railway

domain. When attempting to show that the signalling principles imply domain safety, many of the actual details are not required and are not in a directly amenable format. For this reason, an abstract representation of layouts is considered. This abstract representation combines the relevant parts of the physical layout and control table into a signature expressive enough to represent the signalling principles and domain safety.

The abstract layouts are concerned with trains, routes, and how the routes are connected. This is because routes are the smallest unit of the topology that a train is allowed to occupy, and they are guarded by signals. It is required to know which segments belong to a route, but the order of these segments (or even if they are connected at all) is irrelevant. The routes are required to fulfil the axiom (*WellFormed*) from the previous section. The abstract layout is defined in module `RDM.fixedtrains` of Appendix F as follows:

$$
\begin{array}{l}
\text{record Layout} : \text{Set}_1 \text{ where} \\
\quad \text{field} \\
\qquad \begin{array}{lcl}
\text{Segment} & : & \text{Set} \\
\text{Train} & : & \text{Set} \\
\text{TrainEq} & : & (t_1\ t_2 : \text{Train}) \to t_1 \equiv t_2 \vee t_1 \not\equiv t_2 \\
\text{Route} & : & \text{Set} \\
\text{RouteEq} & : & (r_1\ r_2 : \text{Route}) \to r_1 \equiv r_2 \vee r_1 \not\equiv r_2 \\
\text{RouteConnected} & : & \text{Route} \to \text{Route} \to \text{Set} \\
\text{SegInRoute} & : & \text{Segment} \to \text{Route} \to \text{Set} \\
\text{FacingInRoute} & : & \text{Segment} \to \text{Route} \to \text{Set} \\
\text{WellFormed} & : & \forall rt_1\ rt_2\ rt_3\ . \\
 & & \quad \to \text{RouteConnected}\ rt_1\ rt_2 \\
 & & \quad \to \text{RouteConnected}\ rt_3\ rt_2 \\
 & & \quad \to \exists ts\ .\ \text{SegInRoute}\ ts\ rt_1 \wedge \text{SegInRoute}\ ts\ rt_3
\end{array}
\end{array}
$$

In the later proofs, decidable propositional equality is required on trains and routes. It should be noted that, in this model, the set of trains is fixed, therefore, the models do not allow for trains to couple (or decouple). One problem with allowing trains to couple or decouple is that there would be two trains in the same segment. Possibly this could be modelled by (1) explicitly representing the front and rear positions of trains, (2) having a minimum safe speed, or (3) allocating special track segments for these manoeuvres.

A remark about facing segments: In the layout, it was decided not to include a track segment connection graph, the types of the segments (e.g. set of points, cross-overs, diamonds, or a yet to be designed track segment), and the possible states of these segments (e.g. normal or reverse for a set of points). If this information had been included, then it would be possible to deduce, for

a given route, which segments are traversed in the facing direction. Instead, it was decided to include a relation indicating which segments are traversed in the facing direction, this information is specified in the control table, and it is critical that it is validated. This was decided because it would have been an awkward activity to undertake while maintaining extensibility for new designs of track segments. Although this is possible in Agda due to dependent types, it is left as future work. Essentially the layouts would need to be augmented with a set of track segment types, and each segment type would be assigned a state space. A relation would be needed on segment states that will determine whether it is possible to transition from one segment to another.

In these models, the lie (e.g. state) of a track segment is not relevant to the safety of the railway, but the lie is only relevant when verifying that a system correctly refines a control table. For linear track segments, their lie does not change so it is irrelevant. For a set of points, the lie is either normal or reverse; the set of points are safe regardless of the lie. However, it is critical that the lie does not change while being traversed in the facing direction, this is because the nose of the point will get jammed on the wrong side of a wheel, resulting in a derailment. Note that when traversed in the trailing (converging) direction, if the lie is changed, it is not a safety concern as the train will not derail as the wheels will not get jammed on anything.

The abstract layouts are trivially constructable from the physical layout, control table and set of train identifiers. For completeness the translation is given as follows:

$$
\begin{aligned}
&\text{toLayout} : (p : \text{PhysicalLayout}) \\
&\qquad \to \text{ControlTable}_p \\
&\qquad\quad \to (\text{Train} : \text{Set}) \\
&\qquad\qquad \to ((t_1\ t_2 : \text{Train}) \to t_1 \equiv t_2 \vee t_1 \not\equiv t_2) \\
&\qquad\qquad\quad \to \text{Layout} \\
&\text{toLayout}\ p\ c\ t\ teq\ =\ \text{record}\ \{ \\
&\qquad\qquad\qquad \mathit{Segment}\ = \text{Segment}_p\ ; \\
&\qquad\qquad\qquad \mathit{Train}\ = t\ ; \\
&\qquad\qquad\qquad \mathit{TrainEq}\ = teq\ ; \\
&\qquad\qquad\qquad \mathit{Route}\ = \text{Route}_c\ ; \\
&\qquad\qquad\qquad \mathit{RouteEq}\ = \text{RouteEq}_c\ ; \\
&\qquad\qquad\qquad \mathit{RouteConnected}\ = \text{Connected}_c\ ; \\
&\qquad\qquad\qquad \mathit{SegInRoute}\ = \text{SegInRoute}_c\ ; \\
&\qquad\qquad\qquad \mathit{FacingInRoute}\ = \text{FacingInRoute}_c\ ; \\
&\qquad\qquad\qquad \mathit{WellFormed}\ = \text{WellFormed}_c\ \}
\end{aligned}
$$

8.3.1 Layout State

Up to this point all the models presented do not consider the state of the topology. The state defines whether the track segments are locked, and what aspect a signal displays; also part of the state includes the position of the trains. It has previously been noted that these models are concerned with safety and not operational correctness, therefore, the lie of a track segment is not included, only whether it is locked. For information about the operational correctness, including the lie of a track segment see Section 10.2.

The semantics of a locked segment is that 1) it cannot have its lie changed by the control system, and 2) that it is detected (and clamped) to be in a valid position. Valid in this sense means that it is not in-between recognised positions. For instance, a locked set of points is mechanically clamped in the normal or reverse positions, and it did not get jammed between these positions. Historically, it was common place that pieces of coal fell from the trains steam engine, and sometimes they would fall between the blades of the points and prevent them from being normal or reverse. For obvious reasons, 1) is required to prevent derailments occurring due to controller error (possibly a human), and 2) ensures that the segment does not move after being controlled into a position, perhaps because of vibrations caused by the train, or general wear-and-tear. In both cases, the concern is that when a train travels over the segment in the facing (diverging) direction, the train attempts to travel along orthogonal segments, resulting in a derailment. When travelling in the trailing (converging) direction over an unlocked segment there is only a small chance of derailment, it is more likely, if it is not in a valid position, that the segment will be damaged. It should be noted that the vast majority of train derailments are attributed to faulty sets of points; modern high-speed train lines exaggerate this issue, since it might lead to significant of loss of life. For this reason, in the UK, it became a legal requirement that all facing moves by passenger trains over sets of points must be locked [Cal92]; this gave rise to widespread adoption of locks known as a *facing-point lock* or FPL.

The state of a signal is determined by the aspect it displays. In this work, only 2-aspect signalling is considered. A 2-aspect signal displays either a *proceed* or *danger* aspect. To avoid ambiguities, the aspect of a signal is not given by a colour, but instead by its intended meaning; this is partly for historical reasons, as different train operators used different signalling schemes.

The state of an abstract layout l is defined in module `RDM.fixedtrains`

of Appendix F as follows:

$$\begin{array}{l}\text{record LayoutState}_l : \text{Set where}\\ \quad\text{field}\\ \quad\quad\begin{array}{lll}\text{trainRoute} & : & \text{Train}_l \to \text{Route}_l\\ \text{signalAspect} & : & \text{Route}_l \to \text{Aspect}\\ \text{locked} & : & \text{Segment}_l \to \text{Locking}\end{array}\end{array}$$

where

$$\text{Aspect} := \{\text{Proceed}, \text{Danger}\}$$

and

$$\text{Locking} := \{\text{Locked}, \text{Unlocked}\}$$

The control systems that are of interest, in this work, are discrete and are modelled by a stream of topological states indexed by time.

$$\text{nthState}_l : \mathbb{N} \to \text{LayoutState}_l$$

In Chapter 9 it is shown how to define the implementation of this function from a ladder logic program.

In the following, an improved notation is used to reference the components of a state. For an abstract layout l, time t and train tr, instead of writing

$$\text{trainRoute}_{(\text{nthState}_l\ t)}\ tr$$

when no ambiguities occur, nthState_l is omitted, and the above is written as follows:

$$\text{trainRoute}_t\ tr$$

Similarly for the other components of the state.

For an abstract layout l, the function nthState_l models an arbitrary (infinite) sequence of states. Many of the potential sequences are undesirable, for instance, a sequence that contains a state where two or more trains occupy the same route, or a sequence that contains successive states where a train moves ad-hoc between possibly unconnected routes. There are two distinct issues here. The first is that some sequences are unsafe (according to the domain's safety) and are discussed in the next section. The second issue is that some of these sequences are physically impossible. The issue of trains moving ad-hoc is physically impossible under normal operations, assuming that helicopters (or similar) are not picking up the trains and moving them between unconnected routes. This issue is avoided by placing a well-formedness constraint upon the sequences by requiring that nthState_l fulfils the axiom:

Correct-Train

$$\forall t\ train\ .\ (\text{trainRoute}_t\ train \equiv \text{trainRoute}_{t+1}\ train) \vee (\text{RouteConnected}_l\ (\text{trainRoute}_t\ train)\ (\text{trainRoute}_{t+1}\ train) \wedge \text{signalAspect}_t\ (\text{trainRoute}_{t+1}\ train) \equiv \text{Proceed})$$

Informally, this axiom states that trains only travel between two connected routes, and where the guarding signal between these routes displays a proceed aspect.

8.4 Domain Safety and Signalling Principles

In this section, it is demonstrated how to remove unsafe sequences from the models. This is done by defining a property which expresses the domain safety, and then showing that this property holds for all states in an arbitrary sequence. In general, showing this property always holds requires that a number of assumptions are made about the sequences. These assumptions formalise a minimal selection of the domain's *safety principles*. Informally a safety principle is a rule-of-thumb used by the designers and testers of a critical system. In a formal setting, the safety principles are lemmata expressing domain knowledge. Importantly, with respect to the railway domain, safety principles are synonymous with signalling principles. In later chapters, it will be shown how to prove that an implementation of nthState_l fulfils these safety principles (assumptions).

Initially the concept of domain safety, must be validated and formalised. The formalisation of safety should be clear and easy to understand by domain experts. For instance, in the Pelicon crossing example from Section 1.1.1, the concept of safety is given as follows:

at any point in time exclusive use of the crossing is given to pedestrians or road traffic, but not both

In the railway domain, one such safety requirement is that

at any point in time trains do not derail or collide

To express this requirement over the domain models presented, it is required to formalise what a derailment and a collision are. In Section 8.3.1, it was discussed that a major cause of derailments is when trains travel in the facing direction over a set of points that are not in a well-formed lie, or changes its lie while occupied. Therefore, the condition that trains do not derail is formalised as follows:

S1

$$\forall t\ train\ segment\ .\ \text{FacingInRoute}_l\ segment\ (\text{trainRoute}_t\ train) \rightarrow \text{locked}_t\ segment \equiv \text{Locked}$$

This states that whatever route a train is in, all segments in that route which are traversed in the facing direction are locked. As the facing segments are identified in the control table, and for this condition to be trustworthy it is imperative that the control tables are validated.

The formalisation that trains do not collide is given by a statement that ensures that no two trains occupy the same segment. In this setting, an over approximation is made, such that the potential segments that two distinct trains are allowed to occupy do not overlap.

S2

$$\forall t\ train_1\ train_2\ segment\ .\ train_1 \not\equiv train_2 \rightarrow \neg(\text{SegInRoute}_l\ segment\ (\text{trainRoute}_t\ train_1) \wedge \text{SegInRoute}_l\ segment\ (\text{trainRoute}_t\ train_2))$$

Therefore, safety in the railway domain is given by a conjunction of (*S1*) and (*S2*). In this work, the above formalisations of safety and the models have not been validated; this is because of time constraints on the project.

Before proving that (*S1*) and (*S2*) always hold, it is required to introduce 4 signalling principles. These principles, as already noted, will remove the unsafe sequences.

Signalling Principle 1. (Opposing Signals)

$$\forall t\ route_1\ route_2\ segment\ .\ route_1 \not\equiv route_2 \rightarrow \text{SegInRoute}_l\ segment\ route_1\ \wedge\ \text{SegInRoute}_l\ segment\ route_2 \rightarrow \text{signalAspect}_t\ route_1 \equiv \text{Danger}\ \vee\ \text{signalAspect}_t\ route_2 \equiv \text{Danger}$$

The opposing signals principle ensures that for two distinct signals that protect the same part (or portion) of track must never both be set to proceed aspects. The issue here is straightforward, suppose the situation in Figure 8.6, where there is a train at x and another at z, and both are attempting to enter the same portion of track (y). Should there be no train on y, then at most one of these trains should be shown a proceed aspect, or else there is a significant risk of a collision occurring. That is, signal 1 or signal 2 can clear, but not both.

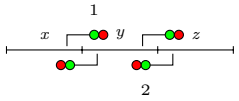


Figure 8.6: Opposing Signals

Signalling Principle 2. (Signals Guard)

$$\forall t\ \mathit{train}\ \mathit{segment}\ \mathit{route}\ .\ \mathrm{SegInRoute}_l\ \mathit{segment}\ (\mathrm{trainRoute}_t\ \mathit{train}) \rightarrow \mathrm{SegInRoute}_l\ \mathit{segment}\ \mathit{route} \rightarrow \mathrm{signalAspect}_t\ \mathit{route} \equiv \mathrm{Danger}$$

The guarding signals signalling principle prevents a signal being cleared while a train is occupying the portion of track that the signal is protecting. See Figure 8.7, suppose there are two trains located on segments x and y, and the signal cleared before the train located at y vacated the segment, then the train located at x could collide with the train located at y.

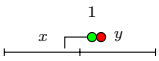


Figure 8.7: Signals Guard

Signalling Principle 3. (Proceed Locked)

$$\forall t\ \mathit{route}\ \mathit{segment}\ .\ \mathrm{signalAspect}_t\ \mathit{route} \equiv \mathrm{Proceed} \rightarrow \mathrm{FacingInRoute}_l\ \mathit{segment}\ \mathit{route} \rightarrow \mathrm{locked}_t\ \mathit{segment} \equiv \mathrm{Locked}$$

The proceed locked signalling principle formalises that if a signal is clear, then all segments protected by the signal in the facing direction are also locked. That is the situation in Figure 8.8; if signal 1 is cleared, then the set of points y is locked. The same constraint is not applied to signal 2 as it is protecting y in the trailing direction. The motivation behind this constraint is that if a train should traverse an unlocked set of points in the facing direction, then there is a significant risk of a derailment.

Signalling Principle 4. (Train Holds Lock)

$$\forall t\ \mathit{train}\ \mathit{segment}\ .\ \mathrm{SegInRoute}_l\ \mathit{segment}\ (\mathrm{trainRoute}_{t+1}\ \mathit{train}) \rightarrow \mathrm{locked}_t\ \mathit{segment} \equiv \mathrm{Locked} \rightarrow \mathrm{locked}_{t+1}\ \mathit{segment} \equiv \mathrm{Locked}$$

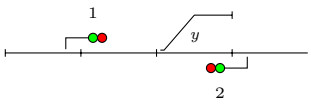


Figure 8.8: Proceed Locked

The final signalling principle prevents a segment in an occupied (or about to be occupied) route from being unlocked, if it was locked in the previous state. If a set of points is unlocked while occupied, then there is a risk that the lie of the points could change, which could result in a derailment.

8.4.1 Proof that Safety Follows

In this section, we prove by induction over time that the signalling principles implies the safety requirements. For this, we define following axioms that correspond to the base-cases of (*S1*) and (*S2*).

S1-Init

$$\forall\, train\ segment\ .\ \text{FacingInRoute}_l\ segment\ (\text{trainRoute}_0\ train) \rightarrow \text{locked}_0\ segment \equiv \text{Locked}$$

S2-Init

$$\forall\, train_1\ train_2\ segment\ .\ train_1 \not\equiv train_2 \rightarrow \neg(\text{SegInRoute}_l\ segment\ (\text{trainRoute}_0\ train_1) \wedge \text{SegInRoute}_l\ segment\ (\text{trainRoute}_0\ train_2))$$

The following two theorems prove that safety follows by the signalling principles. The first theorem proves that the facing-point locks are engaged for all facing moves over the segments. The second theorem shows that the trains do not collide. This order of presentation is because the first theorem is simpler. See module `RDM.fixedtrains` in Appendix F for full technical details.

Theorem 8.4.1 (Facing-Point Lock)**.** *Assume an abstract layout l, and assume axioms (Correct-Train), (S1-Init), and signalling principles 3 and 4 hold. Then (S1) holds.*

Proof. Recall that (*S1*) is defined as follows:

$$\forall t\ train\ segment\ .\ \text{FacingInRoute}_l\ segment\ (\text{trainRoute}_t\ train) \rightarrow \text{locked}_t\ segment \equiv \text{Locked}$$

Proof by induction on time t

Case $t = 0$**:** Follows by (*S1-Init*).

Case $t = t' + 1$**:** Define

$$\begin{aligned}&\text{Stationary} : \text{Train}_l \to \text{Set}\\&\text{Stationary } tr \;=\; \text{trainRoute}_t\ tr \equiv \text{trainRoute}_{t'}\ tr\end{aligned}$$

Note that because of the decidability of the equality on routes, we can make the full case-distinction:

Case Stationary *train***:**

$$\begin{aligned}&\text{FacingInRoute}_l\ \textit{segment}\ (\text{trainRoute}_{t'}\ \textit{train})\\&\quad\to \text{locked}_{t'}\ \textit{segment} \equiv \text{Locked}\end{aligned}$$

by induction hypothesis

$$\begin{aligned}&\text{FacingInRoute}_l\ \textit{segment}\ (\text{trainRoute}_t\ \textit{train}) \qquad (*)\\&\quad\to \text{locked}_{t'}\ \textit{segment} \equiv \text{Locked}\end{aligned}$$

rewrite induction hypothesis using (Stationary *train*)

$$\begin{aligned}&\text{FacingInRoute}_l\ \textit{segment}\ (\text{trainRoute}_t\ \textit{train})\\&\quad\to \text{locked}_t\ \textit{segment} \equiv \text{Locked}\end{aligned}$$

apply $(*)$ *to signalling principle 4*

Case ¬(Stationary *train*)**:**

$$\text{signalAspect}_{t'}\ (\text{trainRoute}_t\ \textit{train}) \equiv \text{Proceed} \qquad (*)$$

by case assumption applied to (Correct-Train)

$$\begin{aligned}&\text{FacingInRoute}_l\ \textit{segment}\ (\text{trainRoute}_t\ \textit{train}) \qquad (**)\\&\quad\to \text{locked}_{t'}\ \textit{segment} \equiv \text{Locked}\end{aligned}$$

apply $(*)$ *to signalling principle 3*

$$\begin{aligned}&\text{FacingInRoute}_l\ \textit{segment}\ (\text{trainRoute}_t\ \textit{train})\\&\quad\to \text{locked}_t\ \textit{segment} \equiv \text{Locked}\end{aligned}$$

apply $(**)$ *to signalling principle 4*

□

Theorem 8.4.2 (Collision Free)**.** *Assume an abstract layout l, and assume axioms (Correct-Train), (S2-Init), and signalling principles 1 and 2 hold. Then (S2) holds.*

Remark

The proof is structurally similar to the proof of Theorem 8.4.2. Both proofs make a case analysis over the trains changing routes.

Proof. Recall that (*S2*) is defined as follows:

$$\begin{aligned}\forall t\ train_1\ train_2\ segment\ .\ train_1 \not\equiv train_2 \rightarrow \\ \neg(\text{SegInRoute}_l\ segment\ (\text{trainRoute}_t\ train_1) \wedge \\ \text{SegInRoute}_l\ segment\ (\text{trainRoute}_t\ train_2))\end{aligned}$$

Proof by induction on time t

Case $t = 0$**:** Follows by (*S2-Init*).

Case $t = t' + 1$**:** Define

$$\begin{aligned}&\text{Stationary} : \text{Train}_l \rightarrow \text{Set} \\ &\text{Stationary}\ tr\ =\ \text{trainRoute}_t\ tr \equiv \text{trainRoute}_{t'}\ tr\end{aligned}$$

Note that because of the decidability of the equality on routes, we can make the full case-distinction:

Case Stationary $train_1 \wedge$ Stationary $train_2$**:**

$$\begin{aligned}&\neg(\text{SegInRoute}_l\ segment\ (\text{trainRoute}_{t'}\ train_1) \wedge \\ &\quad \text{SegInRoute}_l\ segment\ (\text{trainRoute}_{t'}\ train_2))\end{aligned}$$

by induction hypothesis

$$\begin{aligned}&\neg(\text{SegInRoute}_l\ segment\ (\text{trainRoute}_t\ train_1) \wedge \\ &\quad \text{SegInRoute}_l\ segment\ (\text{trainRoute}_t\ train_2))\end{aligned}$$

rewrite induction hypothesis by case assumption

Case $\neg(\text{Stationary } \mathit{train}_1) \wedge \text{Stationary } \mathit{train}_2$**:**

$$\text{signalAspect}_{t'}\ (\text{trainRoute}_t\ \mathit{train}_1) \equiv \text{Proceed} \qquad (1)$$

by case assumption applied to (Correct-Train)

Assume the antecedent of (*S2*)

$$\exists s\ .\ \text{SegInRoute}_l\ s\ (\text{trainRoute}_t\ \mathit{train}_1) \wedge \text{SegInRoute}_l\ s\ (\text{trainRoute}_t\ \mathit{train}_2) \qquad (*)$$

$$\text{signalAspect}_{t'}\ (\text{trainRoute}_t\ \mathit{train}_1) \equiv \text{Danger} \qquad (2)$$

apply $(*)$ *to signalling principle 2*

By (1) and (2), we get a contradiction.

Case $\text{Stationary } \mathit{train}_1 \wedge \neg(\text{Stationary } \mathit{train}_2)$**:** As the previous case, but with train_1 and train_2 interchanged.

Case $\neg(\text{Stationary } \mathit{train}_1) \wedge \neg(\text{Stationary } \mathit{train}_2)$**:**

$$\text{signalAspect}_{t'}\ (\text{trainRoute}_t\ \mathit{train}_1) \equiv \text{Proceed} \wedge \text{signalAspect}_{t'}\ (\text{trainRoute}_t\ \mathit{train}_2) \equiv \text{Proceed} \qquad (1)$$

by case assumption applied to (Correct-Train)

Assume the antecedent of (*S2*)

$$\exists s\ .\ \text{SegInRoute}_l\ s\ (\text{trainRoute}_t\ \mathit{train}_1) \wedge \text{SegInRoute}_l\ s\ (\text{trainRoute}_t\ \mathit{train}_2) \qquad (*)$$

$$\text{signalAspect}_{t'}\ (\text{trainRoute}_t\ \mathit{train}_1) \equiv \text{Danger} \vee \text{signalAspect}_{t'}\ (\text{trainRoute}_t\ \mathit{train}_2) \equiv \text{Danger} \qquad (2)$$

apply $(*)$ *to signalling principle 1*

By (1) and (2) we get a contradiction.

□

Corollary (Always-Safe)**.** *Assume an abstract layout l. Assume axioms (Correct-Train), (S1-Init), (S2-Init), and signalling principles 1, 2, 3 and 4 hold. Then trains do not collide, nor do they derail.*

Proof. The proof is a conjunction of Theorem 8.4.1 and Theorem 8.4.2. □

8.5 Experiences

A significant amount of effort was put into finding the correct representations for the railway domain. In part, this is because the definitions were required to be type-checked by Agda, so clean mathematical definitions that were used on paper during the design were not directly amenable. For example, the use of finite sets for modelling the topology could cause efficiency and usability issues when pattern matching, however, they do result in clean definitions of the abstract models.

It was not clear what level of abstraction was required for the models. This is a common issue that can be seen in general when exploring the literature relating to verification of railway control systems. Many attempts get 'bogged down' in the details of the models, to such an extend that is it is not clear that the details are required, and resulting in a model that is difficult to validate. For example, in [Bjø06] there is an extensive discussion about extensible modelling of track segments, to the extent that each track segment is a state machine with a common interface. By common interface, it is meant that the segments know how to be composed with other segments. However, in this thesis it was chosen to abstract from details such as these, and instead focus on the core issues of safety and correctness.

The choosing and formalising of the signalling principles was also a cumbersome task as in the UK there are no formal mathematical definitions of these principles, only informal, natural language descriptions [IRS01]. Instead, they are documented using a non-formal language. More information regarding the experiences of finding the correct signalling principles is given in Section 11.2.2. This information includes an example of a mistake that was made during formalising one of the principles.

Reversing Trains. It is noted that the models presented here do not allow trains to be reversed because of signalling principle 2. Consider the scenario in Figure 8.9. There are two routes depicted by the arrows, the first from signal 2 to signal 4 and the second from signal 3 to signal 1. Without loss of generality, assume that a train is in one of these routes, then it will never be able to enter the other route because the centre segment is used by both routes. Therefore, a reversing train violates signalling principle 2.

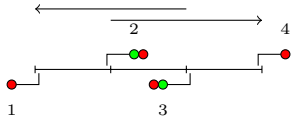


Figure 8.9: Reversing Trains

At one point during the project the concept of a *partial route release* was considered to be added to the model. A partial route release is terminology from the railway domain for when part of a route that a train has already traversed is released [KR01]. This would solve the problem in the above example. Partial route releases were not implemented in our models for simplicity, and lack of time.

Chapter 9
Interlocking Systems

In the previous chapter, a number of discussions about modelling aspects of the railway domain were presented. Importantly the chapter introduced safety with respect to railways. In this chapter, a standard implementation of safe (interlocked) railway control systems is developed, and in the next chapter, a framework is presented to show that the control systems are safe.

Much of the domain knowledge relating to safe railways has been developed over many years. It is sad to say that the majority of this knowledge has been gathered by investigators after tragic disasters. Following these disasters, a number of recommendations were made, and often these recommendations became enshrined in law. This work is based upon UK railway signalling; as such many of the references given for these disasters are from Her Majesty's Railway Inspectorate[1] (HMRI).

Chapter Overview. First, the history of railways is presented, this history shows the need for safety on the railways; hence interlocked control systems. Then the programming language known as ladder logic is introduced and formalised. These ladder logic programs are shown to define decidable transition systems, and then these decidable transition systems are used to construct the nthState of an abstract layout. The chapter is concluded with a discussion of LadderCTL and the Geographic Data programming language.

9.1 History of Interlocking Systems

Since the times of the Greek and Roman empires, the concept of railways or wagonways have been recognised [Lew01]. The early railways were used for a

[1] In recent years (2006 onwards), the organisation responsible for overseeing safety on Britain's railways has been re-branded a number of times. At the time of writing, it is known as the Office of Rail Regulation

limited number of tasks such as helping miners get their ore out of a mine, or getting produce to market. These historic railways were typically powered by people or suitable animals, and the wheels and rails were typically made from wood. Little attention was paid to safety; even a collision would not normally result in a loss of life. Since these early years, a number of improvements have been made. The most notable of which is the speed that the rolling stock travel. This speed increase required a number of technological improvements.

During 1750-1820, arguably one of the most significant advents for railways was developed, rolled wrought iron track [Bir24]. During this period, a number of small development were made that contributed to the start of modern rails, from the production of high quality iron to the design of the wheels and cross sectional shape of the track. These new wheels and tracks could withstand the weight of a steam engine, which was undoubtedly the next significant advancement.

The birth of modern British railways was in 1825; it started with the opening of the *Stockton and Darlington Railway* and used the aptly named *Locomotion No 1* steam engine [Kir02]. From 1825, a plenitude of railways opened, often referred to as the *golden age* of railways. It was not long (1928) before loss of life accidents started occurring, although many of these accidents were not catastrophic. They include incidents such as people falling out of overcrowded moving trains due to doors not being locked. The precise details and numbers of these incidents cannot be verified; this is because it was not until 1840 that the HMRI[2] was formed as a result of the *Railway Regulation Act 1840*. Initially HMRI was only tasked with the investigation of serious incidents, and to report back to Parliament; the rail companies were not required to notify the regulators of minor misdemeanours.

As an example of the type of incidents that HMRI dealt with, in 1840, there was a significant incident where 4 people lost their lives. This was due to railway staff not correctly fastening an iron casting in a goods carriage, attached directly behind the tender. The result was that an iron casting fell off the goods carriage, which caused the derailment of the passenger carriages behind the goods carriage [Smi40]. The result of this investigation was the recommendation of how to delegate responsibilities between different staff. Notably, in this case the *goods department* should have ensured the fastenings were fit for purpose.

Many of the early reports by HMRI describe bad practices with the railway staff and companies. These include gate-keepers at level crossings going into their houses and not closing the gate, or debris on the track. Other issues include hardware failures, a number of axle breaks, and a number of

[2]Initially part of the *Board of Trade* (BoT)

tunnel and bridge collapses were reported. These early issues were mitigated after a small number of years by better training of the staff, and engineers understanding the requirements of the railways. However, there was still an underlying issue of even the best trained human making signalling errors. One of the early reported cases of a signalling error was at Blue Pits, Lancashire, UK, where a head on collision occurred on a foggy night because of a set of points being wrongly set [Wyn49]. This was attributed to a combination of dense fog and the pointsman making a mistake.

These accidents and many more prompted a tremendous effort directed toward controlling the trains that run on the railways in a careful way so that they do not collide, derail or deadlock. This was important because a single accident on the railway has the potential of killing many people. Also, the financial value of trains and the railway infrastructure are considerable, and the reputation of a railway company is badly damaged when a train derails or worse. The control which prevents trains from colliding known as signalling.

In the early days of signalling, policemen were responsible for ensuring safety by using a system of coloured hand held flags (or in some cases hand gestures) during the day and oil lanterns by night. It was the lines running between stations that were of primary interest, i.e. there was a policeman at each station and junction signalling to the trains when it was safe to travel down the line. There was no communication between the policemen, and they simply relied upon a time interval system to prevent following trains from running into the train ahead. The policeman would show a red flag to following trains for 5 minutes after a train had passed him, and a green flag for a further 5 minutes. Only after more than 10 minutes had passed would a white flag be shown to approaching drivers. If a train stopped unexpectedly after passing a policeman and out of his sight, then the driver of a following train only had his own vision for warning. This obviously had serious ramifications for safety. The ongoing advents of the electric telegraph (circa 1837) helped to mitigate these issues, and laid the foundations for *block signalling*. This development involved providing a fail-safe communication medium between the policemen so that a train would only be allowed to enter a block of track, should the policeman at the other end have observed the previous train leaving it, and the poiceman has sent the *all clear* message back.

During the period 1830 to 1860, *block signalling* was developed; it was based upon the ideas of a policeman waving coloured flags to indicate that a section of track was clear or not. Block signalling divides the track up into disjoint blocks. Each of these blocks is protected by a semaphore signal; these are signals that can show a clear or danger aspect. There were still a number

of problems with this implementation of the block system, primarily, the signals and sets of points were individually controlled. No attempt was made to prevent unsafe combinations of signal aspects, such as with the opposing routes signalling principle in the previous chapter. Another problem with this implementation of the block system is that it was not clear where trains should stop when signalled to do so. Responsibility was delegated to the train drivers to select a safe location to stop.

With the growing demand, and hence complexity of the railway network, advents were required to simplify the signalman's job (who took over the policeman's responsibilities) and reduce mental strain. One such advent was when groups of related signals and sets of points had their controls juxtaposed. Typically each station or junction would have all its controls juxtaposed, the locations (buildings) containing the juxtaposed controls became known as a signal box. Inevitably, even the most well-trained signalman could make mistakes and configure the signals into unsafe scenarios. As an example of such a situation, in 1867 at Walton Junction, near Manchester, UK, a collision between a passenger train and coal train occurred due to signalling error. There were 8 fatalities and 73 injured [Yol67]. It is noted in the accident report that previously there had been a minor incident at this junction, and a recommendation previously made by Captain Tyler (from HMRI) "that the most improved apparatus should be applied for working the points and signals in connection with one another" was not adopted. Col. Yolland then remarks "if Captain Tyler's recommendations had been adopted, the recent collision at Walton junction certainly would not have happened." [Yol67].

The apparatus that Captain Tyler referred to is what is now known as an *interlocking system*. Traditionally these were apparatuses installed between the controls (levers) and the actual signals and sets of points. The apparatus ensures that the configurations of signals and sets of points are safe. Figure 9.1 shows the conceptual model of the interlocking system with respect to the signalman and railway.

Traditionally signal boxes (Figure 9.2) were buildings with a good vantage point of the railyard, and housed levers that controlled the signals and sets of points. The signal box also housed the interlocking system, typically on the floor below the levers. These mechanical interlocking systems were essentially *3D jigsaws*, with a number of rods and tappets. Each lever would be connected to a number of rods. When a lever is moved from its *normal* position into its *reverse* position, it moves its associated rods, if they are *free to move*. The design of this jigsaw was such that when certain levers were moved, others could not, Figure 9.2 shows this. These mechanical interlocking systems were specified by a document known as a *locking table*,

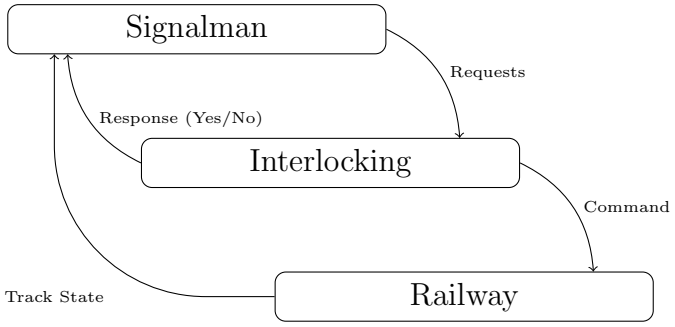


Figure 9.1: Conceptual Responsibility of Interlocking, the yes/no response was achieved by the lever being *free to move*.

see Section 11.1.2 for more information about these tables.

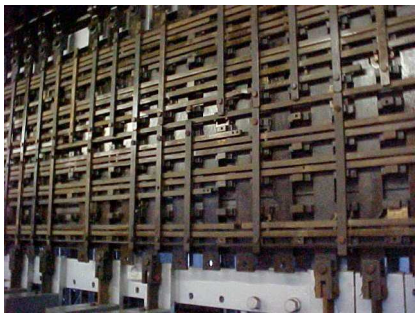


Figure 9.2: A traditional signal box (*left*) with the associated interlocking (*right*). Photographs courtesy of Invensys Rail.

As previously noted, the growing complexities of the railway network meant that the strain on the signalman's mental abilities—memory—was significant. The introduction of the interlocking system significantly aided the signalman in their duties. For this reason, in 1889 following one of the worst rail disasters in the UK, at Armagh, where there were 78 fatalities and 262 injuries, many of whom were children [Hut89], it became law that all passenger train services must be protected by interlocking systems. HMRI/BoT was given powers under *Regulation of Railways Act 1889* to regulate the railway companies in full, and ensure, among other things, that interlocking systems and facing-point locks were in-place for all passenger trains [Cal92].

9.1.1 Modern History

It is clear that these mechanical interlocking systems are limited by the equipment and materials. Coupled with the advent of valves, digital interlocking

systems were developed. This progressed through relays and finally to microprocessors. It is worth noting that currently on UK railways there are various types of interlocking systems in use. There are still signal boxes, such as the one in Figure 9.2, operating on quieter lines, albeit augmented with electromechanical locks to fulfil legal requirements. The rationale for keeping the mechanical interlocking systems is that they have worked for many years, and there is no real need to modify them, besides maintenance, as long as the railyard's topology is not modified, such as adding new signals or track. It is also an extremely costly business replacing an interlocking system.

The first digital interlocking system used in the UK was the *British Rail Solid State Interlocking* (SSI). The SSI project was established in 1976 [Cri87], first piloted at Leamington Spa, and was specified using the formal language Z [Kin94]. This was a great success and paved the way for subsequent interlocking systems. A significant advantage of digital interlocking systems is that they take as inputs not only the requests, but also the state of the topology. Typically the digital interlocking will detect

- occupied track segments,
- positions and locks of sets of points,
- blown light bulbs (signal proving), and
- emergency situations.

Figure 9.1 then becomes:

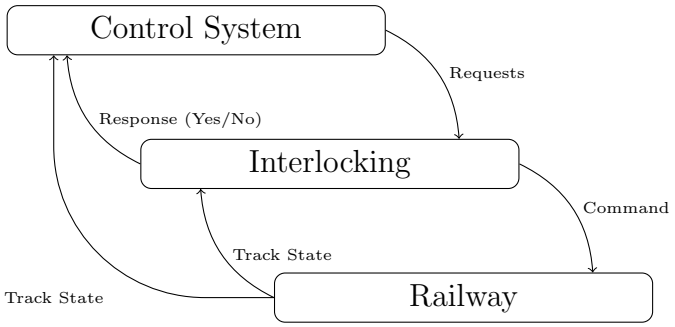


Figure 9.3: Conceptual Responsibility of Digital Interlocking.

One digital interlocking that has been studied extensively during this project is the Westrace. The Westrace is a newer, and simpler (lower-level) interlocking design than the SSI and is programmed using ladder logic.

Figure 9.4: Westrace interlocking system. Photograph courtesy of Invensys Rail.

9.2 Ladder Logic

Interlocking systems are realised using a multitude of techniques; systems programmed using ladder logic are the focus of this research. Ladder logic (IEC 61131-3) is a discrete time, linear system of Boolean valued equations, relating Boolean valued inputs and internal state to Boolean valued outputs. Originally developed in the electrical engineering discipline, ladder logic is a graphical representation of a reactive logic circuit consisting of a sequence of Boolean valued assignments to latches.

As an example ladder logic program, consider the program in Figure 1.2 that realised a toy Pelicon crossing control system. The semantics of these diagrams is as follows: each assignment (or *rung*) assigns to the variable on the right in rounded brackets the result of (propositionally) evaluating the circuit on its left. These circuits are built out of the operators $\wedge$, $\vee$ and $\neg$; and a global set of variable identifiers. Consider Figure 9.5, rung 1 depicts the assignment $c := a \wedge \neg b$ and rung 2 depicts $d := a \vee c$.

The assignments are partitioned into two groups, assignments to output variables and assignments to internal variables; input variables obtain their values from observations about the real world and are not assigned values

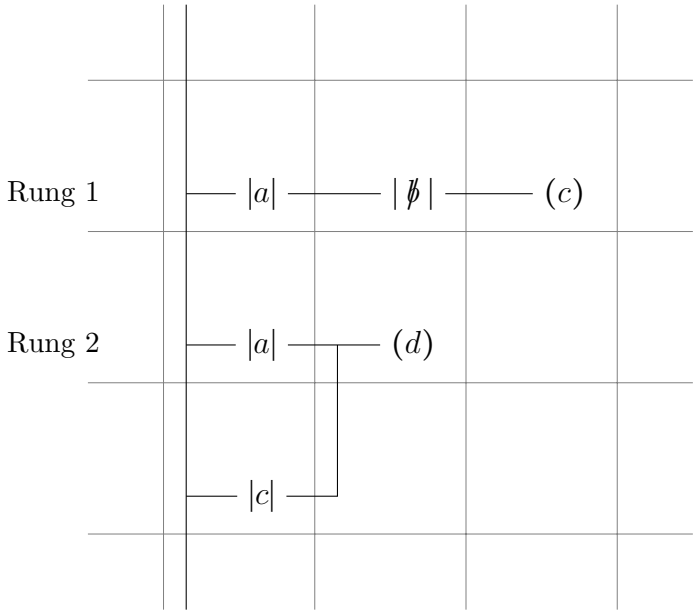


Figure 9.5: An example ladder logic diagram that depicts two rungs. Rung 1 is the assignment: $c \coloneqq a \wedge \neg b$, and rung 2 is the assignment: $d \coloneqq a \vee b$. These assignments are sequential, i.e. earlier assignments affect later ones.

Remark

In this project these diagrams were translated into an equivalent propositional formula by reusing work from the author's masters thesis [Kan08].

by the ladder, but instead contribute to the value of the internal and output variables. Conceptually, a ladder logic program with n rungs and m input variables is the following imperative program:

$$
\begin{array}{l}
\mathit{Initialise}(a_0, \ldots, a_n) \\
\textbf{while } \text{true } \{ \\
\quad \mathit{ReadInputs}(b_0, \ldots, b_m) \\
\quad a_0 \coloneqq \varphi_0 \\
\qquad \vdots \\
\quad a_n \coloneqq \varphi_n \\
\quad \mathit{WriteOutputs}(a_0, \ldots, a_n) \\
\}
\end{array}
$$

Here a_i are state/output variables, b_j are input variables, and φ_k are propositional formulæ built from conjunctions, disjunctions and the propositions

$\{a_0, \ldots, a_n, b_0, \ldots, b_m, \neg a_0, \ldots, \neg a_n, \neg b_0, \ldots, \neg b_m\}$. The loop iterates infinitely many times; at the end of each iteration after all the assignments, when the outputs are written, correctness needs to be shown, see Section 10.1.

It was found that differentiating between the internal and output variables was not required and resulted in two issues: first, the correctness properties are unable to reference the internal variables (by definition), and secondly when formalising and proving properties about ladder logic in Agda, all the definitions are required to carry this extra information, e.g. Boolean formulæ, safety properties, lemmata and generation of problem sets for the external tool. Thus in the following, only state variables are considered which consist of the union of internal and output variable sets.

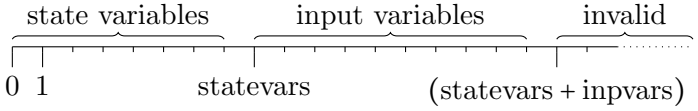
In Agda ladder logic programs are represented by the number of input and state variables, an initial state and a list of assignments between state variables and Boolean formulæ. Ladder logic programs are defined in module `Ladder.Core` of Appendix F as follows:

```
record Ladder : Set where
  constructor
    ladder
  field
    statevars     : ℕ
    inpvars       : ℕ
    rungs         : List (ℕ × BooleanFormula)
    initialstate  : List (ℕ × Bool)
    inp-correct   : BooleanFormula
```

where

$$\text{BooleanFormula} := \{\text{true}, \text{false}, \_\wedge\_, \_\Rightarrow\_, \_\vee\_, \neg\_, \text{var}\_\}$$

Although the connective $\_\Rightarrow\_$ is not used in a ladder, it is included to simplify later definitions, e.g. formalising verification conditions. The variables are indexed by $\mathbb{N}$. A variable with index i is a state variable if $i < \text{statevars}$, and an input variable if $\text{inpvars} \leq i < (\text{statevars} + \text{inpvars})$, otherwise it is badly formed. Badly formed variables are considered later. That is the following number line is observed:



The motivation behind this simple typed definition of a ladder is: usability over a clean mathematical definition. For instance, the rungs and initial state

assignments contain the index of the variable being assigned a value, instead of using their position in the list to determine their indices. Therefore, the variable indices do not need to coincide with the order of the rungs or order of the initial state assignments, which results in an intuitive and easy definition to validate. For example, a ladder with the initial state

$$[(n_1, b_1), \ldots, (n_k, b_k)]$$

expresses that variable n_i is initialised to value b_i, and all variables not mentioned are initialised to false. A ladder with the following rungs

$$[(m_1, \varphi_1), \ldots, (m_j, \varphi_j)]$$

expresses the following assignments, where each φ_i depends upon the assigned values of m_j for $j < i$, and unassigned values of m_j for $j \geq i$.

$$a_{n_i} := \varphi_i$$

Subsequently, the well-formedness of a ladder will be defined by decidable predicates. This well-formedness will include ruling out variables with invalid indices, and rungs/initial states that assign duplicate values to variables. However, first the formula *inp-correct* is considered; it expresses an invariant on the input variables. This invariant is required due to the fact that a number of combinations of the input variables might not be physically possible. For instance, a 3-way switch that has a contact for each of the 3 positions should never make contact with more than one of these contacts at a time. In essence, *inp-correct* expresses domain knowledge specific to the context of the ladder program.

As an example of the formalisation of a ladder program, below is an instance that represents the ladder in Figure 9.5.

$$\begin{array}{lll} \text{record } \{ & & \\ \quad \mathit{statevars} & = & 2 \qquad\qquad\qquad\qquad\quad ; \\ \quad \mathit{inpvars} & = & 2 \qquad\qquad\qquad\qquad\quad ; \\ \quad \mathit{rungs} & = & [(\hat{c}, \hat{a} \wedge \neg \hat{b}), (\hat{d}, \hat{a} \vee \hat{c})] \; ; \\ \quad \mathit{initialstate} & = & [(\hat{c}, \text{true})] \qquad\qquad\quad ; \\ \quad \mathit{inp\text{-}correct} & = & \text{true} \qquad\qquad\qquad\quad \} \end{array}$$

where $\hat{a} = 0$, $\hat{b} = 1$, $\hat{c} = 3$ and $\hat{d} = 2$. Here d gets the initial value of false as it is not explicitly defined, and the indices of c and d are not required to match the execution order of the assignments.

A state of the ladder l, denoted by State_l, is defined as

$$\text{State}_l = (s : \text{List Bool}) \times (\text{length } s \equiv \text{statevars}_l)$$

and an input, denoted by Input_l, is of the type

$$\text{Input}_l = (i : \text{List Bool}) \times (\text{length } i \equiv \text{inputvars}_l) \times [\![\text{ inp-correct}_l \text{ }]\!]_i$$

where $[\![\_]\!]_\_$ has the same definition from Section 4.1 with the exception that the environments have the type $(\mathbb{N} \to \text{Bool})$. It is often convenient to represent these environments by lists (or lists paired with proofs); in which cases the lists are implicitly lifted to functions by assuming a default value of false for all indices that fall outside the bound of the list, and/or any relevant projections. That is, for a list x given as an environment, the following map is used

$$n \mapsto \begin{cases} x[n] & \text{if } n < \text{length } x \\ \text{false} & \text{otherwise} \end{cases}$$

In Section 9.3 these ladders are translated into transition systems. For now it is noted that l is well-formed *iff* it fulfils the following axioms:

$$\forall i \,.\, i < (\text{length initialstate}_l) \to \pi_0\,(\text{initialstate}_l[i]) < \text{statevars}_l \quad (\textit{init-map}<)$$

$$\begin{aligned} &\forall i\ j \,.\, i < j \wedge j < (\text{length initialstate}_l) \qquad (\textit{init-map}\neq) \\ &\qquad \to \pi_0\,(\text{initialstate}_l[i]) \not\equiv \pi_0\,(\text{initialstate}_l[j]) \end{aligned}$$

$$\forall i \,.\, i < (\text{length rungs}_l) \to \pi_0\,(\text{rungs}_l[i]) < \text{statevars}_l \qquad (\textit{trans-map}<)$$

$$\begin{aligned} &\forall i\ j \,.\, i < j \wedge j < (\text{length rungs}_l) \qquad (\textit{trans-map}\neq) \\ &\qquad \to \pi_0\,(\text{rungs}_l[i]) \not\equiv \pi_0\,(\text{rungs}_l[j]) \end{aligned}$$

These four axioms ensure that the initial state and rungs that are given by lists of pairs form partial finite maps. Before defining the final axioms, it is required to introduce a function

$$\text{bound} : (n : \mathbb{N}) \to (\varphi : \text{BooleanFormula}) \to \text{Bool}$$

that returns true, *iff* all variables in φ are less than n. The penultimate axiom ensures that all variable indices are within the correct range

$$\begin{aligned} &\forall i \,.\, i < (\text{length rungs}_l) \qquad (\textit{rung-bound}) \\ &\qquad \to \text{bound}\,(\text{statevars}_l + \text{inputvars}_l)\,(\pi_1\,(\text{rungs}_l[i])) \end{aligned}$$

and the final axiom ensures that the input invariant only references variables in the input

$$\text{bound inputvars}_l \text{ inp-correct}_l \qquad (\textit{invar-bound})$$

Remark

These axioms are required as the definition of a ladder is simply-typed, this decision was taken to improve performance and readability in Agda. For instance, had finite numbers been applied here, then Agda would have required significant amounts of time and space for type-checking large ladders. This is because there is no primitive implementation for finite numbers. With the definitions outlined in this section, the natural numbers are built-in, so are the comparisons on them. Therefore, determining whether a ladder is well-formed is in a large part is performed with machine/CPU integers and operations on these.

9.3 Decidable Ladder

Defining the nthState function (required in Chapter 8) requires a decidable transition system. These are obtained by proving that a decidable transition system follows by a well-formed ladder, which is the subject of this section.

In this section, initially basic transition systems are defined, and a translation from ladder logic into these systems is given. Then these transition systems are shown to be decidable, if the ladder was well-formed.

The definition of a transition system is canonical in Agda. They are defined in module `TransitionSystem` of Appendix F by a set of states, set of inputs/actions, a transition relation between the states and a set of initial states as follows:

```
record TransitionSystem (Input : Set) : Set₁ where
  constructor
    ts
  field
    State      : Set
    Initial    : State → Set
    Transition : State → Input → State → Set
```

To translate a ladder logic program l into these generic transition systems, the types of the inputs and states are given by Input_l and State_l, respectively. To define the initial state relation, first a Boolean formula is given that holds *iff* the given state is as defined by initialstate_l. Intuitively the formula is

defined by the following map

$$\left\{\begin{array}{lll} [] & \mapsto & \text{true}\,; \\ [(n_1,b_1),(n_2,b_2),\ldots,(n_m,b_m)] & \mapsto & (\text{var}\,n_1 \Leftrightarrow \hat{b}_1) \wedge \\ & & \quad (\text{var}\,n_2 \Leftrightarrow \hat{b}_2) \wedge \\ & & \quad\quad \vdots \\ & & \quad (\text{var}\,n_m \Leftrightarrow \hat{b}_m) \end{array}\right\}$$

where $\hat{b}_i$ is b_i injected into BooleanFormula. The formula is generated by the following function mkinit; it is defined in module `Ladder.Core` of Appendix F.

$$\begin{array}{lll} \text{mkinit} : \text{List}\,(\mathbb{N} \times \text{Bool}) \to \text{BooleanFormula} & & \\ \text{mkinit}\,[\,] & = & \text{true} \\ \text{mkinit}\,((a,b) :: as) & = & (\text{var}\ a \Leftrightarrow \hat{b}) \wedge \text{mkinit}\ as \end{array}$$

The initial relation on states is then given as follows:

$$\lambda s\,.\,[\![\,\text{mkinit}\ \text{initialstate}_l\,]\!]_s$$

The transition relation is more complicated as it formalises the assignments; thus it introduces new variables. These introduced variables, due to notation, will be referred to in the following as *primed* variables; they represent the variable after it has been assigned a value by a rung. At any intermediate point, while evaluating the rungs of the ladder, only the un-primed or primed version of the variable being assigned the result of the rung is accessible, this is due to the sequential nature of the evaluation. For instance before the rung

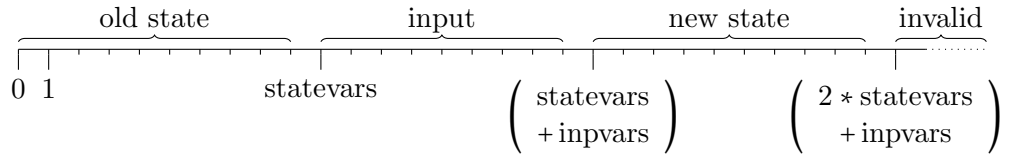
$$x := \varphi$$

is evaluated, only x is accessible, after its evaluation only x' is accessible. For this reason, while translating the rungs into a Boolean formula, a list of visited/processed state variables is built-up. The function mktrans produces this formula from a list of processed rungs, and a list of unprocessed rungs. Before formally defining this function, the transition formula is intuitively given by the following map between a list of rungs and a Boolean formula.

$$\left\{\begin{array}{lll} [] & \mapsto & \text{true}\,; \\ [(n_1,\varphi_1), & & (\text{var}\,n'_1 \Leftrightarrow \varphi_1) \wedge \\ \ (n_2,\varphi_2), & \mapsto & \quad (\text{var}\,n'_2 \Leftrightarrow \varphi_2[n_1 := n'_1]) \wedge \\ \quad \vdots & & \quad\quad \vdots \\ \ (n_m,\varphi_m)] & & \quad (\text{var}\,n_m \Leftrightarrow \varphi_m[n_1 := n'_1] \ldots [n_{m-1} := n'_{m-1}]) \end{array}\right\}$$

where the primed index $n' = n + \text{statevars}_l + \text{inputvars}_l$; if the ladder is well-formed, then these primed indices are fresh. Here $x[t := s]$ means substitute

all occurrences of t in x by s. The resulting formula references variables in a pair of states and an input; it characterises a valid transition of the ladder. Thus, the number line of the variable indices is adjusted as follows:



The compound substitutions above are defined by the following function that is indexed by the size ($\mathbb{N}$) of the shift, and the substitution depends upon a list of visited rungs.

$$\begin{aligned}
&\mathrm{shift}'_n : \mathrm{List}\ \mathbb{N} \to \mathbb{N} \to \mathbb{N} \\
&\mathrm{shift}'_n \quad [\,] \quad a \;=\; a \\
&\mathrm{shift}'_n\ (x :: xs)\ a \;=\; \begin{cases} x + n & \textbf{if } x = a \\ \mathrm{shift}'_n\ xs\ a & \textbf{otherwise} \end{cases}
\end{aligned}$$

This definition is trivially lifted to Boolean formulæ by applying it to each variable, this lifting results in the following signature:

$$\mathrm{shift}_n : \mathrm{List}\ \mathbb{N} \to \mathrm{BooleanFormula} \to \mathrm{BooleanFormula}$$

Lemma 9.3.1.

$$\forall \varphi\ \xi\ n\ xs\ .\ [\![\ \mathrm{shift}_n\ xs\ \varphi\ ]\!]_\xi \equiv [\![\ \varphi\ ]\!]_{\xi \circ (\mathrm{shift}'_n\ xs)}$$

Proof. By induction on φ. Without loss of generality $\varphi = \mathrm{var}\ a$, as the proof in the other cases follows by the induction hypothesis. It remains to show

$$[\![\ \mathrm{shift}_n\ xs\ (\mathrm{var}\ a)\ ]\!]_\xi \equiv [\![\ \mathrm{var}\ a\ ]\!]_{\xi \circ (\mathrm{shift}'_n\ xs)}$$

which follows by definition

$$\begin{aligned}
[\![\ \mathrm{shift}_n\ xs\ (\mathrm{var}\ a)\ ]\!]_\xi &\equiv [\![\ \mathrm{var}\ (\mathrm{shift}'_n\ xs\ a)\ ]\!]_\xi \\
&\equiv \mathrm{T}\ (\xi\ (\mathrm{shift}'_n\ xs\ a)) \\
&\equiv [\![\ \mathrm{var}\ a\ ]\!]_{\xi \circ (\mathrm{shift}'_n\ xs)}
\end{aligned}$$

□

Lemma 9.3.2 (shift elimination)**.** *Assume a list of naturals r, and two naturals k and n. Then*

$$(\exists i\ .\ r[i] \equiv k) \wedge (\mathrm{shift}'_n\ r\ k \equiv n + k)$$

or

$$(\forall i\ .\ r[i] \not\equiv k) \wedge (\mathrm{shift}'_n\ r\ k \equiv k)$$

Proof. Follows by simple induction on r, and decidable equality on naturals. □

The function mktrans is also indexed by the shift amount and is defined as follows:

$$\begin{aligned}
&\text{mktrans}_n : \text{List}\ (\mathbb{N} \times \text{BooleanFormula}) \\
&\qquad \to \text{List}\ (\mathbb{N} \times \text{BooleanFormula}) \\
&\qquad \to \text{BooleanFormula} \\
&\text{mktrans}_n\ v\ [\,] \quad = \quad \text{true} \\
&\text{mktrans}_n\ v\ ((a, \varphi) :: r) \quad = \quad (\text{var}\ (a + n) \Leftrightarrow \text{shift}_n\ v'\ \varphi) \\
&\qquad\qquad \wedge\ \text{mktrans}_n\ (v +\!\!+ [(a, \varphi)])\ r
\end{aligned}$$

where v' is a list of the indices extracted from v. The accumulated list could be simplified by only storing the indices as they are not used here, however, due to use later, it is not simplified. The transition relation is then defined as follows:

$$\lambda s\ i\ s' .\ [\![\ \text{mktrans}\ [\,]\ \text{rungs}_l\ ]\!]_{(s +\!\!+ i +\!\!+ s')}$$

The above definitions will translate a ladder logic program into a transition system. The following function defines this translation:

$$\begin{aligned}
&\text{mkts} : (l : \text{Ladder}) \to \text{TransitionSystem}\ \text{Input}_l \\
&\text{mkts}\ l \quad = \quad \text{ts}\ \text{State}_l \\
&\qquad\qquad (\forall s .\ [\![\ \text{mkinit}\ \text{initialstate}_l\ ]\!]_s) \\
&\qquad\qquad (\forall s\ i\ s' .\ [\![\ \text{mktrans}\ [\,]\ \text{rungs}_l\ ]\!]_{(s +\!\!+ i +\!\!+ s')})
\end{aligned}$$

Remark

In the proofs to follow, for a ladder l the expressions $\text{Initial}_{(\text{mkts}\, l)}$ and $\text{Transition}_{(\text{mkts}\, l)}$ are defined by the function mkts.

Decidable Transition System. It still remains to show that these translated transition systems are decidable, if the ladder was well-formed. In this context decidable means that the next step of the transition system is computable from a given state and input, and that the next step is permitted by the transition relation. To show this, first, for a transition system ts the definition of a decidable transition system is introduced in module

`TransitionSystem.Decidable` of Appendix F as:

$$
\begin{array}{l}
\text{record DecidableTransitionSystem}_{ts} : \text{Set where} \\
\quad \text{field} \\
\qquad \begin{array}{ll}
\text{initialState} & : \text{State}_{ts} \\
\text{initialCorrect} & : \text{Initial}_{ts}\ \text{initialState} \\
\text{transitionFunction} & : \text{State}_{ts} \to \text{Input}_{ts} \to \text{State}_{ts} \\
\text{transitionCorrect} & : \forall\, s\ i\ .\ \text{Transition}_{ts}\ s\ i\ (\text{transitionFunction}\ s\ i)
\end{array}
\end{array}
$$

where Input_{ts} is the set of inputs that the transition system was built over. From this definition, it is a trivial matter to define a function

$$\text{nthState}_{dts} : (\mathit{inputs} : \mathbb{N} \to \text{Input}_{ts}) \to (n : \mathbb{N}) \to \text{State}_{ts}$$

that for a given stream of inputs, produces a stream of states. It calculates the n^{th} state by starting from the initial/0^{th} state, and applying the transition function n times, together with the associated first $n-1$ inputs.

It is now shown that a decidable transition system (dts) follows by a well-formed ladder.

Initial State. The initial state of the dts is canonically constructed from the initial state (of type List $(\mathbb{N} \times \text{Bool})$) defined in the ladder, which is possible because (*init-map*$\neq$) ensures that each of the variable indices in the list is unique. Therefore, the uniqueness defines a partial map. This map is made total by assigning a default value, of false, to the unmapped elements in the domain. The total map is constructed by the function

$$
\begin{array}{l}
\text{lookup} : \forall\{A\}\ .\ \text{List}\ (\mathbb{N} \times A) \to A \to \mathbb{N} \to A \\
\text{lookup}\ [] \qquad\quad d\ n\ =\ d \\
\text{lookup}\ ((i, x) :: l)\ d\ n\ =\ \begin{cases} x & \textbf{if}\ i \equiv n \\ \text{lookup}\ l\ d\ n & \textbf{otherwise} \end{cases}
\end{array}
$$

which takes a partial map, default value and produces a total function. As (*init-map*$<$) holds, all the indices are bounded by statevars_l. The initial state of the dts is then a list of length statevars_l defined point-wise, from the initial state in the ladder.

$$
\begin{array}{l}
\text{mkInitialState} : \forall l\ .\ \text{State}_l \\
\text{mkInitialState}\ l\ =\ \text{lookup}\ \text{initialstate}_l\ \text{false}
\end{array}
$$

See module `Ladder.Decidable` in Appendix F for the technical details.

Theorem 9.3.3 shows that the initial state, as computed above, is correct provided the ladder is well-formed. In the following, the lemmata are presented top-down to improve readability.

Theorem 9.3.3 (*initialCorrect*)**.** *Assume a ladder l and (init-map≠, see page 195) holds, then*

$$\text{Initial}_{(\text{mkts}\, l)}\ (\text{mkInitialState}\, l)$$

holds.

Proof.

$$\begin{aligned}\text{Initial}_{(\text{mkts}\, l)}\ (\text{mkInitialState}\, l) &\equiv [\![\ \text{mkinit}\ \text{initialstate}_l\ ]\!]_{(\text{mkInitialState}\, l)} \\ &\qquad\qquad \textit{by definition} \\ &\equiv [\![\ \text{mkinit}\ \text{initialstate}_l\ ]\!]_{(\text{lookup}\ \text{initialstate}_l\ \text{false})} \\ &\qquad\qquad \textit{by definition}\end{aligned}$$

The proof then follows by Lemma 9.3.4 applied to (*init-map≠*). □

Lemma 9.3.4 (mkinit)**.** *Assume a list* $l : \text{List}\ (\mathbb{N} \times \text{Bool})$ *such that*

$$\forall i\ j\ .\ i < j \wedge j < (\text{length}\ l) \rightarrow\ \pi_0\ (l[i]) \neq \pi_0\ (l[j])$$

holds, then

$$[\![\ \text{mkinit}\ l\ ]\!]_{(\text{lookup}\, l\ \text{false})}$$

holds.

Proof. By induction on l.

Case $l = []$**:** Trivial as mkinit $[] \equiv$ true.

Case $l = (n, b) :: l'$**:** There are two obligations to prove, the first is

$$[\![\ \text{var}\ n \Leftrightarrow \hat{b}\ ]\!]_{(\text{lookup}\ ((n,b)::l')\ \text{false})}$$

which unfolds to $\text{T}\ b \Leftrightarrow \text{T}\ b$, and follows trivially.

The second obligation is

$$[\![\ \text{mkinit}\ l'\ ]\!]_{(\text{lookup}\ ((n,b)::l')\ \text{false})}$$

which by Lemma 9.3.5 is equivalent to

$$[\![\ \text{mkinit}\ l'\ ]\!]_{(\text{lookup}\, l'\ \text{false})}$$

and holds by induction hypothesis.

□

Lemma 9.3.5 (lookup-fv)**.** *For all* $a : \mathbb{N}$ *and* b : Bool*. Assume a list* $l : \text{List}\ (\mathbb{N} \times \text{Bool})$ *and* $x \in FV(\text{mkinit}\ l)$*, such that*

$$\forall j\ .\ j < \text{length}\ l \to a \neq \pi_0(l[j])$$

holds, then

$$\text{lookup}\ l\ \text{false}\ x \equiv \text{lookup}\ ((a, b) :: l)\ \text{false}\ x$$

holds.

Proof. Assume $x = a$. By Lemma 9.3.6

$$\exists j\ .\ j < \text{length}\ l \wedge \pi_0(l[j]) = a$$

A contradiction, therefore $x \neq a$. The proof follows by rewriting obligation by $x \neq a$. □

Lemma 9.3.6 (init-fv)**.** *Assume* $l : \text{List}\ (\mathbb{N} \times \text{Bool})$ *and* $x \in FV(\text{mkinit}\ l)$*. Then there exists a* $j < \text{length}\ l$ *such that* $x = \pi_0(l[j])$*.*

Proof. Induction on l. When $l = [\,]$ there is no such x, therefore assume $l = ((a, b) :: l')$. As $x \in FV(\text{mkinit}\ l)$ there are two cases to consider.

case $x \in FV(\text{var}\ a \Leftrightarrow \hat{b})$**:** Therefore $j = 0$ and $x = a$.

case $x \in FV(\text{mkinit}\ l')$**:** Proof follows by induction hypothesis.

□

Transition Function. Defining the transition function requires that an evaluation of Boolean formulæ under an environment is given. Assume a function

$$\text{eval} : \text{BooleanFormula} \to (\mathbb{N} \to \text{Bool}) \to \text{Bool}$$

with canonical semantics, such that

$$\forall \varphi\ \xi\ .\ \mathrm{T}\ (\text{eval}\ \varphi\ \xi) \leftrightarrow [\![\ \varphi\ ]\!]_\xi \qquad (\textit{eval-correct})$$

holds. The value of eval is then used to evaluate each rung under the correct environment. The environment is updated after each rung is evaluated to reflect the assignment. After all the rungs have been evaluated, the resulting environment is the next state. This gives rise to the function

$$\begin{aligned}
&\text{executeLadder} : \text{List}\ (\mathbb{N} \times \text{BooleanFormula}) \to \text{State}_l \to \text{Input}_l \to \text{State}_l \\
&\text{executeLadder}\quad [\,]\quad s\ i\ =\ s \\
&\text{executeLadder}\ ((a, \varphi) :: r)\ s\ i\ =\ \text{executeLadder}\ r\ (s[a := \text{eval}\ \varphi\ (s +\!\!+ i)])\ i
\end{aligned}$$

where $x[i := y]$ behaves like x, except the i^{th} element is assigned y. In the following three lemmata, properties regarding executeLadder are shown; these will be required in subsequent proofs to follow about the correctness of the transition function. See module `Ladder.Decidable` in Appendix F for the complete definition of all definitions in this sub-section.

Lemma 9.3.7 (executeLadder)**.** *Assume lists of rungs r_1 and r_2, lists of Booleans ξ and ζ, then the following holds.*

$$\text{executeLadder}\ (r_1 +\!\!+ r_2)\ \xi\ \zeta \equiv \text{executeLadder}\ r_2\ (\text{executeLadder}\ r_1\ \xi\ \zeta)\ \zeta$$

Proof. Follows by induction on r_1. □

Lemma 9.3.8 (executeLadder2)**.** *Assume list of rungs r, lists of Booleans ξ and ζ, and $k : \mathbb{N}$ such that*

$$\forall i\ .\ i < \text{length}\ r \to \pi_0(r[i]) \not\equiv k$$

Then the following holds

$$(\text{executeLadder}\ r\ \xi\ \zeta)[k] \equiv\ \xi[k]$$

Proof. By induction on r, when $r = []$ proof follows by definition. Therefore assume $r = (a, \varphi) :: r'$.

$$\begin{aligned}
&(\text{executeLadder}\ ((a,\varphi) :: r')\ \xi\ \zeta)[k] \\
&\qquad \equiv (\text{executeLadder}\ r'\ (\xi[a := \text{eval}\ \varphi\ (\xi +\!\!+ \zeta)])\ \zeta)[k] \\
&\qquad\qquad \textit{by definition} \\
&\qquad \equiv \xi[a := \text{eval}\ \varphi\ (\xi +\!\!+ \zeta)][k] \\
&\qquad\qquad \textit{by induction hypothesis} \\
&\qquad \equiv \xi[k] \\
&\qquad\qquad \textit{since } a \neq k
\end{aligned}$$

□

Lemma 9.3.9 (executeLadder3)**.** *Assume a list of rungs r, two lists of Booleans ξ and ζ, a formula φ and an $a : \mathbb{N}$. Such that*

$$a < \text{length}\ \xi$$

Then the following holds:

$$\text{eval}\ \varphi\ ((\text{executeLadder}\ r\ \xi\ \zeta) +\!\!+ \zeta) \equiv (\text{executeLadder}\ (r +\!\!+ [(a, \varphi)])\ \xi\ \zeta)[a]$$

Proof.

$$
\begin{aligned}
&(\text{executeLadder}\ (r +\!\!+ [(a, \varphi)])\ \xi\ \zeta)[a] \\
&\quad \equiv (\text{executeLadder}\ [(a, \varphi)]\ (\text{executeLadder}\ r\ \xi\ \zeta)\ \zeta)[a] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by Lemma 9.3.7} \\
&\quad \equiv (\text{executeLadder}\ r\ \xi\ \zeta)[a := \text{eval}\ \varphi\ ((\text{executeLadder}\ r\ \xi\ \zeta) +\!\!+ \zeta)][a] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition} \\
&\quad \equiv \text{eval}\ \varphi\ ((\text{executeLadder}\ r_1\ \xi\ \zeta) +\!\!+ \zeta) \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{since } a < \text{length}\ s
\end{aligned}
$$

□

When the function executeLadder is applied to all rungs of a ladder, it yields a decidable transition function.

$$
\begin{aligned}
&\text{mkTransitionFunction} : \forall l\ .\ \text{State}_l \to \text{Input}_l \to \text{State}_l \\
&\text{mkTransitionFunction}\ l\ =\ \text{executeLadder}\ \text{rungs}_l
\end{aligned}
$$

Theorem 9.3.10 shows that the transition function defined above is correct with respect to the transition relation. The following lemmata are presented top-down to aid readability.

Theorem 9.3.10 (*transitionCorrect*)**.** *Assume a ladder l such that (trans-map<, see page 195), (trans-map≠) and (rung-bound) hold. Then*

$$\forall s\ i\ .\ \text{Transition}_{(\text{mkts}\ l)}\ s\ i\ (\text{mkTransitionFunction}\ l\ s\ i)$$

holds.

Proof.

$$
\begin{aligned}
&\forall s\ i\ .\ \text{Transition}_{(\text{mkts}\ l)}\ s\ i\ (\text{mkTransitionFunction}\ l\ s\ i) \\
&\qquad \equiv \forall s\ i\ .\ [\![\ \text{mktrans}\ []\ \text{rungs}_l\ ]\!]_{(s +\!\!+ i +\!\!+ (\text{mkTransitionFunction}\ l\ s\ i))} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition} \\
&\qquad \equiv \forall s\ i\ .\ [\![\ \text{mktrans}\ []\ \text{rungs}_l\ ]\!]_{(s +\!\!+ i +\!\!+ (\text{executeLadder}\ \text{rungs}_l\ s\ i))} \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \textit{by definition}
\end{aligned}
$$

The proof follows trivially by Lemma 9.3.11. □

Lemma 9.3.11. *Assume a ladder l and its induced transition system ts, and assume axioms (trans-map<), (trans-map≠) and (rung-bound). Partition its rungs into two lists r_1 and r_2, such that*

$$r_1 +\!\!+ r_2 \equiv \text{rungs}_l$$

Then the following holds

$$\forall s : \text{State}_l \; \forall i : \text{Input}_l \; . \; [\![\text{ mktrans } r_1 \; r_2 \;]\!]_{(s +\!\!+ i +\!\!+ (\text{executeLadder } (r_1 +\!\!+ r_2) \; s \; i))}$$

Proof. By induction on r_2

Case $[]$**:** Trivial as $\text{mktrans } r_1 \; [] = \text{true}$.

Case $(a, \varphi) :: r_2'$**:** Let

$$\xi = s +\!\!+ i +\!\!+ (\text{executeLadder } (r_1 +\!\!+ [(a, \varphi)] +\!\!+ r_2') \; s \; i)$$

and

$$n = \text{statevars}_l + \text{inputvars}_l$$

and

$$a' = a + n$$

Unfolding the obligation to prove yields

$$[\![\text{ var } a' \Leftrightarrow \text{shift}_n \; r_1 \; \varphi \;]\!]_\xi \wedge [\![\text{ mktrans } (r_1 +\!\!+ [(a, \varphi)]) \; r_2' \;]\!]_\xi$$

The right-hand conjunct follows by the induction hypothesis and associativity of $+\!\!+$. The left-hand conjunct is further expanded to

$$\text{T } (\xi \; a') \leftrightarrow [\![\text{ shift}_n \; r_1 \; \varphi \;]\!]_\xi \qquad (*)$$

Let

$$\xi' = (\text{executeLadder } r_1 \; s \; i) +\!\!+ i$$

By axiom (*eval-correct*) a proof of

$$\text{T } (\text{eval } \varphi \; \xi') \leftrightarrow [\![\varphi \;]\!]_{\xi'}$$

is obtained. $(*)$ follows by (*eval-correct*) after the following equality reasoning.

$$[\![\varphi \;]\!]_{\xi'} \equiv [\![\varphi \;]\!]_{\xi \circ (\text{shift}_n' \; r_1)}$$

by Lemma 9.3.12

$$\equiv [\![\text{ shift}_n \; r_1 \; \varphi \;]\!]_\xi$$

by Lemma 9.3.1

and

$$\begin{aligned}\text{eval } \varphi\ \xi' &\equiv \text{executeLadder } (r_1 +\!\!+ [(a, \varphi)])\ s\ i\ a \\ &\qquad \textit{by Lemma 9.3.9} \\ &\equiv \text{executeLadder } (r_1 +\!\!+ [(a, \varphi)] +\!\!+ r_2')\ s\ i\ a \\ &\qquad \textit{by Lemma 9.3.8 and Lemma 9.3.7} \\ &\equiv \xi\ a' \\ &\qquad \textit{by Lemma 9.3.13}\end{aligned}$$

□

Lemma 9.3.12 (executeLadder-shift)**.** *Assume a list of rungs r, and two list of Booleans s and t, and $n : \mathbb{N}$ such that*

$$r = r_1 +\!\!+ ((a, \varphi) :: r_2)$$

and

$$n = \text{length } (s +\!\!+ t)$$

and

$$\text{bound } \varphi\ n$$

and

$$\forall i\ .\ i < \text{length } r \to \pi_0(r[i]) < \text{length } s$$

and

$$\forall i\ j\ .\ i < j \wedge j < \text{length } r \to r[i] \neq r[j]$$

Then the following holds

$$[\![\varphi]\!]_{((\text{executeLadder } r_1\ s\ i) +\!\!+ i)} \equiv [\![\varphi]\!]_{(s +\!\!+ i +\!\!+ (\text{executeLadder } r\ s\ i)) \circ (\text{shift}'_n\ r_1)}$$

Proof. By induction on φ. Assume that $\varphi = \text{var } x$, since the other cases follow by the induction hypothesis. Therefore the obligation to prove is

$$((\text{executeLadder } r_1\ s\ i) +\!\!+ i)[x] \equiv (s +\!\!+ i +\!\!+ (\text{executeLadder } r\ s\ i))[\text{shift}'_n\ r_1\ x]$$

By Lemma 9.3.2 there are two cases to consider.

case $\exists i \,.\, \pi_0(r_1[i]) \equiv x$**:**

$$
\begin{aligned}
&(s +\!\!+ i +\!\!+ (\text{executeLadder } r\ s\ i))[\text{shift}'_n\ r_1\ x] \\
&\quad \equiv (s +\!\!+ i +\!\!+ (\text{executeLadder } r\ s\ i))[n + x] \\
&\qquad\qquad \textit{by Lemma 9.3.2} \\
&\quad \equiv (\text{executeLadder } r\ s\ i)[x] \\
&\qquad\qquad \textit{by Lemma 9.3.13} \\
&\quad \equiv (\text{executeLadder } ((a, \varphi) :: r_2)\ (\text{executeLadder } r_1\ s\ i)\ i)[x] \\
&\qquad\qquad \textit{by Lemma 9.3.7} \\
&\quad \equiv (\text{executeLadder } r_1\ s\ i)[x] \\
&\qquad\qquad \textit{by Lemma 9.3.8} \\
&\quad \equiv ((\text{executeLadder } r_1\ s\ i) +\!\!+ i)[x] \\
&\qquad\qquad \textit{by Lemma 9.3.14}
\end{aligned}
$$

case $\forall i \,.\, \pi_0(r_1[i]) \not\equiv x$**:**

$$
\begin{aligned}
&(s +\!\!+ i +\!\!+ (\text{executeLadder } r\ s\ i))[\text{shift}'_n\ r_1\ x] \\
&\quad \equiv (s +\!\!+ i +\!\!+ (\text{executeLadder } r\ s\ i))[x] \\
&\qquad\qquad \textit{by Lemma 9.3.2} \\
&\quad \equiv (s +\!\!+ i)[x] \\
&\qquad\qquad \textit{by Lemma 9.3.14} \\
&\quad \equiv ((\text{executeLadder } r_1\ s\ i) +\!\!+ i)[x] \\
&\qquad\qquad \textit{by Lemma 9.3.8 and Lemma 9.3.15}
\end{aligned}
$$

□

Lemma 9.3.13 (extend-env)**.** *Assume two lists of Booleans ξ and ζ. Given an $x : \mathbb{N}$ less-than the length of ξ, then the following holds*

$$\xi[x] \equiv (\zeta +\!\!+ \xi)[(\text{length } \zeta) + x]$$

Proof. Follows by simple induction on ζ. □

Lemma 9.3.14 (extend-env2)**.** *Assume two lists of Booleans ξ and ζ. Given an $x : \mathbb{N}$ less-than the length of ξ, then the following holds*

$$\xi[x] \equiv (\xi +\!\!+ \zeta)[x]$$

Remark

Lemma 9.3.13 also holds when ξ and $\zeta +\!\!+ \xi$ are lifted to maps, in which case the restriction on x is not required.

Proof. Follows by induction on ξ and side induction on x. □

Lemma 9.3.15 (substitute environment)**.** *Assume three lists of Booleans ξ_1, ξ_2 and ζ, a unary relation P on natural numbers, such that*

$$\text{length } \xi_1 = \text{length } \xi_2$$

and

$$\forall i \,.\, P\ i \to \xi_1[i] \equiv \xi_2[i]$$

Then the following holds

$$\forall i \,.\, P\ i \to (\xi_1 +\!\!+ \zeta)[i] \equiv (\xi_2 +\!\!+ \zeta)[i]$$

Proof. Follows by induction on the length of ξ_1. □

Finally, assembling the initial state and transition function into a decidable transition system is defined in the following corollary:

Corollary (Decidable Ladder)**.** *Assume a well-formed ladder, then the ladder defines a decidable transition system.*

Proof. Define the function

```
mkdts : (l : Ladder) → DecidableTransitionSystem_(mkts l)
mkdts l = record {
            initialState      = mkInitialState l      ;
            initialCorrect    = Theorem 9.3.3         ;
            transitionFunction = mkTransitionFunction l ;
            transitionCorrect = Theorem 9.3.10        }
```

□

9.3.1 Architectural State

As previously noted, the function nthState is definable from a decidable transition system. This induced function will generate a sequence of states

from the transition system and a stream of inputs. For a decidable ladder l, the stream of inputs is given by the following type:

$$\text{Inputs}_l = \mathbb{N} \to \text{Input}_l$$

What is required in Chapter 8 is a sequence of architectural states, or equivalently states of the abstract layout. Therefore, a mapping is required between states of the interlocking (lists of Bool) and architectural states (signal aspects, train routes and segment locks).

From the experience, gained while verifying a variety of interlocking systems, the signal aspects and segment locking state are directly extractable from the state of an interlocking. In the case of mechanical interlocking systems, these relate to the configuration of a lever. With modern interlocking systems, these are Boolean values stored in memory.

At least with the interlocking systems explored, identifying which route a train currently occupies requires some work. This is because the interlocking systems are only equipped with track detection inputs. These inputs do not know which train is occupying which route (or segment), only that a segment is occupied. Internally the interlocking systems keep track of which routes have been allocated, but this information does not identify which train is in the route. Therefore, a *train simulator* was devised. Each train is allocated an initial route and an input stream that defines whether the train is attempting to move or not. When it attempts to move, the input also identifies which route the train is requesting to traverse. For an abstract layout a, the train input streams are given by the following type

$$\text{TrainInputs}_a = \text{Train}_a \to \mathbb{N} \to \{\text{Move } \mathit{route}, \text{Stationary}\}$$

and the following functions are assumed:

$$\begin{aligned} &\text{initPosition}_a : \text{Train}_a \to \text{Route}_a \\ &\text{trainInput}_a : \text{TrainInputs}_a \end{aligned}$$

Provided with such input streams, we define a train simulator, such that it satisfies the (Correct-Train) axiom. The simulator is split into two parts, the first selects a route that has a proceed aspect from a selection of routes (or selects a default route, if none of the routes are set to proceed), and the second part recursively calculates the position of the train. Note that it is required that the abstract layout a is defined over the physical layout p in

the following.

$$
\begin{aligned}
&\text{trainPosition}_{l,a} : \text{Inputs}_l \to \text{TrainInputs}_a \to \mathbb{N} \to \text{Train}_a \to \text{Route}_a \\
&\text{trainPosition}_{l,a}\ inp\ tinp\quad 0\quad tr\ =\ \text{initPosition}_a\ tr \\
&\text{trainPosition}_{l,a}\ inp\ tinp\ (\text{suc}\ t)\ tr\ = \\
&\quad \textbf{if}\ (tinp\ tr\ t = \text{Move}\ rt) \\
&\qquad \wedge\ (rt\ \text{isin}\ (\text{connected}_p\ (\text{trainPosition}_{l,a}\ inp\ tinp\ t\ tr))) \\
&\qquad \wedge\ \text{“}rt\ \text{is set to proceed in}\ (\text{nthState}_l\ inp\ t)\text{”} \\
&\quad \textbf{then}\ rt \\
&\quad \textbf{else}\ \text{trainRoute}\ t\ tr
\end{aligned}
$$

Proving that this definition fulfils the (Correct-Train) axiom is only done for concrete instances of the layouts, see Section 11.1.3 for a concrete example of the simulation.

This simulator only allows a train to transition between two connected routes which have a proceed aspect. If a train's input stream indicates that it should move past a danger aspect, or to an unconnected route, then the simulator will overturn that request and stop the train. In effect, it is assumed that trains operate correctly. This assumption is not too significant because modern trains are fitted with a wide range of safety equipment that will prevent it passing a danger aspect (i.e. automatic train protection).

Remark

The above comment is not entirely accurate as there are certain cases where trains are allowed to passed danger signals, one example of this is when manually performing call-on manoeuvres (typically these manoeuvres are used to combine two trains). Another situation is where the signalling scheme explicitly makes provisions for trains accidentally passing a signal at danger, such as when *train stops* are in use. These are devices fixed to the track adjacent to the signal; they will ensure that a train passing a danger aspect will automatically apply its breaks and stop safely in a buffer zone beyond the signal known as an overlap. A radio link is also required in these situations; it informs the driver and train when the signal has cleared, and it is safe to proceed. Nonetheless, these examples are the exception and not the rule, so it is valid to assume that trains do not pass signals at danger.

While proving that the signalling principles held for a concrete interlocking system, it was found that it was required to assume the inputs and states are consistent with the computed train routes. Namely the track detection

inputs to the interlocking, and the internal route allocations are consistent with the trains routes.

Therefore, it is a simple matter to define a mapping between the interlocking states, and the architectural states. However, these mappings are dependent upon the interlocking and should be validated (by domain experts). For instance, modern interlocking systems can have many different variables with very similar semantics, and selecting the correct variable to project into the architectural state is non-trivial. Similarly, when assuming that trains operate correctly, it must be validated that the assumption is sound in terms of domain validation and that it is consistent with previous definitions.

9.4 Related Interlocking Work

The remainder of this chapter focuses on work related to interlocking systems that was undertaken during this project, but not directly related to the previous discussions. This consists of two parts, the first defines LadderCTL, a variant of symbolic model-checking specific to ladder logic (or more generally Boolean programs). The second part discusses an attempt at formalising the *Geographic Data* programming language used for the SSI family of interlocking systems.

9.4.1 LadderCTL

Remark

This section has been deferred from Chapter 4, because the definition of ladder logic was required first.

The symbolic CTL model-checking that was previously defined in Section 4.3 was an attempt to wrap the CTL model-checking (Section 4.2) in a layer of structure. That is the states are defined by variables (from finite domains), and the inputs are also defined by variables (from finite domains) instead of a single variable (from a finite domain). However, the symbolic model-checking interface does not provide any structure with respect to the transition function. Recall (on page 79) that the transition function for a SymFSM is defined as the abstract function:

$$\text{transition} : (s : \text{Record state}) \to \text{Record (arrow } s) \to \text{Record state}$$

where state is a list of $\mathbb{N}$.

As the function behaves as a black box, when generating the transition relation for input to an external model-checker, the only option is to enumerate the state space, and for each state the possible inputs. This information is then written to a file/piped to the model-checker. This of course results in a blow-up (state space explosion), to such an extent that the ladder logic programs relating to this project could not be output to the model-checker. For instance, consider a ladder with 100 state and 50 input variables (the data-sets donated to the project have over 300 state variables), the resulting translation would explicitly generate a transition relation with $2^{100} * 2^{50}$ cases. Writing such a formula to a file is infeasible.

To mitigate this issue, we defined a new model-checking theory (LadderCTL) over symbolic CTL. The purpose of LadderCTL is to facilitate a high-level and usable interface between Agda and the model-checker NuSMV. It builds upon the definition of symbolic CTL model-checking, but fixes all variables to be Booleans. That is, in effect only propositional logic is considered. The principal motivation behind LadderCTL is that it structures the transition function as a sequence of Boolean valued assignments. Therefore, when outputting the problem to an external model-checker, the precise structure of the ladder is canonically encoded. That is each variable in the ladder becomes a Boolean variable in the model, either input or state; each rung of the ladder becomes an equation in the model defining how the variable's value changes over time. Therefore, the state space is not explicitly computed by Agda, see Figure 9.6 for a comparison between the two approaches.

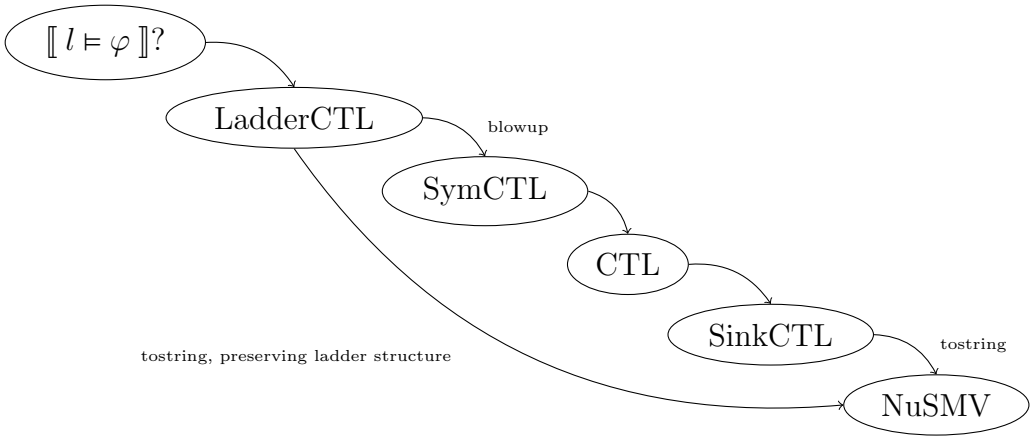


Figure 9.6: Where l is a ladder and φ is a CTL formula, the above shows the different translation options.

To define the plug-in interface for LadderCTL the 7 built-ins from Sec-

tion 5.4 must be defined. See module `CTL.Ladder` in Appendix F for the technical details of the definitions below. As already noted, the definitions from symbolic CTL are reused. The ladders as defined previously defined might not be decidable, for simplicity they are redefined as follows:

record Ladder : Set where
 field
  statevars : $\mathbb{N}$
  inpvars : $\mathbb{N}$
  rungs : List (Fin statevars $\times$ BooleanFormula (statevars + inpvars))
  initialstate : Vec_{Bool} statevars

Here

$$\text{BooleanFormula } n = \{\text{false}, \neg\_, \_\wedge\_, \_\vee\_, \text{var}\_\}$$

and the variables are indexed by Fin n.

The formula are similar to SymCTL except that the state propositions are testing the truth of Boolean values.

data LadderCTL (l : Ladder) : Set where
 false : LadderCTL_l
 $\neg$: $\text{LadderCTL}_l \to \text{LadderCTL}_l$
 $\_\vee\_$: $\text{LadderCTL}_l \to \text{LadderCTL}_l \to \text{LadderCTL}_l$
 $\_\wedge\_$: $\text{LadderCTL}_l \to \text{LadderCTL}_l \to \text{LadderCTL}_l$
 P[_] : (x : Fin statevars$_l$) $\to$ LadderCTL$_l$
 EX : $\text{LadderCTL}_l \to \text{LadderCTL}_l$
 EG : $\text{LadderCTL}_l \to \text{LadderCTL}_l$
 E[_U_] : $\text{LadderCTL}_l \to \text{LadderCTL}_l \to \text{LadderCTL}_l$

Note that in the following for a ladder l

$$\text{State}_l = \text{Vec}_{\text{Bool}}\ \text{statevars}_l$$

and

$$\text{Input}_l = \text{Vec}_{\text{Bool}}\ \text{inpvars}_l$$

In the interest of simplicity, the executeLadder function previously defined (on page 202) is trivially lifted to vectors instead of lists. Therefore the following transition function is obtained:

$\text{mkTransitionFunction}_l : \text{State}_l \to \text{Input}_l \to \text{State}_l$
$\text{mkTransitionFunction}_l = \text{executeLadder rungs}_l$

Runs of the ladder are canonically defined as follows:

$$\begin{aligned}
&\text{data } \text{LadderRun}_l\ (s : \text{State}_l) : \text{Set where}\\
&\quad \text{next} : (i : \text{Input}_l)\\
&\qquad \to \infty\, \text{LadderRun}_l\ (\text{mkTransitionFunction}_l\ s\ i)\\
&\qquad \to \text{LadderRun}_l\ s
\end{aligned}$$

The semantics is given as follows:

$$\begin{aligned}
&[\![_ , _ \vDash _]\!] : (l : \text{Ladder}) \to \text{State}_l \to \text{LadderCTL}_l \to \text{Set}\\
&[\![\ l\ , s \vDash \quad \text{false} \quad]\!] = \bot\\
&[\![\ l\ , s \vDash \quad \neg\varphi \quad]\!] = [\![\ l\ , s \vDash \varphi\ ]\!] \to \bot\\
&[\![\ l\ , s \vDash \quad \varphi \vee \psi \quad]\!] = [\![\ l\ , s \vDash \varphi\ ]\!] + [\![\ l\ , s \vDash \psi\ ]\!]\\
&\boldsymbol{[\![\ l\ , s \vDash \quad \mathbf{P}[\ x\ ] \quad]\!] = \mathbf{T}\ (s[x])}\\
&[\![\ l\ , s \vDash \quad \text{EX}\ \varphi \quad]\!] = \exists\ (run : \text{LadderRun}_l\ s)\ [\![\ l\ , run_1 \vDash \varphi\ ]\!]\\
&[\![\ l\ , s \vDash \quad \text{EG}\ \varphi \quad]\!] = \exists\ (run : \text{LadderRun}_l\ s)\ (\forall i\ .\ [\![\ l\ , run_i \vDash \varphi\ ]\!])\\
&[\![\ l\ , s \vDash \text{E}[\ \varphi\ \text{U}\ \psi\ ]\ ]\!] = \exists\ (run : \text{LadderRun}_l\ s)\ \exists\ (k : \mathbb{N})\ (\\
&\qquad\qquad (\forall j\ .\ j < k \to [\![\ l\ , run_j \vDash \varphi\ ]\!])\\
&\qquad\qquad \times [\![\ l\ , run_k \vDash \psi\ ]\!])
\end{aligned}$$

That concludes the definition of LadderCTL. In the following, it is explained how to show that LadderCTL is correct with respect to symbolic CTL model-checking.

As LadderCTL defines the state space and inputs by the number of Boolean variables, and SymCTL defines the state space and inputs by list of natural numbers. Which in the case of LadderCTL would be lists of '2', where each '2' represents the values each Boolean variable can take. Thus in the following, the following function is required that constructs lists of constant values.

$$\begin{aligned}
&\text{repeat} : \forall\{A\}\ .\ A \to \mathbb{N} \to \text{List}\ A\\
&\text{repeat}\ a\ 0 \quad\quad = [\,]\\
&\text{repeat}\ a\ (\text{suc}\ n) = a :: \text{repeat}\ a\ n
\end{aligned}$$

Lemma 9.4.1 (vector-record-iso)**.** *Assume* $n : \mathbb{N}$*. The following types are isomorphic:* $\text{Vec}_{\text{Bool}}\ n$ *and* $\text{Record}\ (\text{repeat}\ 2\ n)$

Proof. Follows by simple induction on n, and by Bool being isomorphic to Fin 2. □

In the following the bijection between Bool and Fin 2 is given as follows:

$$\begin{aligned}
\text{false} &\mapsto 0\\
\text{true} &\mapsto 1
\end{aligned}$$

By Lemma 9.4.1, for ladder l the following pair of isomorphic functions exist:

$$\begin{aligned}
&\text{toState}_l : \text{State}_l \to \text{Record (repeat 2 statevars}_l\text{)}\\
&\text{fromState}_l : \text{Record (repeat 2 statevars}_l\text{)} \to \text{State}_l\\
&\text{toInput}_l : \text{Input}_l \to \text{Record (repeat 2 inpvars}_l\text{)}\\
&\text{fromInput}_l : \text{Record (repeat 2 inpvars}_l\text{)} \to \text{Input}_l
\end{aligned}$$

Translating a ladder with n state variables and m input variables into a symbolic FSM, results in a SymFSM with the state space given by an n-ary product of Fin 2, and the inputs are fixed to a constant m-ary product of Fin 2. The following is defined:

```
toSymFSM : Ladder → SymFSM
toSymFSM l =
  fsm (repeat statevars_l)
      (λ_ → repeat inpvars_l)
      (toState initialstate_l)
      (λ s a → toState_l (transition_l (fromState_l s) (fromInput_l a)))
```

By Lemma 9.4.1, the toRun/fromRun functions previously introduced for SymRun are trivial to lift to LadderRun. That is, for a ladder l the following functions are obtained

$$\begin{aligned}
&\text{toSymRun}_l : \forall s\ .\ \text{LadderRun}_l\ s \to \text{SymRun}_{(\text{mkSymFSM}\ l)}\ (\text{toState}_l\ s)\\
&\text{fromSymRun}_l : \forall s\ .\ \text{SymRun}_{(\text{mkSymFSM}\ l)}\ (\text{toState}_l\ s) \to \text{LadderRun}_l\ s
\end{aligned}$$

and the formulæ are translated as follows:

$$\begin{aligned}
&\text{toSymCTL}_l : \text{LadderCTL}_l \to \text{SymCTL}_{(\text{toSymFSM}\ l)}\\
&\text{toSymCTL}_l\ \text{false} &&= \text{false}\\
&\text{toSymCTL}_l\ (\neg\ \varphi) &&= \neg\ (\text{toSymCTL}_l\ \varphi)\\
&\text{toSymCTL}_l\ (\varphi \vee \psi) &&= (\text{toSymCTL}_l\ \varphi) \vee (\text{toSymCTL}_l\ \psi)\\
&\text{toSymCTL}_l\ (\varphi \wedge \psi) &&= (\text{toSymCTL}_l\ \varphi) \wedge (\text{toSymCTL}_l\ \psi)\\
&\mathbf{toSymCTL_l\ (P[\ x\ ])} &&\mathbf{= P\ [\ x\ ==\ 1\ ]}\\
&\text{toSymCTL}_l\ (\text{EX}\ \varphi) &&= \text{EX}\ (\text{toSymCTL}_l\ \varphi)\\
&\text{toSymCTL}_l\ (\text{EG}\ \varphi) &&= \text{EG}\ (\text{toSymCTL}_l\ \varphi)\\
&\text{toSymCTL}_l\ (\text{E}[\ \varphi\ \text{U}\ \psi\ ]) &&= \text{E}[\ (\text{toSymCTL}_l\ \varphi)\ \text{U}\ (\text{toSymCTL}_l\ \psi)\ ]
\end{aligned}$$

Finally, the correctness is given by the following lemma:

Lemma 9.4.2 (ladderctl-correct)**.** *Assume a ladder l, a state s of the ladder and a CTL formula φ. Then the following holds*

$$[\![\, l, s \vDash \varphi \,]\!]^{\text{Ladder}} \leftrightarrow [\![\, \text{toSymFSM}\, l, \text{toState}_l\ s \vDash \text{toSymCTL}_l\ \varphi \,]\!]^{\text{Sym}}$$

Proof. Follows by induction on φ. The propositional cases follow by induction hypothesis, the quantified CTL cases follow the same argument as Theorem 4.3.13 but using the isomorphism Lemma 9.4.1. It remains to prove the case when $\varphi = \text{P}[x]$. The obligation to prove is

$$\text{T}\ (s[x]) \leftrightarrow \text{T}\ (\text{lookup}\ (\text{repeat}\ 2\ \text{statevars}_l)\ (\text{toState}_l\ s)\ (x, 1))$$

which follows by side induction on s and x. □

The built-in function ATPInput (see page 104) then simply recurses over the list of assignments, translating them into assignments that NuSMV understands. This is slightly worse than linear time because the assignments in the NuSMV are unordered, whereas, in the ladder, the order of the assignments matters. Therefore, ATPInput keeps track of which rungs have been processed (and which have not), it then adjusts any reference to the processed variables with the `next` function. This means that it fetches the next value of the variable, instead of the current value. See Figure 9.7 for an example of the Pelicon crossing ladder.

Concluding Remarks The LadderCTL example demonstrates the power and flexibility of this approach; it was quick (approximately 2 days) to implement this extension on top of symbolic model-checking.

In practice, this interface was only successfully used with toy examples, i.e. that of the Pelicon crossing. When tried with a large ladder with approximately 300 state variables and 200 input variables, it was discovered that NuSMV was not terminating in a reasonable amount of time. Agda could generate the input files in a couple of minutes. The same problem set the SAT solver could inductively solve in < 1 second, could not be solved by NuSMV in ≈ 46 days. The test was terminated due to a segmentation fault. A number of different command line arguments for NuSMV were explored, but either it would run out of memory, or take significant amounts of time during a reachablity analysis.

For this reason, a much larger portion of this project was done by SAT solvers.

```
MODULE main
IVAR
    pressed : boolean;
VAR
    requested : boolean;
    crossing : boolean;
    plightg : boolean;
    tlightg : boolean;
ASSIGN
    init(requested) := FALSE;
    init(crossing) := FALSE;
    init(plightg) := FALSE;
    init(tlightg) := FALSE;
    next(crossing) := (!(crossing) & requested);
    next(requested) := (pressed & !(next(crossing)));
    next(plightg) := next(crossing);
    next(tlightg) := (!(next(crossing)) & !(next(requested)));
SPEC
    !(E[!(FALSE) U !((!(plightg)) | (!(tlightg)))]);
```

Figure 9.7: Structure preserving translation of Pelicon ladder from Figure 1.2 into NuSMV input. The property to be checked is *there does not exist a path, such that at some point on the path* $\neg(\neg plightg \vee \neg tlightg)$ *holds*, which is equivalent to *for all paths its always the case that* $\neg plightg \vee \neg tlightg$ *holds.* See module `CTL.Pelicon` in Appendix F for more information.

9.4.2 Geographic Data

During this project, an attempt was made to formalise the geographic data language. This language is used to program solid-state interlocking (SSI) systems [Cri87, Net05]. These are conceptually different from the ladder logic interlocking systems. The name geographic data refers to the geographically customised part of the interlocking program. Thus, an interlocking program consists of two parts, generic and specific. The generic part is the same across all SSI's, whereas the specific part is changed depending on the topology and signalling principles.

The computational model of the SSI is based upon sending and receiving telegrams with the track-side equipment, other control systems and the trains. These telegrams have a fixed binary representation. When a telegram is received, an associated block of instructions is evaluated; these instructions prepare an output telegram, or a number of them. This computational model

is amenable to being verified using a process calculi based technique—as was done successfully by Mathew Morley in [Mor96].

The geographic data was explored in this work because of the problem that ladder logic had to be translated from a graphical language into propositional logic outside of Agda. This means that trust must be placed in an external tool that implemented the translation. It was the goal to directly, or almost directly, copy and paste the geographic data into Agda, then using Agda's flexible parsing, parse the geographic data into an Agda data-type. From this data-type, it would be possible to verify properties, and hence safety of the geographic data.

Significant portions of the geographic data and generic part of the SSI were formalised into Agda. But it soon became clear that there were two issues

Non-Termination: The generic part of the interlocking program allowed for non-terminating programs to be written. Namely, when calling blocks of code that would test bits in telegrams, these blocks could reference other blocks resulting in non-termination. This issue was raised with Invensys Rail, who stated that this is a known problem. Further, it was also stated that a newer version of the language had been devised with strict rules to mitigate non-termination, but the coders did not like using it due to the restrictions. Sadly, the newer version was never formalised in this work.

There was also a non-termination issue when performing map-searches, the discussion is omitted due to the technical nature of a map-search.

Parsing: A second issue was that parsing the geographic data directly (using the Agda parser) would have required an "explosion" of data constructors. This issue was also related to testing bits in a telegram, where the tests were specified by a string of characters. In most cases these characters were un-ordered, so the test "abc" is equivalent to the test "bca". In essence, a pre-processing of the data would have been required that ordered these strings. This was not desirable as the main reason for exploring geographic data was to directly verify the source code. There were also small issues with special symbols in Agda such as the full-stop.

These issues resulted in the geographic data branch of the project to be abandoned. However, it is conjectured that due to the Boolean/binary nature of the data, tests on the data, and internals of the SSI that SAT based verification of the interlocking is possible.

Chapter 10
Verification

Chapter 8 showed how to model safety in the railway domain, and Chapter 9 introduced railway interlocking systems realised by ladder logic programs. This chapter presents a method of showing interlocking systems, realised by ladder logic programs, fulfil the domain safety by using a combination of SAT and interactive theorem proving. This framework has been successfully applied to two interlocking systems.

Chapter Overview. In Section 10.1 the issue of how expressive the properties that need to verified is explored; then a type of verification conditions is defined. To illustrate how these verification conditions can be used, a method of systematically generating them from a control table is presented in Section 10.2. Section 10.3 discusses how verification conditions are defined to show that an interlocking system fulfils the necessary signalling principles, and Section 10.4 demonstrates how to prove that the pelicon crossing ladder from Chapter 1 is safe. In Section 10.5 remarks are made about our experience verifying industrial railway interlocking systems using this framework. Finally Section 10.6 discusses how to extract an executable control system from the verification framework.

The next chapter contains a complete example of the verification for Gwili Steam Railway.

10.1 Verification Conditions

The first step to proving safety properties about ladder logic programs is to formalise the properties that we wish to prove. Then, from these formalisations, a framework is defined which is expressive enough to represent the properties with respect to a ladder logic program.

Recall from Section 9.2 that ladder logic programs are transitions systems

that consist of states, and arrows between these states. Some of the safety properties that are of interest only reference a singular state, or equivalently an output as there is no distinction made between output and state variables. For instance, one such condition would be *for all (reachable) states, two opposing signals are not both set to proceed*. This property should always hold[1], regardless of the previous state and inputs. There are also properties which depend upon the input and output of the ladder. For instance, one such property is *if a track segment protected by a signal is occupied, then that signal displays the danger aspect*. Furthermore, the most expressive properties that are required to hold in this work also depend upon the previous state. Thus, the properties of interest here are relations between two states and an input, i.e. transitions. For a given transition system ts, the type of a verification condition is given by

$$\mathrm{State}_{ts} \to \mathrm{Input}_{ts} \to \mathrm{State}_{ts} \to \mathrm{Set}$$

which, in the case of

$$ts = \mathrm{mkts}\ l \tag{$*$}$$

for some ladder l, becomes

$$\mathrm{State}_{l} \to \mathrm{Input}_{l} \to \mathrm{State}_{l} \to \mathrm{Set}$$

In most of this chapter, the transition systems considered follow from ladder programs, i.e. as in $(*)$ above.

Intuitively, the correctness of a verification condition φ is formalised by requiring that for all reachable states s, and inputs i,

$$\varphi\ s\ i\ s'$$

holds, where s' is the resulting state from being in state s and following arrow i. That is, verification conditions are defined over transitions.

Before formally defining correctness, it is required to consider reachable states. Due to the dependent type theory that Agda is built on, it is possible to define the type of reachable states for a transition system, provided there is an initial state.

In general, this can be achieved by applying induction-recursion [DS01]. However, the transitions systems used here allow for a simpler definition of reachable states. An inductive data-type is given that witnesses for a given state a trace from the initial state to this state. For a transition system

[1] It might not be the case in manual/maintenance mode.

ts, the data-type of reachablity is given in module `TransitionSystem` of Appendix F as follows:

$$
\begin{array}{l}
\text{data ReachableState}_{ts} : \text{State}_{ts} \to \text{Set where} \\
\quad \text{initial} \;:\; (s : \text{State}_{ts}) \to \text{Initial}_{ts}\ s \to \text{ReachableState}_{ts}\ s \\
\quad \text{next} \quad :\; (s : \text{State}_{ts}) \\
\qquad\qquad \to (r : \text{ReachableState}_{ts}\ s) \\
\qquad\qquad \to (i : \text{Input}_{ts}) \\
\qquad\qquad \to (s' : \text{State}_{ts}) \\
\qquad\qquad \to \text{Transition}_{ts}\ s\ i\ s' \\
\qquad\qquad \to \text{ReachableState}_{ts}\ s'
\end{array}
$$

The correctness of a verification condition φ for transition in the transition system ts is now formalised as follows:

$$
\begin{array}{l}
\text{Correct}_{ts}\ \varphi = (s : \text{State}_{ts}) \\
\qquad\qquad \to \text{ReachableState}_{ts}\ s \\
\qquad\qquad \to (i : \text{Input}_{ts}) \\
\qquad\qquad \to (s' : \text{State}_{ts}) \\
\qquad\qquad \to \text{Transition}_{ts}\ s\ i\ s' \\
\qquad\qquad \to \varphi\ s\ i\ s'
\end{array}
$$

In the case of ladder logic's transition systems, verification conditions are given by Boolean formulæ that reference variables in the previous state, input and successor state. The correctness of a transition system is then lifted to ladder logic in module `Ladder.Core` of Appendix F as follows:

$$
\begin{array}{l}
\text{LadderCorrect} : \text{Ladder} \to \text{BooleanFormula} \to \text{Set} \\
\text{LadderCorrect}\ l\ \varphi \;=\; \text{Correct}_{(\text{mkts}\ l)}\ (\lambda\ s\ i\ s' \,.\, [\![\ \varphi\ ]\!]_{(s +\!\!+ i +\!\!+ s')})
\end{array}
$$

Correctness. In this chapter, proving a verification condition φ over a ladder program uses the principle of induction. This works by (1) proving that φ holds for all the initial transitions; and (2) that for any transition which satisfies φ, all the next transitions also satisfy φ. A transition here is the tuple $< s, i, s' >$ such that $\text{Transition}_{ts}\ s\ i\ s'$. It is a well-known issue that some states might be unreachable, causing false negatives while applying an inductive proof strategy. A full discussion of this issue in the context ladder logic and the railway domain can be found in [JR10].

In the system presented here, membership decidability of the correctness condition for ladder logic is by a Boolean formula. This results in Boolean valued proof objectives. In practice, interactively proving the validity of these formulæ is a tedious job, especially for industrial applications. Therefore, the

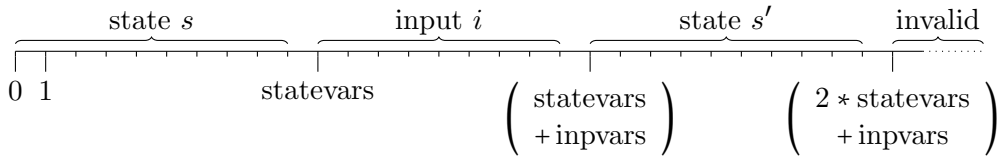
use of an integrated SAT solver, as presented in the first part of this thesis is applied. The Boolean formulæ representing (1) and (2) are given as

$$\begin{aligned}&\text{base-obligation} : \text{Ladder} \to \text{BooleanFormula} \to \text{BooleanFormula} \\ &\text{base-obligation}\ l\ \varphi\ =\ (\text{mkinit initstate}_l) \wedge (\text{mktrans}\ []\ \text{rungs}_l) \Rightarrow \varphi\end{aligned} \tag{1}$$

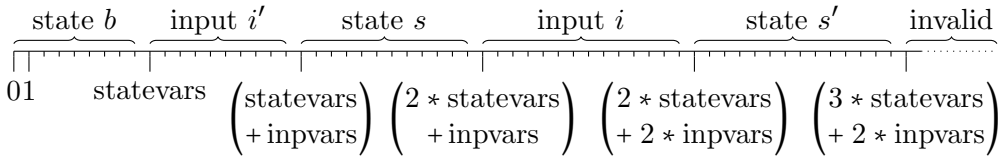
and

$$\begin{aligned}&\text{ind-obligation} : \text{Ladder} \to \text{BooleanFormula} \to \text{BooleanFormula} \\ &\text{ind-obligation}\ l\ \varphi\ =\ \big(\varphi \wedge (\text{mktrans}\ []\ \text{rungs}_l) \wedge \\ &\qquad (\text{shift}_{\text{statevars}_l}\ \text{inp-correct}_l) \wedge \\ &\qquad (\text{shift}_{(n+\text{statevars}_l)}\ \text{inp-correct}_l) \wedge \\ &\qquad (\text{shift}_n\ (\text{mktrans}\ []\ \text{rungs}_l))\big) \Rightarrow \text{shift}_n\ \varphi\end{aligned} \tag{2}$$

where $n = \text{statevars}_l + \text{inpvars}_l$. To clarify the formulæ above, two number lines are presented. First, for the base case (1) the following number line is used for the variable indices:



It splits up the variables into a state s, an input i and a state s'. The beauty of this technique is that it is trivial to shift the formulæ for the inductive case (2) and adjust the number line as follows:



Although all the pieces are available to prove/dis-prove these safety properties automatically, it is cumbersome to use. One significant issue faced was with the soundness proof of the SAT solver: if applied in certain ways, then the ladder would be normalised inside of Agda a number of times. This excessive normalisation reduced efficiency, particularly when generating the transition relation for large ladders. This inefficincy was mitigated by devising a *tactic-like* function. It is an *abstractly*[2] defined Agda function that encapsulated the soundness proof, and inductive strategy. Thus, when applied, the user would only have to provide the proofs of (1) and (2), which are

[2]Under normal circumstances in Agda, an abstract function does not β-reduce.

decided by a SAT solver. This resulted in the end-user not having to understand anything about any of the translations, transition systems or reachable states. Due to the technical nature of this function, only the type signature is presented in Agda code, and an intuitive proof is given for its implementation. However, the full definition can be found in module `Ladder.Core` of Appendix F.

$$
\begin{aligned}
\text{inductiveProof} : (l : \text{Ladder}) \\
&\to (\varphi : \text{BooleanFormula}) \\
&\to \{p : \mathrm{T}\ (\text{tautology}\ (\text{base-obligation}_l\ \varphi))\} && (1)\\
&\to \{q : \mathrm{T}\ (\text{tautology}\ (\text{ind-obligation}_l\ \varphi))\} && (2)\\
&\to \{bi : \mathrm{T}\ (\text{bound}\ \text{statevars}_l\ (\text{mkinit}\ \text{initstate}_l))\} && (3)\\
&\to \{bt : \mathrm{T}\ (\text{bound}\ n\ (\text{mktrans}\ [\,]\ \text{rungs}_l))\} && (4)\\
&\to \{b\varphi : \mathrm{T}\ (\text{bound}\ n\ \varphi)\} && (5)\\
&\to \{binv : \mathrm{T}\ (\text{bound}\ \text{inputvars}_l\ \text{inp-correct}_l)\} && (6)\\
&\to \text{LadderCorrect}\ l\ \varphi
\end{aligned}
$$

where $n = \text{statevars}_l + \text{inputvars}_l + \text{statevars}_l$. In the above, tautology is replaced by a call to a SAT solver, thus when using this tactic, first, the SAT solver is called on the base-case (1), then it is called on the inductive-case (2). Finally, Agda checks that the formulæ are well-formed (3), (4), (5) and (6). The result is a proof-object that states that φ always holds for all reachable states of ladder l.

Proof. The proof of inductiveProof is by induction on the reachable states. First, assume a reachable state s, and a proof of its reachability r, an input i and a second state s' such that t is a proof that s' is reachable from s by input i.

case $r = \text{initial}\ s\ x$**:** By Theorem 4.1.2 the following function is obtained:

$$[\![\ (\text{mkinit}\ \text{initstate}_l) \wedge (\text{mktrans}\ [\,]\ \text{rungs}_l) \Rightarrow \varphi\ ]\!]_{(s +\!\!+ i +\!\!+ s')}$$

The proof follows by Lemma 9.3.14 applied to x and (3).

case $r = \text{next}\ b\ r'\ i'\ s\ t'$**:** Let $n = \text{statevars}_l + \text{inpvars}_l$. By Theorem 4.1.2 and the induction hypothesis, the following function is obtained:

$$
\left[\!\!\left[\ \begin{array}{l}
\big(\varphi \wedge (\text{mktrans}\ [\,]\ \text{rungs}_l) \wedge \\
\ (\text{shift}_{\text{statevars}_l}\ \text{inp-correct}_l) \wedge \\
\ (\text{shift}_{(n+\text{statevars}_l)}\ \text{inp-correct}_l) \wedge \\
\ (\text{shift}_n\ (\text{mktrans}\ [\,]\ \text{rungs}_l))\big) \Rightarrow \text{shift}_n\ \varphi
\end{array}\ \right]\!\!\right]_{(b +\!\!+ i' +\!\!+ s +\!\!+ i +\!\!+ s')}
$$

The obligation to prove is:

$$\begin{array}{l}\big(\llbracket\ \varphi\ \rrbracket_{(b +\!\!+ i' +\!\!+ s)} \wedge \\ \llbracket\ \text{mktrans}\ []\ \text{rungs}_l\ \rrbracket_{(b +\!\!+ i' +\!\!+ s)} \wedge \\ \llbracket\ \text{inp-correct}_l\ \rrbracket_{i'} \wedge \\ \llbracket\ \text{inp-correct}_l\ \rrbracket_{i} \wedge \\ \llbracket\ \text{mktrans}\ []\ \text{rungs}_l\ \rrbracket_{(s +\!\!+ i +\!\!+ s')}\big) \rightarrow \llbracket\ \varphi\ \rrbracket_{(s +\!\!+ i +\!\!+ s')}\end{array}$$

Which follows by (4), (5), (6), Lemma 9.3.13 and Lemma 9.3.14.

□

Remark

As this function is intended to be applied to closed terms, the proof-objects (1..6) should be decidable during type-checking. If any of the proof-objects fail to be inferred by Agda, either because of an open term or that they do not hold, then Agda will output a message stating that there is a hidden open term that could not be inferred. Also in the Emacs mode, the call-site of the function will be highlighted, indicating to the user that something has gone wrong.

10.2 Control Table Verification Conditions

It was hinted at in Chapter 8 that it is possible to derive verification conditions from a control table automatically. In this section, a basic framework is outlined for this purpose. It should be noted that this form of verification refers to operational correctness, rather than safety. The operational correctness means that the system does what it is supposed to do, i.e. fulfils its specification. Traditionally operational correctness is checked by testers, who check the system against a number of test-cases, which are derived from the specifications during the design of a system. From the author's experience, automatically verifying operational correctness is of interest to industry, namely because it allows for erroneous behaviours to be eradicated at an early stage of the development cycle; otherwise these behaviours might not be identified until the software has been passed to testers, increasing the total development time.

Intuitively, a ladder l is a model of a control table c *iff* l never violates the constraints in c. It is noted that some, but not all of the signalling principles

previously identified follow from control tables, one such principle would be *signals guard*.

From the definition of a control table c (see page 169), there is the field entries_c that maps a route to a control table entry. Thus to determine whether an interlocking system fulfils these constraints, a mapping from control table entries into verification conditions (Boolean formulæ) is required. Although it is possible to craft these formulæ by hand, it is preferable to automate the process. This automation requires the inverses of the projections from Section 9.3.1. That is maps from elements in the physical layout to the variables in the interlocking that represent properties of these elements. For example, a map from signals to the variables that represent when a signal shows a proceed or stop aspect is required. Along side the maps from Section 9.3.1, one additional map is required that relates routes (from the control table) to formulæ that represent whether they have been selected. In practice, these maps produce a Boolean formula rather than a single variable. This is because an abstract property in the model can be represented by a combination of interlocking variables. For example, when multiple routes start at the same signal, a route indicator is also present that indicates to the train driver which route the signal's aspect corresponds to; this means from a verification perspective that each route has a distinct signal, but is represented by multiple variables, one for the signal aspect and one for the route indicator.

To formalise the correctness of a ladder logic program l with respect to a control table c, defined over the physical layout p, assume the following maps are correct:

$$\text{rtSet} : \text{Route}_c \to \text{BooleanFormula}$$

Whether the route is currently in use.

$$\text{segNormal} : \text{Segment}_p \to \text{BooleanFormula}$$

Whether the segment is being controlled normal.

$$\text{segReverse} : \text{Segment}_p \to \text{BooleanFormula}$$

Whether the segment is being controlled reverse.

$$\text{segLocked} : \text{Segment}_p \to \text{BooleanFormula}$$

Whether the segment is locked.

$$\text{segOccupied} : \text{Segment}_p \to \text{BooleanFormula}$$

Whether the segment is occupied.

$$\text{sigProceed} : \text{Signal}_p \to \text{BooleanFormula}$$

Whether the signal displays proceed.

Then the verification condition that l correctly implements a control table entry is given by the following function:

$$\begin{aligned}
&\text{CorrectTable}_c : \text{Route}_c \to \text{BooleanFormula} \\
&\text{CorrectTable}_c\ rt\ = \\
&\left(\text{rtSet}\ rt \Rightarrow \left(\bigwedge_{i=0}^{\text{length normalpoints}_{rt}} \text{segNormal}\ \text{normalpoints}_{rt}[i]\right)\right. \\
&\qquad \wedge \left(\bigwedge_{i=0}^{\text{length reversepoints}_{rt}} \text{segReverse}\ \text{reversepoints}_{rt}[i]\right) \\
&\qquad \left.\wedge \left(\bigwedge_{i=0}^{\text{length facing}_{rt}} \text{segLocked}\ \text{facing}_{rt}[i]\right)\right) \quad \wedge \\
&\left(\text{sigProceed}\ rt \Rightarrow \left(\bigwedge_{i=0}^{\text{length segments}_{rt}} \neg\, \text{segOccupied}\ \text{segments}_{rt}[i]\right)\right)
\end{aligned}$$

It looks-up for a given route, its entry in the table. Then using the maps, constructs a Boolean formula that represents *if the route is selected, then all segments identified as normal/reverse/locked should be normal/reverse/locked* and *if a signal displays a proceed aspect, then its associated route is selected and all required segments are unoccupied.*

The formalisations of control tables in this work only have one entry, namely entries that define routes. In practice, there are many different types of entries, and the method outlined above is easily extended to allow for these.

Some verification conditions depend upon multiple control table entries, an example of such a condition would be "routes that share at least one track segment cannot both be set". This requires considering all pairs of routes. It is a straightforward matter to define a version of CorrectTable that depends upon multiple entries, but this is left to the reader.

Once all conditions have been generated, then just taking their conjunction formalises when a ladder logic program correctly refines the control table. It is then possible to apply the function inductiveProof described previously to determine automatically whether the ladder logic program is a correct model of the control table.

Remark

In this framework, the control table entries have no formal semantics; they are a collection of identifiers, routes, signals, tracks, etc. The verification conditions give semantics to the entries, as inputs and outputs are related back to the layout. Moreover, verification conditions are sentences built over a physical layout; they can be generated from the relations in the control table by instantiating a template sentence. For instance, the route entry can be mapped to a correctness relation which states "if a track segment in the route is occupied, then the signal shows a danger aspect".

It should be noted that this project is concerned with verifying whether an interlocking system is safe, and not by whether it is a correct model of a control table. However, it is conjectured that safety verification can be directly performed on the control tables.

For this reason, a generic framework to determine whether an interlocking system is a correct model of a control table was not implemented. What was successfully done, was to perform control table verification on a small number of interlocking systems of various sizes. This involved having to alter the formalisation of a correct table to reflect the interlocking systems technology. For example in Chapter 11, the interlocking cannot detect occupied segments, so the second conjunct of CorrectTable had to be removed. The control table verifications explored showed that it is feasible, at least in the railway domain to determine the correctness of an interlocking system with respect to its specification automatically.

10.3 Signalling Principles – Tying the Knot

Proving that an interlocking system is safe, as was discussed in Chapter 8, requires a proof that the interlocking system fulfils the four signalling principles (cf. Section 8.4). In the case of interlocking systems realised with ladder logic, verification conditions can be given that imply the signalling principles. Formulating these conditions requires maps from components in the physical layout to the variables. These are the same maps previously identified on page 225. It is important that these maps are validated.

It is often the case that the signalling principles are too abstract for a SAT solver to prove directly, usually this is the case because the maps are applied to an open term, such as a term quantified route or signal. Thus, case-distinctions are required on these data-types that the principle is quantified

over. This is the case with the opposing routes signalling principle, where any two routes that share a track segment cannot both be clear at the same time. To prove this, first, all pairs of routes that share a track segment are identified (by case-distinction and a dependent type formalising non-disjoint track segments), and then, for these pairs, it is shown that, for all time, they are never both clear. After identifying the pairs of routes, the formula representing the signalling principle becomes concrete enough (as the routes are now concrete) to be represented by a concrete Boolean formula, which a SAT solver then decides.

However, it was found that for large interlocking systems, performing these case-distinctions could become rather slow. This was because of unnecessary normalisations. In attempts to mitigate this, the nthState function was made abstract, which in effect inhibited unfolding of the decidable ladder. Also, the functions that generated propositional formula representing the ladder were marked as abstract[3], this prevented a large blow-up in the size of the terms, and significantly reduced the time required for type-checking.

10.4 Pelicon Crossing

Remark

A full Agda code listing is in Appendix F, under the modules that begin with `Pelicon`.

The Pelicon crossing example from Chapter 1 was the main toy example explored. It was verified using SAT and CTL; also safety verification was performed (including SAT). In this section, a formal treatment of the safety verification of the Pelicon crossing is presented. The verification builds upon the definitions in Section 1.1.1 of the Pelicon model, safety requirement and safety principle. To aid the reader, the relevant definitions are repeated below.

$$\begin{aligned} \text{numbercars}_0\ \text{MUX} &\equiv 0 && (\textit{pelicon-init}) \\ \text{movingcars}_0\ \text{T1}\ \text{MUX} &\equiv 0 \\ \text{movingcars}_0\ \text{T2}\ \text{MUX} &\equiv 0 \\ \text{movingcars}_{(t+1)}\ \text{MUX}\ \text{T2} &\equiv \text{movingcars}_t\ \text{T1}\ \text{MUX} && (\textit{taxm2}) \end{aligned}$$

[3]These formulæ were still un-folded before executing the SAT solver, but only for this purpose.

$$\text{movingcars}_{(t+1)}\ \text{MUX}\ \text{T1} \equiv \text{movingcars}_t\ \text{T2}\ \text{MUX} \qquad (\textit{taxm3})$$

$$\begin{aligned}\text{numbercars}_{(t+1)}\ \text{MUX} \equiv\ & (\text{numbercars}_t\ \text{MUX}) \\ & + (\text{movingcars}_{(t+1)}\ \text{T1}\ \text{MUX}) \\ & + (\text{movingcars}_{(t+1)}\ \text{T2}\ \text{MUX}) \\ & - (\text{movingcars}_{(t+1)}\ \text{MUX}\ \text{T1}) \\ & - (\text{movingcars}_{(t+1)}\ \text{MUX}\ \text{T2})\end{aligned} \qquad (\textit{taxm6})$$

Note that the pedestrian axioms *paxm2*, *paxm3* and *paxm6* are symmetric to *taxm2*, *taxm3* and *taxm6*, respectively.

pelicon-principle

$$\begin{aligned}\forall t\ .\ \big(&\text{movingcars}_t\ \text{T1}\ \text{MUX} \equiv 0 \wedge \text{movingcars}_t\ \text{T2}\ \text{MUX} \equiv 0\big) \\ \vee\ \big(&\text{movingpeds}_t\ \text{P1}\ \text{MUX} \equiv 0 \wedge \text{movingpeds}_t\ \text{P2}\ \text{MUX} \equiv 0\big)\end{aligned}$$

pelicon-safety

$$\forall t\ .\ \text{numbercars}_t\ \text{MUX} \equiv 0 \vee \text{numberpeds}_t\ \text{MUX} \equiv 0$$

From examining the axioms that show how cars and pedestrians interact with the crossings, it is clear that axiom *taxm6* and the pedestrian counter part *paxm6* are stream equations. These stream can be simplified by the following lemmata.

Lemma 10.4.1 (stream simplify)**.** *Given two streams X and Y of natural numbers, such that $X0 \equiv Y0$. If*

$$\forall n\ .\ X(n+1) \equiv Xn + Y(n+1) - Yn$$

holds, then

$$\forall m\ .\ Xm \equiv Ym$$

holds.

Proof. Follows by induction on n. Here, $Xn - Yn$ is safe, and always zero by the induction hypothesis. □

Lemma 10.4.2 (pelicon simplify)**.** *Assume a pelicon control system that fulfils axioms (pelicon-init), (taxm2), (taxm3) and (taxm6) (also the pedestrian*

versions of these axioms (paxm2), (paxm3) and (paxm6)), then for all time $t : \mathbb{N}$ *the following holds*

$$\text{numbercars}_t\ \text{MUX} \equiv \text{movingcars}_t\ \text{T1}\ \text{MUX} + \text{movingcars}_t\ \text{T2}\ \text{MUX}$$

and

$$\text{numberpeds}_t\ \text{MUX} \equiv \text{movingpeds}_t\ \text{P1}\ \text{MUX} + \text{movingpeds}_t\ \text{P2}\ \text{MUX}$$

Proof. Only the left conjunct is proved, as the right is symmetric. By (*taxm2*) and (*taxm3*) applied to (*taxm6*) the following is obtained.

$$\begin{aligned}\text{numbercars}_{(t+1)}\ \text{MUX} \equiv\ & (\text{numbercars}_t\ \text{MUX}) \\ & + (\text{movingcars}_{(t+1)}\ \text{T1}\ \text{MUX}) \\ & + (\text{movingcars}_{(t+1)}\ \text{T2}\ \text{MUX}) \\ & - ((\text{movingcars}_t\ \text{T1}\ \text{MUX}) \\ & \quad + (\text{movingcars}_t\ \text{T2}\ \text{MUX}))\end{aligned}$$

Therefore by Lemma 10.4.1 applied to

$$\begin{aligned}X\ t &:= \text{numbercars}_t\ \text{MUX} \\ Y\ t &:= \text{movingcars}_t\ \text{T1}\ \text{MUX} + \text{movingcars}_t\ \text{T2}\ \text{MUX}\end{aligned}$$

We obtain

$$\forall t\ .\ \text{numbercars}_t\ \text{MUX} \equiv \text{movingcars}_t\ \text{T1}\ \text{MUX} + \text{movingcars}_t\ \text{T2}\ \text{MUX}$$

□

Lemma 10.4.3 (pelicon safe)**.** *Assume a pelicon control system that fulfils axioms (pelicon-init), (taxm2), (taxm3) and (taxm6) (also the pedestrian versions of these axioms (paxm2), (paxm3) and (paxm6)), and the safety principle (pelicon-principle) holds, then (pelicon-safety) holds.*

Proof. Assume t. By (pelicon-principle) the following is obtained

$$\begin{aligned}&\big(\text{movingcars}_t\ \text{T1}\ \text{MUX} \equiv 0 \wedge \text{movingcars}_t\ \text{T2}\ \text{MUX} \equiv 0\big) \\ &\quad \vee \big(\text{movingpeds}_t\ \text{P1}\ \text{MUX} \equiv 0 \wedge \text{movingpeds}_t\ \text{P2}\ \text{MUX} \equiv 0\big)\end{aligned}$$

Without loss of generality assume, as the proof for the other case is symmetric.

$$\text{movingcars}_t\ \text{T1}\ \text{MUX} \equiv 0 \wedge \text{movingcars}_t\ \text{T2}\ \text{MUX} \equiv 0$$

Proof follows by Lemma 10.4.2. □

For full technical details of the above definitions, see Appendix F, module: `Pelicon.PeliconModel`.

The ladder logic program in Figure 1.2 is translated into the nthState function using the same methods as previously described in Section 9.3. See module `Pelicon.Ladder` of Appendix F. This nthState is then translated into the architectural signal aspects. See module `Pelicon.State` of Appendix F. The following two functions perform this translation:

$$\begin{array}{ll} \text{traffic} : \text{State}_{fig1.2} \to \text{Bool} & \text{pedestrian} : \text{State}_{fig1.2} \to \text{Bool} \\ \text{traffic } s \ = \ s[tlight.g] & \text{pedestrian } s \ = \ s[plight.g] \end{array}$$

The whole architectural state is a little more complicated; recall that the architectural state of the Pelicon crossing also contains the number of cars and pedestrians there are in each of the 5 areas, and the number that are transitioning between these areas. This information is not contained within a state of the ladder; thus a simulator is provided. The simulator works in a similar manner as the previously described train simulator. The input to the simulator, at a given time, is a 4-ary tuple $< t1, t2, p1, p2 >$, where each component is of type $\mathbb{N}$, and represents the number of cars or pedestrians that are approaching the crossing. For instance, *t1* describes the number of cars embarking the crossing in area *T1*. This information is then used to construct an architectural state, where movement of the cars/pedestrians is inhibited while a red signal is displayed to the cars/pedestrians, respectively. It is a straightforward matter to prove that this simulator fulfils the required axioms. The car positions and movements are given in module `Pelicon.State` of Appendix F by the following (mutual) functions. The pedestrian versions are similar but not presented. In the following, let i be a stream of inputs to the ladder, and *ui* be a stream of 4-ary tuples.

$$\begin{array}{lllll} \text{carmove} : \mathbb{N} \to \text{Area} \to \text{Area} \to \mathbb{N} & & & & \\ \text{carmove } 0 & \_ & \_ & = & 0 \\ \text{carmove } (\text{suc } t) & T1 & MUX & = & \textbf{if } \text{traffic } (\text{nthState } i\ t) \\ & & & & \textbf{then } \text{carpos } t\ T1 \textbf{ else } 0 \\ \text{carmove } (\text{suc } t) & T2 & MUX & = & \textbf{if } \text{traffic } (\text{nthState } i\ t) \\ & & & & \textbf{then } \text{carpos } t\ T2 \textbf{ else } 0 \\ \text{carmove } (\text{suc } t) & MUX & T1 & = & \text{carmove } t\ T2\ MUX \\ \text{carmove } (\text{suc } t) & MUX & T2 & = & \text{carmove } t\ T1\ MUX \\ \text{carmove } (\text{suc } t) & \_ & \_ & = & 0 \end{array}$$

$$
\begin{array}{lll}
\text{carpos} : \mathbb{N} \to \text{Area} \to \mathbb{N} & & \\
\text{carpos } 0 \quad \_ & = & 0 \\
\text{carpos } (\text{suc } t) \quad \mathit{T1} & = & \text{carpos } t \ \mathit{T1} + (\mathit{ui}\ t)_{t1} - \text{carmove } (\text{suc } t)\ \mathit{T1}\ \mathit{MUX} \\
\text{carpos } (\text{suc } t) \quad \mathit{T2} & = & \text{carpos } t \ \mathit{T2} + (\mathit{ui}\ t)_{t2} - \text{carmove } (\text{suc } t)\ \mathit{T2}\ \mathit{MUX} \\
\text{carpos } (\text{suc } t) \quad \mathit{MUX} & = & \text{carpos } t\ \mathit{MUX} + (\text{carmove } (\text{suc } t)\ \mathit{T1}\ \mathit{MUX} \\
 & & \quad + \text{carmove } (\text{suc } t)\ \mathit{T2}\ \mathit{MUX}) \\
 & & - (\text{carmove } (\text{suc } t)\ \mathit{MUX}\ \mathit{T1} \\
 & & \quad + \text{carmove } (\text{suc } t)\ \mathit{MUX}\ \mathit{T2}) \\
\text{carpos } (\text{suc } t) \quad \_ & = & 0
\end{array}
$$

It still remains to show that the ladder logic program fulfils the signalling principle. This is done by defining a less abstract instance of the signalling principle that can be represented in terms of the ladder logic program, i.e. a verification condition. One such condition is

$$\neg(\text{var } \mathit{tlight}.g) \vee \neg(\text{var } \mathit{plight}.g)$$

which is provable for the ladder in Figure 1.2. From this condition it is obtained that:

$$\forall i\ t\ .\ [\![\ \neg(\text{var } \mathit{tlight}.g) \vee \neg(\text{var } \mathit{plight}.g)\ ]\!]_{(\text{nthState } i\ t)}$$

The safety principle (pelicon-principle) follows by this because at time t without loss of generality assume the left disjunct. By unfolding the definition of carmove and traffic, it follows that

$$\text{carmove } t\ \mathit{T1}\ \mathit{MUX} \equiv 0 \wedge \text{carmove } t\ \mathit{T2}\ \mathit{MUX} \equiv 0$$

In Section 10.6 the ladder program of the Pelicon crossing is compiled, and executed. For now the interested reader can see the result of the simulation in Appendix C, and the implementation in modules `Pelicon.Simulator` and `Pelicon.SimulatorFull` of Appendix F.

Remark

The proof that the Pelicon crossing is safe demonstrates how the verification framework presented in this thesis can easily be applied to other domains. In such situations, what has previously been referred to as signalling principles for the train domain, are synonymous with safety principles in general.

10.5 Industrial Test Cases

Throughout the project, two interlocking systems were investigated. These interlocking systems were of substantially different design, and technology. The first was Gwili Steam Railway's (GSR) interlocking system; it is a mechanical interlocking of traditional design (levers, rods and bars). This interlocking is the subject of Chapter 11. The second interlocking was of a London Underground (LU) station; it is programmed using ladder logic and is the subject of this section.

10.5.1 A London Underground Station

Remark

The project sponsor requested that the name and specifics of the underground station is redacted from publication. For this reason, no detailed information is given relating to this interlocking, also the Agda code relating to this station is not appended.

What can be said is that the verification proceeded in a similar manner to the GSR verification. This includes using the same data-structures, signalling principles and code-base. The only difference, apart from the size and complexity of the systems was that GSR interlocking is defined by a locking table (see Chapter 11), which had to be translated into ladder logic.

Scenario

The interlocking controlled a terminal station (cf. Figure 10.1) consisting of two lines, and two platforms. Each of the lines is directed, and either brings trains into the station or takes trains out of the station. There are also two sets of points; these allow trains to change between the lines, that is to allow trains to arrive on one line, and leave on the other line after stopping at an available platform. There are also a number of signals that protect the sets of points and platforms.

Although the topology appears simple (especially when compared to stations with many platforms), the interlocking system was not so simple. This is because when the interlocking system was designed/installed there were a number of planned improvements to the line. These improvements were planned in three phases. Also, due to the cost of replacing the interlocking for each phase, it was decided that the interlocking system should be

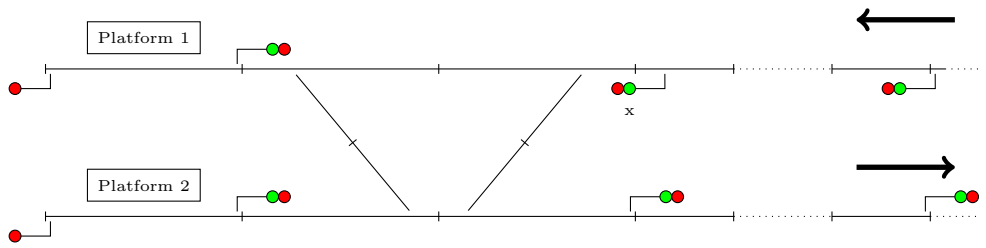


Figure 10.1: An illustration of the topology controlled by the interlocking system. The dotted lines on the right-hand side indicate lines outside of the interlocking systems control. The dotted lines located right of centre indicate a number of contiguous linear track segments. The arrows on the right represent the intended direction of travel on these lines. The signal marked with an 'x' has two routes associated with it; it also has a route indicator, not depicted, to indicate to the train driver which route the proceed aspect, when displayed, belongs to.

compatible with all three phases. In essence the ladder logic, consisted of three programs, where the selection between these programs was by an input variable. Correctness of this input is enforced by keeping the selection under lock-and-key, and thus this input can only be changed by the maintenance crew after completion of a phase.

In addition, there are three more modes that determined who had control. In this sense control means who/what can request or cancel a route. The choice of the controller is central, emergency or maintenance. Central indicates that the interlocking is being controlled by the central control room (usually a computer) and is the normal mode of operation. The emergency mode is triggered locally on the train station by a person activating the emergency stop plunger. When the plunger is triggered the interlocking tries to put itself, and the topology into a safe configuration[4]. The final mode is used by the station supervisor, and activated by a key-switch; it allows the station supervisor to override the interlocking, for example, in the case of faulty hardware.

As an example of the complexities of the interlocking, it is now considered how a route is selected. The process is initiated by the control system requesting a route, and then the interlocking determines whether it is safe to select it with respect to the control table and topological state. This process is described subsequently, but first the relevant variables are introduced.

[4]The trains are also informed that they must stop by other systems not considered here.

Note that input variables are denoted by italic, and state variables by roman font faces. The first three variables are general for the interlocking system:

***cent*:** Central has control.

***mtce*:** Maintainer has control.

***emergency*:** Emergency plunger has been activated.

The remaining variables are per route:

***request*:** The route has been requested by the central or maintenance controller.

***erequest*:** The route has been requested by the emergency sub-system.

***cancel*:** The route has been requested to be cancelled by the central or maintenance controller.

***ecancel*:** The route has been requested to be cancelled by the emergency sub-system.

requested: Internally the route has been marked as requested.

cancelled: Internally the route has been marked as requested to be cancelled.

called: The route has been requested and its constraints from the control table are fulfilled.

notinuse: Indicates whether the route has been internally allocated, does not reflect whether the route has been cancelled.

selected: Indicates whether the route has been internally allocated, a train may or may not be occupying the route.

available: Constraints in the control table for the route to be selected are fulfilled.

The process of setting a route is described by the following annotated equations that relate to rungs of a ladder logic program.

$$\text{requested} := (\mathit{request} \wedge (\mathit{cent} \vee \mathit{mtce})) \vee (\mathit{erequest} \wedge \mathit{emergency})$$
$$\text{cancelled} := (\mathit{cancel} \wedge (\mathit{cent} \vee \mathit{mtce})) \vee (\mathit{ecancel} \wedge \mathit{emergency})$$

Initially the inputs are inspected to determine whether the route has been requested, or whether it has been requested to be cancelled (un-select). The interlocking then computes 'called' which determines whether the request to select the route should be taken seriously.

$$\text{called} \quad := \text{requested} \wedge \text{available} \qquad (*)$$

Determining whether a route is in use, is by the following assignment. It only determines whether the abstract concept of a route is in use, not whether the route is occupied.

$$\text{notinuse} \quad := \neg\, \text{called} \wedge (\neg\, \text{selected} \vee \text{notinuse})$$

It is now computed whether the route should be selected, this does not imply a proceed aspect, but is a necessary condition. The proceed aspect also requires that the track segments are unoccupied.

$$\text{selected} \quad := \text{notinuse} \wedge \text{available} \wedge (\text{called} \vee (\neg\, \text{cancelled} \wedge \text{selected}))$$

Before the final rung that determines for the next cycle of the ladder whether the route is available to be selected, the rest of the ladder is executed. The remainder of the ladder will configure sets of points and set signal aspects. Thus the value of 'available' will be up-to-date in the next cycle, and its definition is not part of the initialisation where the inputs are inspected. Although it should be noted that $(*)$ will also make fresh checks to ensure that the detected configuration of relevant sets of points are valid.

$$\text{available} \quad := \text{constraints from control table for route to be selected}$$

Remark

Although the above example is correct, it is an abstraction. There are more corner cases relating to which phase the interlocking is currently in; also the initialisation of the variables is overly simplified. Nonetheless, it is clear in general that at least 10 variables are required for the logic to select a route.

Similar to the above process of selecting a route, positioning a set of points is a complicated process. For instance, the input to a set of points is given

by three variables indicating for a three-way switch which contact is closed. This three-way switch is for maintenance mode where the set of points is controlled to the normal or reverse position, the third position indicates that the maintainer did not request either position. Also, there are two detection inputs that determine if the set of points is detected in the normal or reverse positions. There are a further 21 state variables (and associated rungs), and 9 input variables for each set of points. The logic for a set of points firstly decides, after the routes have been selected, which position the set of points should be in, this also considers the maintenance mode inputs. Then it determines whether the set of points needs to be moved, if it does, then the move request is issued. It will then continue executing cycles of the ladder until the set of points is detected in the correct position, from which the route availability is computed.

Up to now, all reference to ladder logic programs has carefully omitted the issue of measuring time. For example, consider the situation when moving a set of points, it might be the case that the blades of the points become jammed, perhaps by a rock getting in the wrong place. In such situations, it is required that if the set of points is not detected in the correct position within a given time (e.g. 7.5 seconds), then the power to the points-machine is turned off. The model of ladder logic does not allow for such rules, but the interlocking system is equipped with timing modules. These modules support a number of timers; each timer is interfaced with the ladder logic by two allocated variables. One of which is a state variable called *trigger* and one is an input called *elapsed*. The intuition being that when a time duration is required, the *trigger* variable is set, which indicates to the timer module that it should start the associated timer. If *trigger* is unset before the time duration has elapsed, then the timer is reset. Otherwise, when the time duration has elapsed, and *trigger* is still set, the timer module sets *elapsed*. See Figure 10.2 for the behaviour of these two variables plotted over time. There are 2^2 possibilities of these variables, but one of these possibilities is not well-formed, namely when $elapsed \wedge \neg trigger$. The verification performed did not require special treatment of the timers, and even the ill-formed combinations were not catered for. It is assumed that a fully validated verification, which reasons about timed events, would require timers to be specially treated. The interlocking system contained 49 timers.

Verification

The ladder logic was formalised as described in Chapter 9, and the topology and control table were formalised as described in Chapter 8. Defining a map between the interlocking state and the architectural state required a

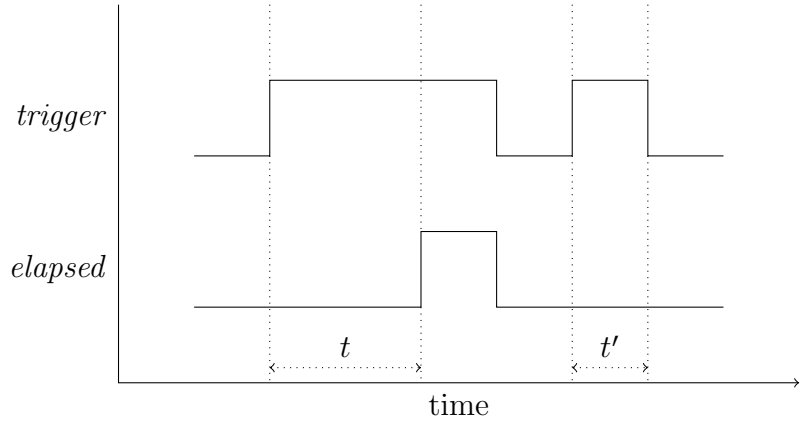


Figure 10.2: Timer variable interactions plotted over time, where t denotes the timer's duration and $t' < t$. The upward portions of the plots indicate the variable being assigned true, and the downward portions indicate the variable being assigned false.

significant amount of research. As already indicated, each set of points, signal and route have many associated interlocking variables, each with a slightly differing meaning. In some cases, a value of the architectural state depended upon multiple variables of the interlocking. For example, the signal marked with an 'x' in Figure 10.1 has two routes associated with it. Determining which route its aspect corresponded to was achieved by inspecting the route indicator output.

To prove that the interlocking system is safe, it is required to provide a proof of the 4 signalling principles (from Chapter 8). In the most part, proving that the signalling principles held for the LU station was successful. Although some of the initial attempts were problematic, after examining the satisfying assignment, structure of the ladder and semantics of the variables, these problems were mitigated. This was in part due to incorrect maps from the physical layout into variables, which were amended. One example of this was when determining what position a set of points was in: Initial attempts used variables that depend upon inputs which detected the physical position of the points combined (and'ed) with the position they are being controlled into. However, it was found that the signalling principle 4 (train holds lock) could not be proved because the train detection inputs to the system were unconstrained. That is, assume a route rt and a set of points pt in the route, then the condition which could not be proved is as follows:

$$\text{occupied } rt \Rightarrow \text{locked } pt \wedge \text{detected } pt \Rightarrow (\text{locked } pt)' \wedge (\text{detected } pt)'$$

where detected pt denotes an input variable, and the primes are the values in the next state. This condition is not provable because the value of $(\text{detected } pt)'$ is not determined by the interlocking system. Subsequently the state map was amended to only inspect the variables representing the position the set of points was being controlled to, this allowed the principle to be proved. That is

$$\text{occupied } rt \Rightarrow \text{locked } pt \Rightarrow (\text{locked } pt)'$$

which essentially means that while rt is occupied, the interlocking does not attempt to change the position of pt. It is noted that this is safe as a route can only become occupied after the signal protecting the route is set to proceed, which requires that the set of points is also detected to be in the correct position, this is implied by the previous verification where the interlocking system is proved to be a correct refinement of the control table.

The only signalling principle that did not universally hold was principle 4 (train holds lock). It could be violated if the interlocking system was placed into maintenance mode and the set of points was released from the interlocking's control. Maintenance mode is enabled by an input to the interlocking being set to true, and thus it is assumed that this input is never set to true for the verification to succeed. Although this weakens the safety, it is not a serious issue because by design, when in maintenance mode, the interlocking system's decisions can be overridden by an expert. The expert would assume responsibility for their actions.

The interlocking system contained ≈ 330 rungs, and 250 input variables. When expanded using our inductive verification technique, we obtained a Boolean formula with ≈ 1500 variables (in the inductive step). These formulæ could be solved very quickly by the SAT solver, but generating them took a couple of minutes. This time is due to (1) constructing the transition formula by building up a list of processed rungs (and shifting the variables accordingly), and (2) shifting the variables of all the formulæ between the cycles.

Figure 10.3 shows a breakdown of the time spent on different tasks while proving a typical condition. The total time taken for the proof seems large, especially when it is considered that the actual SAT solver spent less than 0.5 of a second running. This is due to inefficiencies in Agda. For instance, due to lack of sharing inside Agda, the SAT solver is evaluated twice for each problem set. The transition formula is generated a number of times, e.g. when checking the bounds and generating the proof obligations. Also, the GHC run-time memory management algorithm is not well suited to modifying large data-structures because every data-structure is immutable, thus for every

Task	Time (Seconds)
Initial Type-Checking of Ladder	377.84
Generate Initial Condition	23.28
Generate Inductive Condition	48.05
Execute SAT Total (Initial)	31.13
Execute SAT Solver (Initial)	0.05
Execute SAT Total (Inductive)	64.33
Execute SAT Solver (Inductive)	0.08
Generate Transition Formula	24.03
Checking Bounds	21.39
Total for Proof	518.07

Figure 10.3: Breakdown for running a selection of tasks while verifying a typical condition for the LU interlocking. Z3 [dMB08] was the SAT solver used. All times were gathered by running Agda from Emacs, with `:set +s`, then evaluating the individual functions. The SAT solver times were gathered from the output of the solver. The total SAT time refers to the time taken in Agda to execute the solver, this includes generating the formulæ.

update to a data-structure in memory, the original is duplicated. Therefore, the maps and shifting of variable indices in large Boolean formulæ[5] resulted in undesirable behaviour from the underlying memory subsystem.

Type-checking of the ladder is time consuming, but is a one-off task as the checked files are cached. Initial type-checking includes defining two sets (data-types) of variables, one for the state variables and one for the input variables. It also includes type-checking maps from these variables into natural numbers, the rungs (as a list) and an initial state (as a list), and then type-checks a Ladder data structure constructed from these components.

10.5.2 Remarks

Applying the verification framework built-up over the course of this thesis was not difficult. Much of the hard work was in crafting the framework with a simple user interface. For instance, ladder logic programs are given by a simply-typed definition, and topology models are easy to instantiate, and the proofs are given by automated tools. It is important to understand what the goal to be proved is. When proving the signalling principles the human should guide the proof, and know which cases are expected to be trivially provable and which others follow by applying the ATP tool.

[5] Agda terms are represented in Haskell by a list of lists.

In total, it required approximately 2 weeks work, from start to finish, to verify the LU interlocking. This entailed instantiating the various definitions, providing proofs about them and applying the ATP tool. The work was undertaken after verifying that GSR was safe, which was a smaller exercise. The GSR verification was used to fine-tune the framework, and as such from start to end took significantly longer.

It is noted that when defining an instance of the topology model, considerable care should be given to the definitions so that when proving the signalling principles, the case-distinctions (on the topology) purvey human readable information. Contrary to this, the early attempts to define components of topology models (and ladder logic variables) used the enumeration sets Fin. This was to simplify the definitions and to obtain decidable equality for free. However, proving properties about these definitions, such as a signalling principle would result in extremely large, hard to read case-distinctions. The issue here is that one would have to count the number of suc's and manually validate the models are correct, and in the case of an erroneous definition, it can be hard to identify. Similar problems with ladder logic were encountered, where to identify a variable, one would have to count the (possibly many hundred) applications of suc. There was also an issue with efficiency as the type-checker would infer the hidden terms. It is recommended to use a non-recursive data-type, where the constructors have meaningful names. This vastly simplifies identifying problem cases and validating the models.

More generally, there is an issue of understanding what has actually been verified. Not only do the models need to be validated, but also that the interlocking is correctly represented in Agda and that the mappings between the variables of the interlocking and the architectural state are correct. Should one of these projections be incorrect (e.g. referencing variable 145 instead of 154), then performing the verification does not result in a proof of safety, however, it would require an astute eye to catch mistakes such as this.

It was found that a number of definitions, such as the function nthState, had to be made abstract[6] to prevent unnecessary normalisation. Particularly any references to the function nthState would normalise the ladder, resulting in a serious efficiency problem. For example, there were occasions when type-checking simple lemmata that depended upon (indexed by) a ladder would take over 30 minutes due to overly eager normalisation. However, after carefully abstracting definitions the time required to type-check these problematic definitions was significantly reduced to a couple of minutes. One such lemma that was problematic was applying a proof of the correctness of

[6]By the use of abstract blocks in Agda.

the ladder to another proof of correctness of the ladder, that is for a concrete ladder l and two formulæ φ and ψ, to prove the following lemma:

$$\text{LadderCorrect}_l\ \varphi \to \text{LadderCorrect}_l\ (\varphi \Rightarrow \psi) \to \text{LadderCorrect}_l\ \psi$$

This lemma is trivial to prove on paper as a human would not attempt to normalise l. However, if proved for a concrete ladder in Agda, then l is normalised; depending on the size of l, it could suffer efficiency problems. Of course, this lemma can be proved in general, in Agda efficiently, but once it is applied to a concrete ladder the original situation of excessive normalisation is re-encountered. One solution to this problem was to define in Agda an abstract version of the ladder as follows:

```
abstract
  rungs′ = rungsₗ
  initialstate′ = initialstateₗ

l′ : Ladder
l′ = ladder statevarsₗ inpvarsₗ rungs′ initialstate′ inp-correctₗ

abstract
  l-eq : l′ ≡ l
  l-eq = refl
```

Then we worked with the ladder l' in-place of l as much as possible as the rungs and initial state will not normalise. However, l' is still usable in the majority of situations as the number of state and input variables (and inp-correct) are obtainable. Hence the type of the states, and types of inputs are concretely derivable for l' and are definitionally equal to those of l.

Moreover, the only time that the ladder needs to be fully normalised is when it is compiled and executed, problem sets are generated for the external tool, or checking that the ladder is well-formed. In these cases, it is possible to normalise the ladder by forcing Agda to unfold the definitions using l-eq.

10.6 Extracted Control Systems

The verified ladder logic interlocking systems can be compiled and executed as a standalone interactive program. Therefore, fully verified code is obtained, and the whole process is a full example of the proofs-as-programs paradigm. That is, proofs are programs, and proofs are written in the same language as programs.

The extraction is performed by Agda, using the MAlonzo back-end. MAlonzo translates Agda code into Haskell code. The result of the extraction is

a number of Haskell files, which are then compiled by GHC. Due to Agda being dependently typed and Haskell being simply typed, a direct translation requires that type coercions are inserted into the translated program. This is safe as Agda has type-checked the code. However as a result of these coercions, many optimisations that would normally be applied by GHC are not carried out. A good discussion about the benefits and limitations of different code generation back-ends for Agda can be found in [FG11, Tur10].

Remark

No external tools (e.g. SAT solvers) are called upon during execution of the ladder, all verifications (including calling an external tool) are performed in Agda during type-checking. It is the type-checked term that is extracted into a program.

The extracted control systems are intended to be used for simulation purposes; this is because the code generation and compilers have not been certified. There is also a potential issue of running a live critical system under a garbage collected environment, such as GHC, because there are no assurances that a resource would not be exhausted. If such a situation did occur, then the state that it crashes in would be safe (as the system has been verified).

Remark

To mitigate the constraint that the extracted program is only for simulation purposes, and as future work, it would be possible to define a small, safe subset of a language such as C, in Agda. This subset would need to include the logical operators for conjunction, disjunction and negation, it would also need to define simple while loops, Boolean variables and assignments to these variables. It might also be required to ensure that the subset includes standard IO functions. It would then be a simple matter to translate the representation of a Ladder in Agda into a representation of a C program, and prove the correctness of this translation. Furthermore, converting this representation of a C program into an actual C program would be canonical, i.e. a toString function. Provided this program is compiled using a verified C compiler, the resulting executable would be a fully safety verified ladder logic program that could be used in life critical situations.

The simulations are basic. They will start at the initial state, then request an input from the user, transition into the next state, and repeat. When implementing the simulation, there is a choice about which transition system to execute.

Ladder. Simulating only the ladder logic, where the states and inputs are given by lists of Booleans of the correct length. Although this type of simulation is sufficient to test the interlocking, not enough information is given to derive the architectural states. In effect, simulating an interlocking using this method does not consider the positions of trains; but it does consider the states of signals, sets of points and track segments. This type of simulation is ideal if there is already a test-bed setup (as found in industrial development environments) so that the interlocking system can be plugged-in.

Architectural. Building upon the previous ladder logic simulation, it is possible to augment the transition system to, in addition, simulate where the trains are (in the case of the railway domain). In this case, the transition system that is executed consists of an augmented ladder. The inputs are given by the inputs to the ladder and the train inputs; train inputs are defined in Section 9.3.1. The states are given by the states of the ladder paired with the train positions.

Provided that the ladder logic is well-formed, and hence decidable, producing a transition function for the ladder logic is explained in Section 9.3. For the architectural simulation, it is also required to have a transition function that depends upon the ladders transitions, that is the trains positions depend upon the signal aspects projected from the ladder state. This architectural transition function should have been defined already when defining the n^{th} architectural state during verification. Therefore, the extracted programs are straightforward interfaces to the underlying data structures previously described throughout this thesis. There is one caveat: If the inputs to the ladder logic must fulfil an invariant, then this invariant must be enforced by the simulation.

For simple ladders, such as the Pelicon crossing or Gwili, the program's user interface is by means of a terminal application. The application will pretty print the state and request the inputs one-by-one. See Appendix C for an output of the simulation for the Pelicon crossing. For larger ladders, this becomes cumbersome, and time consuming for a human to understand. Therefore, it is possible to write a user interface in another language that will graphically depict the state of the topology, and provide a usable method of entering the inputs. This interface then executes the extracted ladder by

treating it as a stream transformer, that is from an input stream to an output stream.

Due to the infinite loop that ladder logic is evaluated within, special treatment was required to write the interactive program such that it would pass the Agda termination checker. This was achieved by using Anton Setzer's IO library [Set09]. The interactive programs are encoded into a co-algebraic data-type, which is then translated into an IO program by the library. Inside the library, the termination-checker is disabled in a safe way.

Chapter 11

Gwili Steam Railway

On 28^{th} October 2011, a visit was made to Gwili Steam Railway (GSR) with the ambition to use it as a case study for the railway verification framework explained in this thesis. It should be noted that Gwili Steam Railway is independent of the project sponsor, Invensis. It was a useful, and hands-on, experience that helped to sharpen our interpretation of the railway domain. It gave us the opportunity to understand from a practical perspective the requirements of the domain, a level of understanding that is hard to obtain from reading books alone. The verification proceeds along the lines of formalising the topology, interlocking system, and control table; and then proving that it fulfils the four required signalling principles to obtain a proof that trains do not derail or collide.

Remark

For historic railways, such as Gwili Steam Railway, the same safety requirements as modern railways are taken. This is because the roots of modern railway signalling are found in historic railways, the only major difference is the speed that the events occur.

Chapter Overview. In Section 11.1 the GSR scenario is introduced and explained. Alongside the introduction, formal definitions of the topology, control table and interlocking system are presented. As the interlocking system is mechanical, it was not realised using ladder logic, which the verification framework is defined over. It was decided to translate the locking table[1] into ladder logic (Section 11.1.2), rather than redefine the semantics

[1] A document describing the mechanical interlocking between levers of a mechanical interlocking system.

Figure 11.1: Gwili Railway Layout

of the framework.

In Section 11.2 the interlocking is verified to be a correct refinement of the control table, and that it fulfils the required signalling principles from Chapter 8. Finally in Section 11.2 the interlocking system is compiled and executed to validate that the interlocking is correct. The full code listing can be found in Appendix F.

11.1 Scenario

GSR is a small railway located in Carmarthenshire, South Wales, UK; it is maintained and operated by a team of volunteer rail enthusiasts as a tourist attraction. It was originally part of the Carmarthen to Aberystwyth line, which was opened by the Carmarthen and Cardigan Railway Company in 1860. The line changed ownership many times before being decommissioned in 1965, and finally dismantled in 1975. Soon after, (in 1977) the Gwili Railway company was formed to preserve an 8 mile stretch of the line (Figure 11.1), since these humble beginnings their ambitions have grown, at no time more so than after a 2009 merger with another local railway society (Swansea). Currently (at time of writing) the GSR is laying new track and will build a new station juxtapositioned with Carmarthen. It should be noted that the models and verifications described here relate to the unmodified railway.

The railway consists of three stations connected by a single track. The station, Bronwydd Arms (Figure 11.2), is the main station, it is the only part of the railway that is protected by an interlocking system. However, it should be noted that there are a small number of ground frames situated throughout the railway. The other two stations are within the same block segment, and not controlled by the interlocking. The interlocking is contained inside the signal box (Figure 11.3); the levers are located on the first floor (Figure 11.4), and the interlocking is located on the ground floor (Figure 11.5). The interlocking consists of three locking trays, and each tray has

Figure 11.2: Terminal view of Bronwydd Arms Station. From the level crossing to behind the camera's view is the beginning of the new line.

a number of *tappets* (steel blades) which are directly connected to the levers (at most one tappet for each lever in each tray), and slide back and forth as the lever is moved back and forth. Each tappet has a number of ports (notches) carved into their edges at right-angles to the tray. As the tappets move, they push *dogs* (a special nut and bolt) along the tray. It is when these dogs are pushed and constrained into the port of a tappet, that the tappet (hence lever) is locked. In Section 11.1.2 more information is given about the functionality of these interlocking systems, and how to formalise them. Details of mechanical interlocking systems, and a discussion of how to test them can be found in Woodbridge's notes [Woo12].

11.1.1 Layout

Viewed from inside the signal box there is the scheme plan in Figure 11.6 that informs the signalman of the topology which is controlled by the interlocking, and which lever controls, which piece of hardware. There are two sets of points, 13 and 14, that switch trains between the "MAIN" and "LOOP" lines. Passenger services are only allowed to use the "MAIN" line. This

Figure 11.3: Bronwydd Arms Signal Box

Figure 11.4: Top side of lever frame.

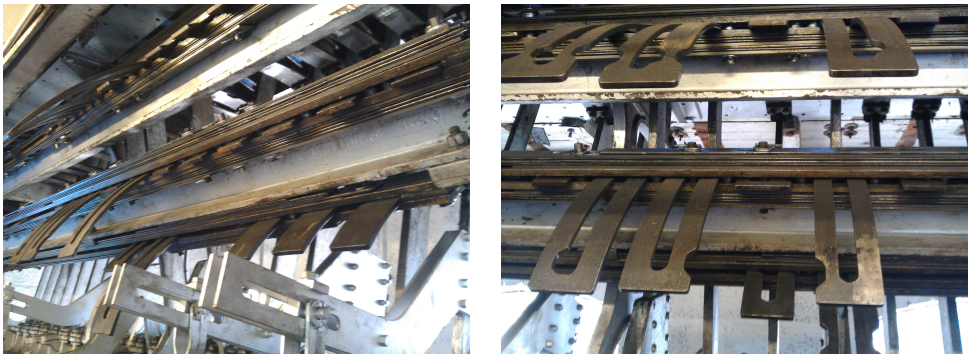


Figure 11.5: Under side of lever frame.

is enforced by the interlocking and signalling scheme—as passenger services must only obey main signals and not the shunt signals. For instance, a passenger service approaching from the left can only enter the station if signal 4 is clear, and not C2 or 6. There are 11 signals that are controlled by the interlocking, the majority of which are shunt signals. The scheme plan is formalised by Figure 11.7. In the scheme plan (Figure 11.6) there are more tracks than in Figure 11.7, this is because part of the railyard is not controlled by the interlocking and instead controlled directly by ground frames. Note that track segment 8 (in Figure 11.7) is not part of any route, and not safety controlled by the interlocking.

In reference to Figure 11.7. The dots on the left indicate a single block segment protected by signal 16 (spanning to and including the next stations) that only one train is allowed to occupy at a time. The numbers relate to the controlling lever, i.e. when lever 3 has not been pulled, signal 3 shows the danger aspect, and when lever 3 is pulled, signal 3 shows a proceed aspect. The precise differences between the three types of signals, main, shunt and call-on (e.g. signals 3, 17 and C5) are not relevant for this verification, and they are treated as main signals. With respect to safety, it is clear that signals 4 and 19 should never both be set to proceed at the same time, this is because the portions of track they protect are not disjoint, similarly for signals: C5, 7, 18, 20.

A further discussion of the signals and sets of points is required before giving formal definitions of the topology. The three types of signal in-use are considered. The main signals are of a traditional semaphore design, and the shunting signals are small (approximately 1m in height) that are located on the ground. Both of these types of signal show only proceed or danger aspects. See Figure 11.8 for an image of these signals. There is also a third type of signal that is prefixed with the letter C, these are call-on

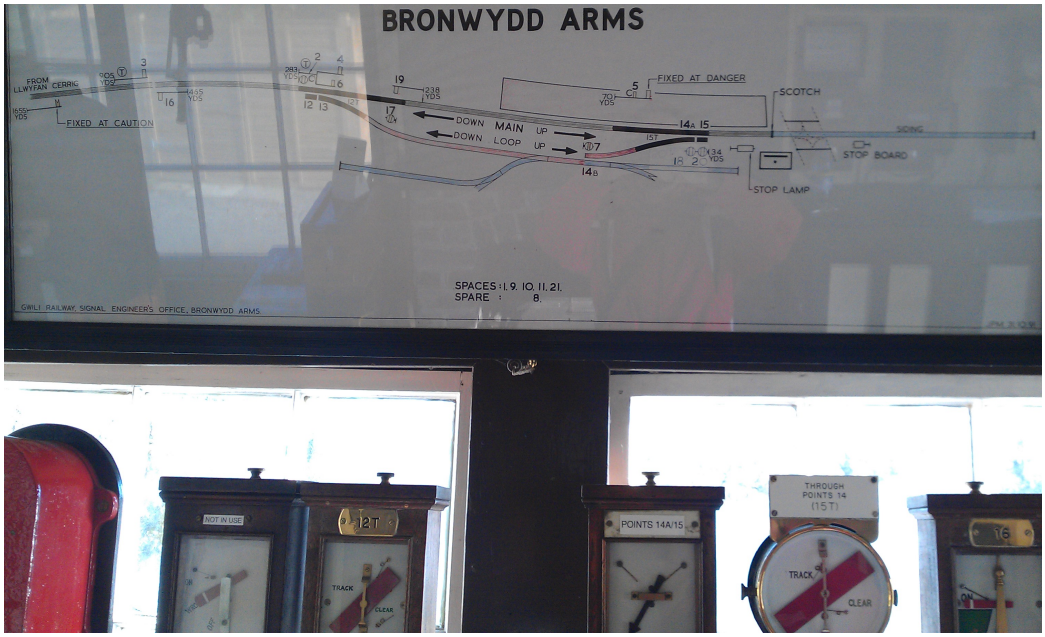


Figure 11.6: Scheme plan from signal box. Relates levers to the hardware they control.

signals and are smaller, subordinate versions of the semaphore signal. The distinction given here is not universal, and it is possible to have shunt signals implemented by a small semaphore signal. All these signals have electric lights in them to aid with poor visibility. When a route is set, it is required that the signalman knows that the light is fully functioning. This is known as lamp proving, which in modern interlocking systems is by an input to the system indicating whether the bulb has blown. For mechanical interlocking systems, the lamp proving is achieved in two ways, first the signals that are clearly visible from the signal box are proved by eye, and secondly for distant signals a system of relays are used, such that, should the bulb blow, then an audible alarm is raised in the signal box.

The two sets of points have facing-point locks, which are achieved by a metal bar (the lock) being inserted into a cutout on the stretcher bar of a set of points (Figure 11.9). This lock not only holds the set of points in position, but will detect if the set of points is not in a valid position. That is, if the set of points is not fully controlled in to a valid position, perhaps because of wear-and-tear, then the cutout will not line up with the lock and cannot be engaged, as a result, any signals requiring the points to be locked will not be able to be cleared. At Gwili, the cutout had 2mm play (so the set of points could be at most 2mm out of alignment). This is contrary to

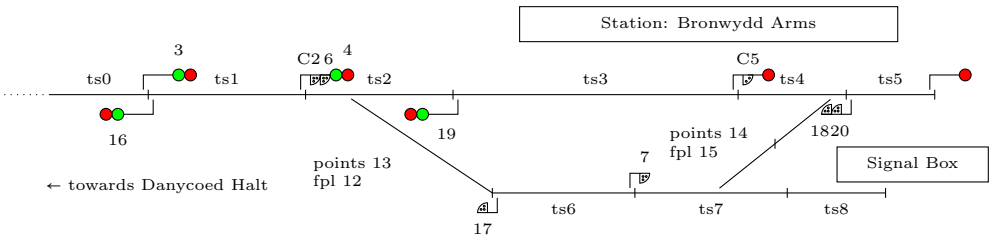


Figure 11.7: Bronwydd Arms topology. See the accompanying discussions in the text for information about this diagram. For now, it is noted that a slightly different notation is used for this image. This notation allows for multiple signals to be composed onto the same *signal post*, e.g. C2, 6 and 4 are three distinct signals located at the same position; C2 is a call-on, 6 is a shunt and 4 is a main signal.

modern systems where the lock (nowadays electronic clamps) and detection are separate systems from the interlocking systems perspective.

Formalising Topology

On the scheme plan in Figure 11.6, there are no clear track segments. This is because track segments are only required to occur explicitly in modern interlocking systems that are equipped with track circuits for train detection, see Appendix B for details. So the first step to formalising the topology, is to divide the track up into discrete segments. This has been done (by myself) in Figure 11.7, where there are 9 segments. As there are only 9 segments they are represented using the set Fin 9, and are numbered horizontally, starting at the top left, finishing bottom right. The connections between these segments are given by the following function:

$$
\begin{aligned}
&\mathrm{gwiliReachable} : \mathrm{Fin}\ 9 \to \mathrm{List}\ (\mathrm{Fin}\ 9) \\
&\mathrm{gwiliReachable}\ 0 \ = \ [1] \\
&\mathrm{gwiliReachable}\ 1 \ = \ [0, 2] \\
&\mathrm{gwiliReachable}\ 2 \ = \ [1, 3, 6] \\
&\mathrm{gwiliReachable}\ \vdots \ = \ \vdots \\
&\mathrm{gwiliReachable}\ 8 \ = \ [7]
\end{aligned}
$$

The rightmost signal (cf. Figure 11.7) is ignored in the models (and verification) as it is fixed to the danger aspect, and not controlled by the interlocking. The set of signals is given as

$$\mathrm{Signal} := \{2, 3, 4, 5, 6, 7, 16, 17, 18, 19, 20\}$$

Figure 11.8: Different types of signals in use, left is a semaphore signal (19) and right is a shunting signal (17).

Figure 11.9: Facing-point lock is clearly visible in centre of image, the bar parallel to the tracks slides back and forth to lock/unlock the set of points. The bar orthogonal to the track is called the stretcher bar, and has two cutouts (one for normal and one for reverse position) that the first bar will slide into/out of.

which is represented in Agda using a (non-recursive) data-type. It was chosen to define these identifiers using a representation that preserves the names of the signals, as opposed to using a generic enumeration set such as Fin 11. Preserving the names will subsequently aid the user for future definitions, proofs and validation.

The position of the signals is given by the following function, which determines the segments before and after it, and a proof that they are connected.

```
gwiliSignals : Signal → SignalLocation_{Fin 9,(λa b . b isin gwiliReachable a)}
gwiliSignals 2  = record {
                    facing     = 1              ;
                    trailing   = 2              ;
                    connected  = inj₂ (inj₁ refl) }
gwiliSignals 3  = record {
                    facing     = 0       ;
                    trailing   = 1       ;
                    connected  = inj₁ refl }
gwiliSignals ⋮  =                 ⋮
gwiliSignals 20 = record {
                    facing     = 5       ;
                    trailing   = 4       ;
                    connected  = inj₁ refl }
```

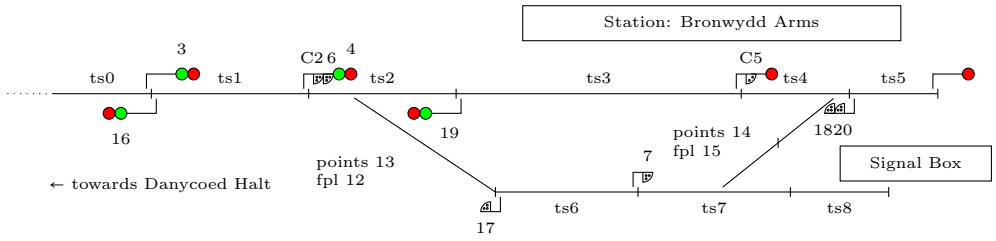
The physical layout (see page 165) of Gwili is then defined as follows:

```
gwiliPL : PhysicalLayout
gwiliPL  = record {
             Segment         = Fin 9           ;
             Signal          = Signal          ;
             connections     = gwiliReachable  ;
             signalLocation  = gwiliSignals    }
```

The control table entries (see page 169) are given by a function mapping routes to the entries. As each signal has exactly one route associated with it, routes are identified by the signals they start with. The control table is as follows (note that the sets of points are referenced by their containing track segments):

Start Signal	Segments	Normal Points	Reverse Points	Facing
2	2 , 3	2	–	2
3	1	–	–	–
4	2 , 3	2	–	2
6	2 , 6	–	2	2
5	4 , 5	4	–	–
7	7 , 4 , 5	–	4 , 7	7
16	0	–	–	–
17	2 , 1	–	2	–
18	4 , 7 , 6	–	4 , 7	4
19	2 , 1	2	–	–
20	4 , 3	4	–	4

To aid with the understanding of the control table, Figure 11.7 is repeated below:



This table is built out of control table entries, therefore it defines the function:

$$\text{gwiliEntries} : \text{Signal} \to \text{ControlTableEntry}_{\text{gwiliPL}}$$

Connections between these routes are defined using the same method that defines connections between track segments. That is

$$\begin{aligned}
&\text{gwiliRouteConnections} : \text{Signal} \to \text{List Signal} \\
&\text{gwiliRouteConnections } 2 \;\; = \; [5] \\
&\text{gwiliRouteConnections } 3 \;\; = \; [2, 4, 6] \\
&\text{gwiliRouteConnections } \vdots \;\; = \; \vdots \\
&\text{gwiliRouteConnections } 20 \; = \; [19]
\end{aligned}$$

To finalise formalising the control table, proofs that the routes are (1) well-formed and (2) non-empty, are required. In the case of (2), the proof follows trivially by the control table entries as all entries are non-empty under the segments heading. In the case of (1), first recall from page 171 that the well-formedness of the control table is informally defined as:

1.1) all pairs of routes that are connected to a common route share a track segment, and

1.2) all pairs of connected routes have their last and first track segments connected, respectively.

1.3) all segments in a route are continuous

1.4) all sets of points identified (normal/reverse/facing) are contained in the routes segments.

The proofs in all cases follow by the definition of GSR's control table; they can be found in Appendix F. In this case study, proving of these properties was manually performed and required case-distinction on the routes/signals, this was cumbersome, especially in the case of (1.1) as two of the three routes must be analysed, which results in 11^2 combinations. However, this manual proof would inhibit formalising control tables for large topologies. Provided the sets of routes and track segments have a decidable equality, then the properties (1.1) – (1.4) are decidable and could be automated by finite conjunctions/disjunctions of the routes and track segments (i.e. encoding the case-distinction into Boolean formula). There would also be a small number of proofs required to translate the result of the decision procedure into the formal definitions on page 171. This automation is left as future work.

For completeness the control table is formalised as follows:

$$
\begin{array}{l}
\mathrm{gwiliControl} : \mathrm{ControlTable}_{\mathrm{gwiliPL}} \\
\mathrm{gwiliControl} \;=\; \mathrm{record}\ \{ \\
\qquad\qquad \begin{array}{lcll}
\mathrm{Route} & = & \mathrm{Signal} & ; \\
\mathrm{routeSignal} & = & \lambda x \to x & ; \\
\mathrm{entries} & = & \mathrm{gwiliEntries} & ; \\
\mathrm{connections} & = & \mathrm{gwiliRouteConnections} & \}
\end{array}
\end{array}
$$

Note that the fields: NonEmptyRoutes, WellFormed, RoutesConnected, Continuous, InRoute1, InRoute2 and InRoute3; have been omitted as they are not enlightening, and as discussed above the proofs follow trivially from the control table.

The final step is to select the number of trains, then instantiate an abstract layout. For the verification performed, two trains were chosen as it is the smallest number that is interesting. The set of trains is represented using Fin 2, which is trivial to define decidable propositional equality for. Thus, by applying the function toLayout on page 173, to the physical layout, control table and set of trains, an abstract layout called gwiliLayout is obtained.

The above definitions of the physical layout, control tables and abstract layout are in module `Gwili.Layout` of Appendix F.

In subsequent sections, the ladder logic and state of the abstract layout are defined. Then the models are used to formulate the signalling principles during the verification of the ladder logic.

11.1.2 Translation of Locking Tables into Ladder Logic

In this subsection, it is shown using railway domain terminology how mechanical interlocking systems work. Then it is shown how the interlocking systems relate to locking tables, and finally the translation from these locking tables into ladder logic programs is given.

Mechanical interlocking systems consist of a number of levers (see Figure 11.4) that are moved backwards and forwards. These levers connect directly to blades in the interlocking system, so that when the lever moves, it moves a shaped metal blade called a tappet inside the interlocking. The shape of a tappet is specific to each lever; they can be intuitively thought of as a key in a traditional lock. If the tappet is locked in place (typically by the configuration of the other tappets), then the lever is also locked (i.e. cannot be moved).

Locking tables are documents that detail constraints that the interlocking system must fulfil. They are at a much lower-level than the constraints in the control tables, and specify the relationships/constraints between the levers, i.e. which levers lock/unlock other levers.

Remark

Nomenclature of lever frames, when a lever is pushed away from the signalman (as the majority are in Figure 11.4), it is said to in the normal position. When pulled towards the signalman, it is said to be in the reverse position. Interestingly this is the origin of the terms normal and reverse for sets of points.

A locking table defines a transition system. The states are given by combinations (normal/reverse) of the levers, and the transitions between the states are given by a singular allowed movement of a lever. The locking tables define constraints/invariants on the states and transitions.

The following convention is used to represent the position of a lever in the ladder logic: For a lever l, a corresponding Boolean value l in the state is identified, such that $\neg l$ represents that the lever is in the normal position, and l represents that the lever is in the reverse position. There are also occasions where the position of the lever in the next state is referenced, for

this a priming of the variables is used, i.e. l' represents the position of lever l in the next state.

To help explain the translation, a small example of a set of points, a facing-point lock and a signal is introduced in Figure 11.10. This example is easy to extend to the GSR interlocking as it is a microcosm with the same functionality. The intention is that the signal protects the set of points, and it will only be able to clear once the set of points is locked in the normal position. For information regarding the purpose of a facing-point lock, see Section 8.3.1.

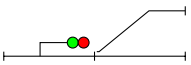


Figure 11.10: Locking table scenario.

In such a situation, there would be three controlling levers, p, l, and s, one for the set of points, one for the lock and one for the signal, respectively. It is required that when l has been pulled, p should not be able to be moved; also when l has not been pulled or p has, then s should not be pulled. Formally the following must always hold:

$$(l \wedge p) \to p' \qquad (l \wedge \neg p) \to \neg p' \qquad (\neg l \vee p) \to \neg s$$

Here, the first two constrain the transitions, and the last one constrains the state space. Constraints such as these are formalised in the locking table. Before explaining the locking table, we consider how a mechanical interlocking could be realised for this example.

See Figure 11.11 for a depiction of the interlocking. The interlocking consists of three tappets (coloured grey), one for each lever, similar to those in Figure 11.5. There are three dogs (special nuts and bolts) that form the interlocks, they are depicted by black circles. Two of the dogs (left and rightmost) are connected by means of a metal bar (depicted by a line) and move together. Note that all levers are shown in the normal position.

The interlocking from Figure 11.11 is fully functional. Enumerating its possible transitions (by moving levers) from the initial state gives rise to the transition system in Figure 11.12. (a) is the initial configuration where all levers are normal, that is the points are unlocked in the normal position, and the signal shows the danger aspect. From (a), it is not possible to clear the signal because the rightmost dog cannot move out of the notch on s. It is only possible to pull s after l has been pulled, that is from (a), to (d)

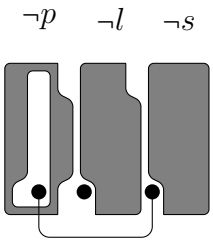


Figure 11.11: Example mechanical interlocking. In this diagram, the tappets move up and down, and the dogs move left and right. For instance tappet p is free to be moved into the reverse position, whereas tappet s is constrained by a dog and cannot be moved until l has been reversed.

then (e). Conversely from (e) it is not possible to reverse p without first normalising l, which requires that s is also normalised. Thus, regardless of the order the levers are used, it is never possible to clear the signal without the set of points being locked in the normal position. The configurations (b) and (c) consider when the points have been reversed.

Implicit in the transition system is the (provable) invariant on states:

$$(\neg l \vee p) \rightarrow \neg s$$

This invariant was not used directly (apart from requiring that the initial state fulfils it), but instead is translated into a dynamic constraint that restricts the transitions. That is the following invariants on the transitions are obtained from the above invariant (and its contraposition).

$$\neg l \rightarrow \neg s' \qquad\qquad s \rightarrow l'$$
$$p \rightarrow \neg s' \qquad\qquad s \rightarrow \neg p'$$

With this basic understanding of how mechanical interlocking systems operate, it is now considered what different types of interlocks are possible, and how they relate to invariants. The locking tables considered in this work specify three different types of locking between the levers. The locking tables define for each lever, (1) which levers are locked in the normal position when it is pulled, and (2) which levers are required to be pulled before it can be pulled. The final lock, (3) defines, which levers will lock another lever into both the normal or reverse positions. Locking (3) is required to prevent a lever changing position, for instance these locks are used to implement facing-point locks. For a given pair of levers a and b the three locks are formalised as follows

Locks Normal: When a is pulled it will lock b in the normal position. Conversely a will only be free to be pulled once b is in the normal position.

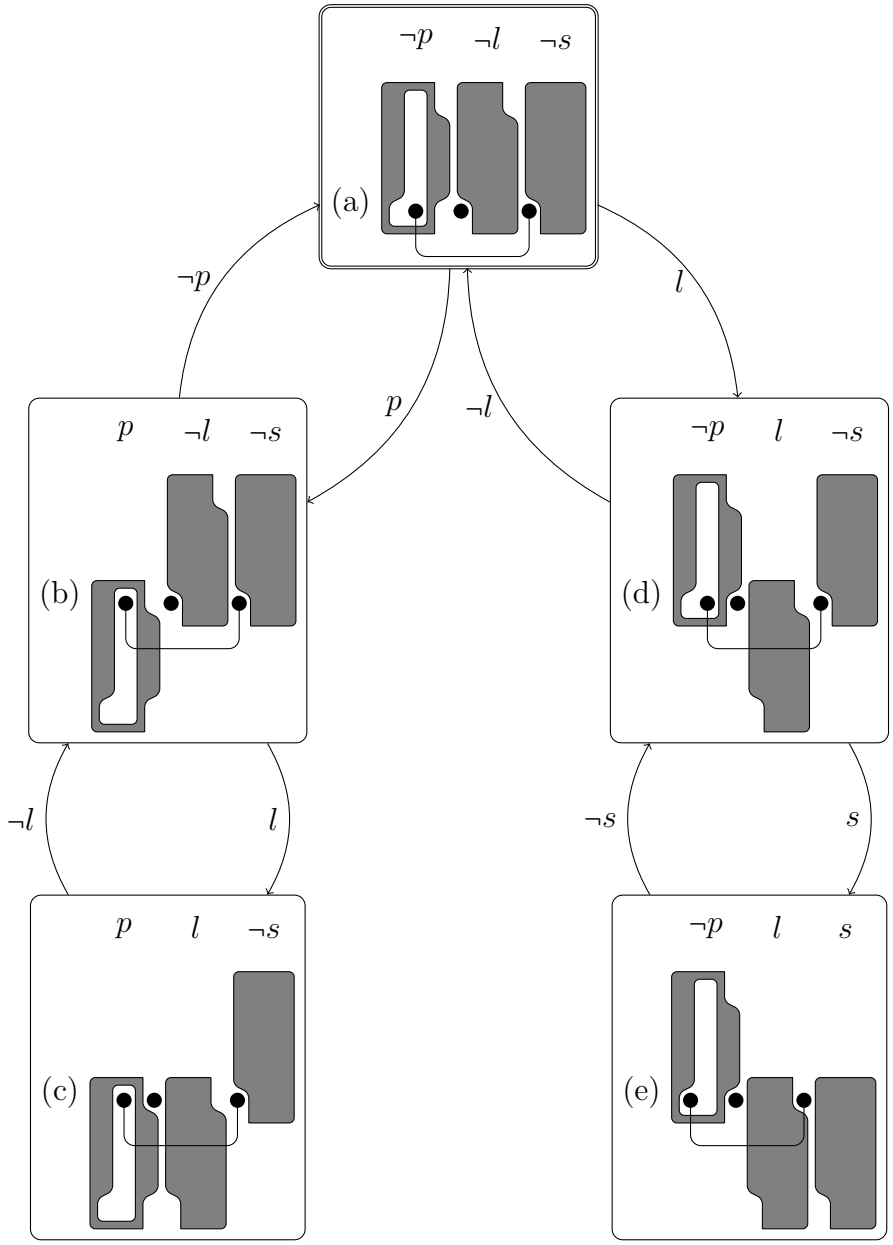


Figure 11.12: Example interlocking transition system.

For example, in Figure 11.11, s locks p normal. This corresponds to the following invariant on the states:

$$a \to \neg b$$

and the following invariant on transitions:

$$a \to \neg b' \wedge \neg b \to a'$$

Released by: a is free to be reversed after b has been reversed. Conversely b is locked in the reverse position when a is reversed. For example, in Figure 11.11, s is released by l. This corresponds to the following invariant on the states:

$$a \to b$$

and the following invariant on transitions:

$$a \to b' \wedge \neg b \to \neg a'$$

Both-ways: When a is reversed, it will lock b in the normal or reverse position. b cannot change position while a is reversed. For example, in Figure 11.11, l locks p both-ways. This does not restrict the state space, it does however correspond to the following invariant on transitions:

$$((a \wedge b) \to b') \wedge ((a \wedge \neg b) \to \neg b')$$

The majority of the interlocking logic is realised by 'locks normal' and 'released by' locks. It is possible to define long sequences of these interlocks that prevent complicated undesirable configurations. A detailed discussion about what is possible with these locks can be found in Appendix B. The both-way locks are typically used for the facing-point locks, as it is required to interlock the points in either the normal or reverse position.

The locking table for the example interlocking in Figure 11.11 consists of three entries, one for each lever. See Figure 11.13.

Lever	Locks Normal	Released By	Both-Ways
p	–	–	–
l	–	–	p
s	p	l	–

Figure 11.13: Example Locking Table

The translation of a locking table t into a ladder logic program is done by a direct encoding of the dynamic constraints. As t has n entries (one for each lever), the state space of the ladder program is given by n Boolean variables $(s_1, s_2, \ldots, s_n)$. When s_i is false, it means that lever i is normal, and when true, it means that lever i is reversed. The initial state of the

Remark

The naming of these locks is noteworthy; they are named from the perspective of the signalman. In fact, each entry is etched onto a metal plate and attached to its respective lever. So when the signalman wants to pull a lever, they know which levers need to be pulled or normalised first. The signalman is responsible for keeping the interlocking tidy by returning pulled levers when no longer required. This is the reason that in Figure 11.13 it appears that p does not have any locks specified, but they are implied by the locks on l and s.

Remark

Ladder logic is a language used to formalise decidable transition functions, for this reason the translation of locking tables into ladder logic does not make use of the state invariants. It only makes use of the transition invariants. Note that the state invariant is a provable theorem and it is considered at the end of the section.

ladder is defined by all the levers being in the normal position. Although it is possible to define an interlocking where this configuration is not possible, in general a design principle of the lever frames is that all levers should be returned to the normal position when not in use. Thus for the interlocking systems explored, this is a reasonable assumption about the initial state.

The input to the ladder program defines which lever is requested to be moved. By move, it is meant negate its position. To simplify the ladder logic, it is assumed that only one lever can move at a time. This essentially means that the rungs are unordered. This assumption is valid as, if two levers are moved simultaneously, either their logic is disjoint, in which case they could have been moved sequentially. Or if the logic is not disjoint, such as attempting to move l and s in state (d) of the transition system in Figure 11.12, at most one of the levers will move, or if all things are equal, neither lever will move as the dog will get wedged between them[2]. This is due to the shape of the tappets; the notches are at either end, and not midway. The tappets will only start to move if the logic allows them to, and if they start to move then there is nothing to stop them completing the move.

Now consider if the lever is half-way, i.e. neither normal or reverse. Any

[2]Similar to what happens when two people attempt to get through a normal sized door at the same time.

safety critical decisions that depend on that lever being in the normal or reverse position will require that it is interlocked into that position. Therefore, if the lever is half-way, any lever corresponding to the safety critical decision wont be able to be moved. Thus, if it is safe before the partial move of the lever, then it is safe after the partial move. As an example consider setting a signal. It will require that their opposing counterparts are interlocked to the danger aspects before it can be moved. If this is not the case, then the interlocking would violate signalling principle 1 and could be shown to by a counterexample of sequential moves of the levers.

This results in n possible inputs, although a compact representation of this information is possible, it becomes cumbersome to use as it must be encoded into Boolean variables. For this reason, a loose representation is used that has n^2 possible inputs, given by n variables, $(i_1, i_2, \ldots, i_n)$. Each input variable i_j indicates whether the lever j should be moved. To prevent multiple levers being moved simultaneously, an input invariant is given that allows at most one of the $(i_1, i_2, \ldots, i_n)$ variables to be true.

All that remains is to define the rungs of the ladder program. This requires introducing two functions that produce Boolean formulæ corresponding to the constraints in the table for a given lever. These functions determine for a given lever, whether it is free to move to the normal or reverse positions, respectively. The first function determines whether lever l is free to move into the normal position, and has the signature

$$\text{freetomove}_t^{\text{normal}} : (l : \text{Lever}) \to \text{BooleanFormula}$$

where t is the locking table. The formula produced by this function holds *iff* the conjunction of

1. All levers that l releases are in the normal position.

$$\forall l' \,.\, l' \text{ 'released by' } l \to \neg l'$$

2. All levers that lock l both-ways are in the normal position.

$$\forall l' \,.\, l' \text{ 'both-ways' } l \to \neg l'$$

holds. The implementation of this function requires recursion over the table, and in the case of (1), all entries that mention l in the 'released by' field should be normal. In the case of (2), all entries that mention l in the both-ways lock should be normal.

The second function determines whether l is free to be moved into the reverse position. It is a little more complicated than the first function as it

has more cases to consider (due to the semantics of the 'locks normal' and 'released by' locks being from the signalman). It has the signature

$$\text{freetomove}_t^{\text{reverse}} : (l : \text{Lever}) \to \text{BooleanFormula}$$

where t is the locking table. The formula produced by this function holds *iff* the conjunction of

1. All levers that lock l in the normal position are normal.

$$\forall l' \,.\, l' \text{ 'locks normal' } l \to \neg l'$$

2. All levers that lock l both-ways are in the normal position.

$$\forall l' \,.\, l' \text{ 'both-ways' } l \to \neg l'$$

3. All levers that l locks in the normal position are normal.

$$\forall l' \,.\, l \text{ 'locks normal' } l' \to \neg l'$$

4. All levers that release l are in the reverse position.

$$\forall l' \,.\, l \text{ 'released by' } l' \to l'$$

holds. Similarly, in the case of (1) and (2) the implementation of this function requires recursion over the locking table. In the case of conjunct (3), all levers mentioned by the locks field for the l entry in t are normal, and in the case of (4) all levers mentioned by the released by field for the l entry in t are reversed.

Due to the input invariant, the definitions of the rungs are simplified. Most notably, the execution order of the rungs becomes irrelevant as each rung does not depend upon previous rungs. Thus, it suffices to show for a given lever l and locking table t how the corresponding rung is defined.

$$\begin{aligned} s_l \coloneqq [i_l \wedge ((\neg s_l \wedge (\text{freetomove}_t^{\text{reverse}}\ l)) \\ \vee (s_l \wedge \neg(\text{freetomove}_t^{\text{normal}}\ l)))] \\ \vee (\neg i_l \wedge s_l) \end{aligned}$$

Intuitively, this rung performs case analysis on i_l, the variable indicating whether lever l should be moved. If it is false, then s_l is assigned s_l, i.e. does not change value. If it is true, then a further case analysis on variable s_l is performed. If s_l is false, then provided the constraints for it to move

Remark

A decidable transition system is constructable out of this ladder logic program; it is a fairly simple process as Agda will infer the relevant proofs.

reverse are fulfilled, it is assigned true. On the other hand if s_l is true, then (symmetrically to the reverse case) provided the constraints for it to be moved into the normal position are fulfilled, it is assigned false.

This concludes the translation of a locking table into a ladder logic program. Let gwiliLadder be the result of the translation.

Due to the inductive verification used by the framework, an issue of reachability is raised, namely in the inductive step when one of the antecedent states is not physically possible due to the interlocks. For instance, in Figure 11.12 it is physically impossible under normal conditions for $p \wedge s$ to hold. There are cases when these ill-formed states caused verification to fail, for this reason it is proved that the states are well-formed. This is done by defining a formula characterising well-formed states. Each entry $(l,\vec{n},\vec{r},\vec{b})$ in the locking table is mapped to the formula

$$s_l \to \bigwedge \overrightarrow{\neg s_n} \wedge \bigwedge \overrightarrow{s_r}$$

The well-formed states are given by a conjunction of this formula for each lever. The resulting formula is then proved to always hold, and this proof is used later when proving the signalling principles.

The above formalisation of locking tables, and their translation into ladder logic programs is in module `Ladder.LockingTable` of Appendix F.

For GSR the proof of well-formedness is done using a SAT solver. However, it should be possible to prove, in general, that any translated locking table has well-formed states. The definitions used for the Agda proof are problematic, and the GSR interlocking was the only occasion that the proof is required, so it was simpler to prove it directly using a SAT solver. It is left as future work to prove the correctness of the translation.

GSR interlocking. The GSR interlocking system consisted of 19 interlocked levers (2 spare levers), and each is attached to (at most) three tappets, one on each tier. When formalising the GSR interlocking, a hand translation from the engineering diagrams (similar to Figure 11.11) into a locking table was performed. There was one diagram for each tier. This resulted in the locking table in Figure 11.14. The table was then translated in Agda into a ladder logic program as previously described. See module `Gwili.Ladder` in Appendix F.

Lever Nº	Released By	Locks Normal	Both-Ways
1 (-)	–	–	–
2 (S)	12	3, 4, 5, 7, 8, 13, 16, 19, 20	–
3 (S)	12	2, 4, 5, 7, 8, 13, 16, 19, 20	–
4 (S)	12	2, 5, 7, 8, 13, 14, 16, 19, 20	–
5 (S)	–	2, 3, 4, 6, 11, 13, 14, 19, 20	–
6 (S)	12, 13	5, 7, 8, 16, 17, 18	–
7 (S)	14, 15	2, 3, 4, 6, 8, 11, 13, 17, 18	–
8 (-)	–	10, 14, 18, 20, 21	–
9 (-)	–	–	–
10 (-)	–	8, 11	–
11 (-)	–	5, 7, 8, 10, 21	–
12 (L)	–	–	13
13 (P)	–	2, 3, 4, 6, 19, 20	–
14 (P)	–	4, 5, 20	–
15 (L)	–	–	14
16 (S)	–	2, 3, 4, 6, 13	–
17 (S)	13	6, 7, 14	–
18 (S)	14, 15	6, 7, 8, 13, 19	–
19 (S)	–	2, 3, 4, 13, 14, 18	–
20 (S)	15	2, 3, 4, 8, 13, 14	–
21 (-)	–	8, 11	–

Figure 11.14: Bronwydd Arms locking table. Semantics of the levers are in the brackets, S is a signal, L is a facing-point lock and P are points, see Figure 11.7 for more information. The levers with a dash (-) were not connected to any hardware, but contribute to the interlocks and are reserved for future used.

11.1.3 State

Constructing the architectural states from the ladder states is a fairly simple process with the GSR interlocking. This is because all components of the architectural state (except the train's position) have a corresponding state variable/controlling lever. Therefore, these functions are defined by projecting the correct variable out of the ladder state (list of Boolean), which in the case of GSR relates to the lever number. For instance determining the aspect of signal 17, requires examining the 17^{th} state variable, when false the signal is displaying the danger aspect, and when true the signal is displaying

the proceed aspect. This gives rise to the projection

$$\begin{array}{l}\text{archSignalState} : \text{State}_{\text{gwiliLadder}} \to \text{Signal} \to \text{Aspect} \\ \text{archSignalState}\ s\ r\ =\ \{\ \text{false} \mapsto \text{Danger}\ ;\ \text{true} \mapsto \text{Proceed}\ \}\ s[r]\end{array}$$

where it is known that s has a of length 21, and $r \leq 20$.

In the case of segment locks, things are a little more interesting. The architectural state allows each segment to be locked/unlocked, but the interlocking only has two facing-point locks controlled by levers 12 and 15. Lever 12 locks set of points 13, which is track segment 2, and lever 15 locks set of points 14, which spans track segments 4 and 7. All segments not mentioned are linear, and are assumed to be permanently locked in the normal position (as they cannot move). Recall that locked refers to inhibition of physical movement. This gives rise to the following function:

$$\begin{array}{l}\text{archLockState} : \text{State}_{\text{gwiliLadder}} \to \text{Fin}\ 9 \to \text{Locking} \\ \text{archLockState}\ s\ \ 2\ =\ \{\ \text{false} \mapsto \text{Unlocked}\ ;\ \text{true} \mapsto \text{Locked}\ \}\ s[12] \\ \text{archLockState}\ s\ \ 4\ =\ \{\ \text{false} \mapsto \text{Unlocked}\ ;\ \text{true} \mapsto \text{Locked}\ \}\ s[15] \\ \text{archLockState}\ s\ \ 7\ =\ \{\ \text{false} \mapsto \text{Unlocked}\ ;\ \text{true} \mapsto \text{Locked}\ \}\ s[15] \\ \text{archLockState}\ s\ \ \_\ =\ \text{Locked}\end{array}$$

For all the technical details of constructing the state, see the module `Gwili.State` in Appendix F.

Train Simulator. The last component of the architectural state is the train position, i.e. a function relating trains to routes. This is achieved by defining a train simulator as described in Section 9.3.1.

The train simulator is built around a function that will select an available route for a train, which in effect will look-up for a given route, the connected routes, then select one of these routes that has a proceed aspect. If there is no available route, then the train will not proceed into a new route. For each train, an input stream is assumed. This input stream indicates whether the train wishes to move forward into a route, or to be stationary. First, the inputs for the trains are defined as follows:

$$\begin{array}{l}\text{data}\ \text{TrainInput} : \text{Set}\ \text{where} \\ \quad \begin{array}{lll}\text{Stationary} & : & \text{TrainInput} \\ \text{Move} & : & \text{Route} \to \text{TrainInput}\end{array}\end{array}$$

Assume a stream of inputs

$$inp : \mathbb{N} \to \text{Input}_{\text{gwiliLadder}}$$

for the interlocking, and stream of inputs

$$tinp : \text{Train}_{\text{gwiliLayout}} \to \mathbb{N} \to \text{TrainInput}$$

for the trains, then the following function elaborates on the details of simulating the train position.

$$\begin{array}{l}
\text{trainRoute} : \mathbb{N} \to \text{Train}_{\text{gwiliLayout}} \to \text{Route} \\
\text{trainRoute } 0 \quad tr \;=\; \text{initialroute } tr \\
\text{trainRoute } (\text{suc } t) \quad tr \;= \\
\quad \textbf{if } (tinp\ tr\ t \equiv \text{Move } rt) \\
\quad\quad \wedge\ (rt \text{ isin } (\text{gwiliRouteConnections } (\text{trainRoute } t\ tr))) \\
\quad\quad \wedge\ (\text{archSignalState}_{(\text{nthState } inp\ t)}\ rt \equiv \text{Proceed}) \\
\quad \textbf{then } rt \\
\quad \textbf{else } \text{trainRoute } t\ tr
\end{array}$$

11.2 Verification

Two types of verification were performed on Gwili's interlocking. One verification checked the operational correctness of the interlocking, i.e. the interlocking correctly refines the control table. The other verification checked that the interlocking fulfilled the signalling principles, and in turn the safety requirements.

11.2.1 Control Table

Verifying that the interlocking correctly refined the control table was achieved by mapping each entry of the control table to a formula representing correctness of the entry. Then using a SAT solver it was inductively proved that these formulæ always hold.

As previously noted the mechanical interlocking does not have track circuit inputs, so it cannot be formulated that a route with an occupied segment does not have a proceed aspect. Instead the correctness is formulated as *if a signal/route displays a proceed aspect, then all set of points in that route are in the correct position and the facing sets of points are also locked*. To formulate this condition, first, the following functions are defined that map states of the hardware to propositional formulæ representing these states.

$$\begin{array}{l}
\text{rtSet} : \text{Signal} \to \text{BooleanFormula} \\
\text{rtSet } n \;=\; \text{var } n
\end{array}$$

The following three functions relate to properties that only sets of points have, i.e. normal, reverse and locked. As there are only three segments (2,4,7) that relate to sets of points, the following case-distinctions are valid.

$$
\begin{array}{l}
\text{segNormal} : \text{Fin } 9 \to \text{BooleanFormula} \\
\text{segNormal } 2 \;=\; \neg(\text{var } 13) \\
\text{segNormal } 4 \;=\; \neg(\text{var } 14) \\
\text{segNormal } 7 \;=\; \neg(\text{var } 14) \\
\text{segNormal } \_ \;=\; \text{true}
\end{array}
$$

$$
\begin{array}{l}
\text{segReverse} : \text{Fin } 9 \to \text{BooleanFormula} \\
\text{segReverse } ts \;=\; \neg\,(\text{segNormal } ts)
\end{array}
$$

$$
\begin{array}{l}
\text{segLocked} : \text{Fin } 9 \to \text{BooleanFormula} \\
\text{segLocked } 2 \;=\; \text{var } 12 \\
\text{segLocked } 4 \;=\; \text{var } 15 \\
\text{segLocked } 7 \;=\; \text{var } 15 \\
\text{segLocked } \_ \;=\; \text{true}
\end{array}
$$

Remark

Sets of points are only considered unsafe when traversed in the facing (diverging) direction, and they are unlocked. It could be the case that the set of points changes its physical position, then part of the train will attempt to diverge from the rest of the train resulting in a very significant risk of derailment. When traversed in the other direction (converging), it is not essential with respect to safety that the set of points is locked, or even in the correct position as the weight of the train will force the points into the correct position. For more information see Section 8.3.

Thus a function mapping Gwili's control table entries into a Boolean formula is given as follows:

$$
\begin{array}{l}
\text{CorrectEntry} : \text{ControlTableEntry}_{\text{gwiliLayout}} \to \text{BooleanFormula} \\
\text{CorrectEntry } e \;=
\end{array}
$$

$$
\text{rtSet } s \Rightarrow \left(\begin{array}{l}
\text{foldr } (\lambda\, s\; x \to \text{segNormal } s \wedge x)\ \text{true}\ \text{normalpoints}_e \\
\wedge\ \text{foldr } (\lambda\, s\; x \to \text{segReverse } s \wedge x)\ \text{true}\ \text{reversepoints}_e \\
\wedge\ \text{foldr } (\lambda\, s\; x \to \text{segLocked } s \wedge x)\ \text{true}\ \text{facing}_e
\end{array}\right)
$$

Let $\psi_r = \text{CorrectEntry } r$, and $\psi = \psi_2 \wedge \psi_3 \wedge \cdots \wedge \psi_{20}$. ψ expresses that all entries in the control table are correctly refined by the interlocking. The

proof of correctness is given by applying the function inductiveProof (see page 223) to gwiliLadder and to ψ.

```
ControlTableCorrect : LadderCorrect gwiliLadder ψ
ControlTableCorrect = inductiveProof gwiliLadder ψ
```

Type-checking the above function takes ≈ 28 seconds. This includes computing the ladder, ψ, the base and inductive formula, and executing the SAT solver. As usual, the SAT solver (Z3 [dMB08]) could solve the problem in a fraction (< 0.1) of a second. See module `Gwili.ControlTableCorrect` in Appendix F for full details.

11.2.2 Safety

The second verification relates to proving that the interlocking is safe, with respect to the requirements in Chapter 8. Recall from Theorem 8.4.1 and Theorem 8.4.2 that to prove the interlocking system is safe, requires that the interlocking fulfils the signalling principles 1, 2, 3 and 4 (cf. page 177).

First the signalling principles 2 and 4 are considered. Both of these principles formalise the requirement of the interlocking not to perform actions (clearing a signal or unlocking a set of points) to a route which is occupied. As previously remarked, the interlocking system is not equipped with train detection inputs, and the route a train is currently in, is simulated; therefore, the interlocking is unable to enforce these principles. It is the responsibility of the signalman to enforce these signalling principles by eye. The signalman should look out of the window in the signal box and decide whether it is safe to set a signal to proceed (or unlock a set of points)—if there is a train occupying the route that the signal protects, then the signalman should not clear the signal. Although perhaps obvious, this issue highlights a substantial safety related weakness with mechanical interlocking systems. Mechanical interlocking systems provide a reliable means of preventing unsafe combinations of signals and points, particularly when setting a route to a proceed aspect. However, they are unable to enforce constraints that relate to the train position, or more generally, to occupied routes. This is because the guarding signal of an occupied route should be set to danger to prevent subsequent trains entering the route, thus releasing the interlocks. For this reason, the signalling principles 2 and 4 are assumed to hold, i.e. it is assumed that the signalman operates correctly. See Appendix B for more information about the limitations of these systems.

In the case of signalling principle 1, which formalises that opposing signals should be exclusive. The proof proceeds in Agda by performing case-distinctions on the routes to identify all distinct pairs of routes that share a common

Remark

It should be noted that, for modern interlocking systems, the signal aspects and route selection mechanisms are distinct. This, when coupled with track circuit inputs, allows for a greater degree of expressiveness with respect to the constraints the interlocking can enforce. There, the issue of assuming that the signalling principles 2 and 4 hold vanishes.

track segment. Then for these cases, the proof follows by applying a SAT solver. For example without loss of generality, consider the pair of routes 4 and 19, that share a common track segment, namely segment 2. The proof obligation becomes

$$\begin{array}{l}\forall t\ inp\ .\\ \quad \text{archSignalState}_{(\text{nthState}\ inp\ t)}\ 4 \equiv \text{Danger} \vee \\ \qquad\qquad\qquad\qquad \text{archSignalState}_{(\text{nthState}\ inp\ t)}\ 19 \equiv \text{Danger}\end{array}$$

after unfolding archSignalState, it becomes

$$\begin{array}{l}\forall t\ inp\ .\\ \quad \{\text{false} \mapsto \text{Danger}; \text{true} \mapsto \text{Proceed}\}\ (\text{nthState}\ inp\ t)[4] \equiv \text{Danger} \vee \\ \qquad \{\text{false} \mapsto \text{Danger}; \text{true} \mapsto \text{Proceed}\}\ (\text{nthState}\ inp\ t)[19] \equiv \text{Danger}\end{array}$$

Therefore, it suffices to show that

$$\forall t\ inp\ .\ [\![\ \neg(\text{var}\ 4) \vee \neg(\text{var}\ 19)\ ]\!]_{(\text{nthState}\ inp\ t)} \qquad (*)$$

holds. Note that $(*)$ is almost at a low enough level to follow directly by the SAT solver. However, $(*)$ is of the form *for all time ...* and the SAT solver yields proof-objects of the form *for all environments ...*, so in the following, it is demonstrated in a bottom-up manner how to translate the SAT solver proof-object into a proof of $(*)$. The application of the SAT solver yields the following:

$$\begin{array}{l}\text{opp-4-19} : \text{LadderCorrect}\ \text{gwiliLadder}\ (\neg(\text{var}\ 4) \vee \neg(\text{var}\ 19))\\ \text{opp-4-19} = \text{inductiveProof}\ \text{gwiliLadder}\ (\neg(\text{var}\ 4) \vee \neg(\text{var}\ 19))\end{array}$$

By the decidability of the ladder logic program, it is possible to translate the general proof that $(\neg(\text{var}\ 4) \vee \neg(\text{var}\ 19))$ always holds, into a proof that for any sequence of input values it also holds. Thus an element of the following type is obtained:

$$\forall t\ inp\ .\quad [\![\ \neg(\text{var}\ 4) \vee \neg(\text{var}\ 19)\ ]\!]_{(\text{nthState}\ inp\ t +\!\!+\ inp\ t +\!\!+\ \text{nthState}\ inp\ (t+1))}$$

The environment consists of two states and an input; this is because the induction used by LadderCorrect is defined over transitions (pairs of states, and an input) of the ladder. To prove $(*)$, the environment need to be truncated. Note that the state consists of 21 variables, and all variables indicies in the formula are less than 21. Therefore, the environment is truncated by Lemma 9.3.14, and a proof of $(*)$ is obtained.

The proofs in the remaining 41 cases of the opposing routes signalling principle are trivially adapted from this proof. It is noted that for efficiency reasons all 42 cases are proved by the SAT solver at the same time (conjunction of all 42 cases), then split afterwards; this removes the considerable overhead that each application of inductiveProof entails, i.e. generating the ladder, proof obligations and inferring well-formedness proofs of the ladder.

Remark

A number of these 42 cases required that the state space was well-formed (cf. end of Section 11.1.2). This was achieved by showing that a proof of these 42 cases followed from a proof that the states are well-formed. Then, a proof of the well-formedness was applied to obtain a proof of all 42 cases. As described above, the proof is split.

For full details about the proof of signalling principle 1, see the modules `Gwili.Ladder.OpposingRoutes` and `Gwili.OpposingSignals` in Appendix F.

The proof of signalling principle 3, which formalises that when a route is set to proceed all track segments in that route are locked, is similar. It proceeds by performing case-distinction on the route, and then checks for each route that contains a facing set of points that if the signal/route displays a proceed aspect, then that facing set of points is locked. In fact, this principle could be reduced from the operational correctness proof of the last section; however, it is directly performed again here so that the proof is tractable and the safety/operational correctness proofs are independent. There are 6 cases to consider, one for each of the following routes C2, 4, 6, 7, 18 and 20; the first three routes traverse segment 2/points 13 in the facing direction and require that FPL 12 is engaged, and the penultimate 2 routes traverse segment 4/points 14 in the facing direction and require that FPL 15 is engaged. Finally, route 7 traverses segment 7/points 14 in the facing direction, and also requires that FPL 15 is engaged. Without loss of generality consider the

case of route 6, the proof obligation becomes

$$\begin{array}{l}\forall t\ inp\ .\\ \quad \text{archSignalState}_{(\text{nthState}\ inp\ t)}\ 6 \equiv \text{Proceed}\\ \quad\quad \rightarrow \text{archLockState}_{(\text{nthState}\ inp\ (t+1))}\ 2 \equiv \text{Locked}\end{array}$$

After unfolding the definitions of archSignalState and archLockState, the following obligation is obtained.

$$\begin{array}{l}\forall t\ inp\ .\\ \quad \{\text{false} \mapsto \text{Danger}; \text{true} \mapsto \text{Proceed}\}\ (\text{nthState}\ inp\ t)[6] \equiv \text{Proceed}\\ \quad \rightarrow \{\text{false} \mapsto \text{Unlocked}; \text{true} \mapsto \text{Locked}\}\ (\text{nthState}\ inp\ (t+1))[12] \equiv \text{Locked}\end{array}$$

Which follows from a proof of

$$\forall t\ inp\ .\ [\![\ \text{var}\ 6 \Rightarrow \text{var}\ 55\ ]\!]_{(\text{nthState}\ inp\ t +\!\!+ inp\ t +\!\!+ \text{nthState}\ inp\ (t+1))}$$

where var 55 refers to the state of FPL 12 in the successor state. That is 55 = 21 + 22 + 12, where the number of state variables is 21 and number of input variables is 22. A proof of the above is obtained from

$$\text{inductiveProof}\ \text{gwiliLadder}\ (\text{var}\ 6 \Rightarrow \text{var}\ 55)$$

and the decidability of the ladder.

For full details about the proof of signalling principle 3, see modules `Gwili.Ladder.Facing` and `Gwili.FacingPointLock` in Appendix F.

Thus, two of the signalling principles are proved to hold, and two of them are assumed[3] because in the context of mechanical interlocking systems they refer to the signalman's behaviour. To show that GSR is always safe requires one final consideration: proving that the initial configuration fulfils the two safety requirements. In this verification, it is only considered that there are two trains, initially the trains are located in routes 3 and 16. Recall that the initial state of the interlocking is defined by all levers being in the normal position, most importantly the signals are all set to danger.

The initial proof obligation for Theorem 8.4.1 (*S1*) is defined with respect to GSR for time 0 as follows

$$\begin{array}{l}\forall train\ segment\ .\ \text{FacingInRoute}_{\text{gwiliLayout}}\ segment\ (\text{trainRoute}\ 0\ train)\\ \quad\quad\quad \rightarrow \text{archSegmentLock}_{(\lambda\_ \rightarrow \text{false})}\ segment \equiv \text{Locked}\end{array}$$

where, depending on the train

$$\text{trainRoute}\ 0\ train = 3 \vee \text{trainRoute}\ 0\ train = 16$$

[3]They are passed as parameters to the proof of safety so that it is clear they are assumed.

In both cases, the routes do not contain any sets of points so the proof follows trivially.

The proof obligation that GSR is initially safe with respect to Theorem 8.4.2 (*S2*) is formulated as follows:

$$\begin{aligned}\forall\, train_1\ train_2\ segment\ .\ train_1 \not\equiv train_2 \rightarrow \\ \neg(\mathrm{SegInRoute}_{\mathrm{gwiliLayout}}\ segment\ (\mathrm{trainRoute}_0\ train_1) \wedge \\ \mathrm{SegInRoute}_{\mathrm{gwiliLayout}}\ segment\ (\mathrm{trainRoute}_0\ train_2))\end{aligned}$$

This trivially holds as routes 3 and 16 do not share any common track segments.

Thus, GSR is initially safe. This concludes the verification. It has been shown (provided the signalman operates correctly) that it is not possible for trains to collide on GSR, nor can they make facing moves over unlocked sets of points. For the final proof, see module `Gwili.Safe` in Appendix F.

It is possible to increase the number of trains, provided that the initial constraints are not violated. For example, it is possible to add a train to route 17 or to C5 without any problems. In these cases, there are issues of liveliness that must be considered.

Remarks

Although the verification of GSR appears fairly painless, it was still time consuming as it was the first concrete railway system that the safety of which was fully verified. In total, about 10 weeks of my time were dedicated to the verification of GSR. A lot of effort was required to understand the semantics of the mechanical interlocking from the engineering diagrams and construct a locking table from them. While attempting to verifying GSR, the definitions in Chapter 8 relating to the models and safety had to be modified, This was because of a small number of shortcomings in the early formalisations of the layout and railway safety that were incompatible with the actual topology or interlocking. For instance, one such shortcoming was that signalling principle 4 was initially formulated as follows:

$$\begin{aligned}\forall t\ train\ segment\ .\ \mathrm{SegInRoute}_l\ segment\ (\mathrm{trainRoute}_t\ train) \\ \rightarrow \mathrm{locked}_t\ segment \equiv \mathrm{Locked} \\ \rightarrow \mathrm{locked}_{t+1}\ segment \equiv \mathrm{Locked}\end{aligned}$$

This was not correct as it requires that the set of points is locked after the train has left the route. It should be that it requires the set of points is

locked before the train enters the route. It was correctly reformulated to:

$$\begin{aligned}\forall t\ \mathit{train}\ \mathit{segment}\ .\ &\mathrm{SegInRoute}_l\ \mathit{segment}\ (\mathrm{trainRoute}_{t+1}\ \mathit{train})\\ &\rightarrow \mathrm{locked}_t\ \mathit{segment} \equiv \mathrm{Locked}\\ &\rightarrow \mathrm{locked}_{t+1}\ \mathit{segment} \equiv \mathrm{Locked}\end{aligned}$$

Remark

This example of an incorrect signalling principle highlights the importance of requiring that the models are validated by experts. The almost unnoticeable difference (by eye) between these two definitions was the difference between the signalling principle being provable or unprovable. It could be worse: unknowingly proving the wrong statement.

During the verification, it became apparent that shunt signal 7 did not require the set of points 14 to be locked (by FPL 15) before being cleared; however, it does require set of points 14 to be in the reverse position. This meant that a proof of signalling principle 3 was not possible. To repair this, the constraint that *lever 7 is released by lever 15* was appended to the locking table to allow the proof to succeed. Moreover, it is further emphasised that this does not mean that GSR is considered unsafe; this is due to the fact that passenger trains are not (legally) allowed to respond to a shunt signal clearing, and thus passenger services are unable to travel onto the "LOOP" line, see Figure 11.7, hence they can never make a facing move over an unlocked set of points. It could be argued that non-passenger trains could make such moves, but they would be classed as shunting moves, something that is considered to be a manual operation.

This issue arose because the model of topology presented in Chapter 8 does not consider different types of signals. That is all signals are treated as main signals. Had shunt signals been differentiated from main signals, allowing for a different formalisation of the signalling principles, then the locking table would fulfil signalling principle 3 without being modified. Similarly, set of points 14 (segment 7) could have been omitted from the 'facing' field on the control table for route 7.

The type-checking of the GSR scenario, including the verification required 8:10 minutes; this includes executing the SAT solver 16 times, which as always completed in a fraction of a second.

Future Work

There is one final, yet significant remark to be made. For the purposes of this case study, only the locking table and the topology were formalised. This meant that there were safety features outside the scope of the verification that should be remarked on. One such feature was the use of track circuits over the sets of points (represented by 12T/15T in Figure 11.6, also the indicator dials at the base of the image). These track circuits used supplementary electromechanical locks to prevent the associated FPL levers being moved while the set of points is occupied. If these supplementary locks and track circuits had been formalised, it is presumed that signalling principle 4 would have also been provable. Nonetheless, signalling principle 2 would still have had to be assumed.

Another safety related device not considered was the use of a token system. Trains travelling into route 16 (out of sight from the signal box) were given a token to prevent multiple trains being allowed to enter route 16. Formalising this system would be necessary to prove signalling principle 2, but not sufficient as it is still the case that some routes are checked to be un/occupied by the signalman, e.g. route 3.

Perhaps a better solution is to formalise the signalman as a liveware component of the system. This would require adding an input that represents detection of the trains position, and relating it to constraints on setting the signals. From this, a proof of the two remaining signalling principles would be derivable. Formalising the signalman would further reduce the validation requirements, it would also make precise the job of the signalman. I.e. the signalman would be informed of precisely what is required of them.

11.3 Simulation

The final part of this case study was to compile and execute the GSR interlocking system. This was achieved in a similar method to the Pelicon crossing simulation described in Section 10.6. Note that the simulation is executing verified code, i.e. it is an example of the proofs-as-programs paradigm.

A recursive non-terminating terminal application was created that would take input for each iteration the number of a lever (1..21), which has been requested to be moved. The input is defined in such a way as it must fulfil the input invariant of the ladder, that is it requires only one lever is moved at a time. Then one iteration of the ladder program is executed, and the programs state is printed to the terminal in a meaningful way. That is, instead of printing true/false for each variable, sets of points are displayed

as normal/reverse, facing-point locks are displayed as locked/unlocked and signals are displayed as proceed/danger. See module `Gwili.GwiliSimulator` in Appendix F for the implementation of the simulator.

Simulating the interlocking using this method provided an insight that intuitively demonstrated the interlocking was correct and functional (i.e. validation). The amount of work to simulate the interlocking is minimal as all required definitions were already available (notably decidable transition systems), and compiling programs is fully supported by Agda.

Below is a transcript of the simulation. In the simulation, first lever 6 is requested to move reverse, which is not possible because it requires that the set of points 13 is locked in the reverse position. Once that constraint has been fulfilled lever 6 is then requested to be moved to the reverse position again, which is successful this time. Then FPL 15 is engaged, and signal 20 is requested to be cleared. This is not possible as it requires the set of points 13 to be in the normal position. Therefore, signal 6 is normalised, FPL 12 is unlocked and set of points 13 is normalised. Finally, signal 20 is cleared. The simulation corresponds to the following input sequence:

$$\mathbf{6}, 13, 12, 6, 15, \mathbf{20}, 6, 12, 13, 20$$

The emboldened requests are disallowed by the interlocking. The following is a transcript of the simulation:

```
Gwili Rail simulator
entering main loop...

initial state:

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Danger | 20: Danger
7: Danger

fpl 12 / point 13:
  Unlocked / Normal
fpl 15 / point 14:
  Unlocked / Normal
```

```
enter leaver to move [1..21]
6
## Request failed ##

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Danger | 20: Danger
7: Danger

fpl 12 / point 13:
  Unlocked / Normal
fpl 15 / point 14:
  Unlocked / Normal


enter leaver to move [1..21]
```

```
13

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Danger | 20: Danger
7: Danger

fpl 12 / point 13:
  Unlocked / Reverse
fpl 15 / point 14:
  Unlocked / Normal


enter leaver to move [1..21]
12

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Danger | 20: Danger
7: Danger

fpl 12 / point 13:
  Locked / Reverse
fpl 15 / point 14:
  Unlocked / Normal


enter leaver to move [1..21]
6

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Clear  | 20: Danger
7: Danger

fpl 12 / point 13:
  Locked / Reverse
fpl 15 / point 14:
  Unlocked / Normal


enter leaver to move [1..21]
15

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Clear  | 20: Danger
7: Danger

fpl 12 / point 13:
  Locked / Reverse
fpl 15 / point 14:
  Locked / Normal


enter leaver to move [1..21]
20
## Request failed ##

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Clear  | 20: Danger
7: Danger

fpl 12 / point 13:
  Locked / Reverse
fpl 15 / point 14:
  Locked / Normal


enter leaver to move [1..21]
```

```
6

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Danger | 20: Danger
7: Danger

fpl 12 / point 13:
  Locked / Reverse
fpl 15 / point 14:
  Locked / Normal


enter leaver to move [1..21]
12

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Danger | 20: Danger
7: Danger

fpl 12 / point 13:
  Unlocked / Reverse
fpl 15 / point 14:
  Locked / Normal
```

```
enter leaver to move [1..21]
13

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Danger | 20: Danger
7: Danger

fpl 12 / point 13:
  Unlocked / Normal
fpl 15 / point 14:
  Locked / Normal


enter leaver to move [1..21]
20

Signals:
2: Danger | 16: Danger
3: Danger | 17: Danger
4: Danger | 18: Danger
5: Danger | 19: Danger
6: Danger | 20: Clear
7: Danger

fpl 12 / point 13:
  Unlocked / Normal
fpl 15 / point 14:
  Locked / Normal
```

END SIMULATION

Chapter 12

Railways – Summary

Throughout this part, verification in the railway domain has been explored. This has entailed formalising the physical layouts and control tables at a low enough level that they can be easily validated by domain experts. Then deriving abstract models of the topology and the state of the components in the topology (e.g. signals). Abstract notions of signalling principles and safety requirements were then formalised for these abstract models. The formalisation of the safety requirements was then shown to follow by a selection of 4 signalling principles.

Proving for a given interlocking system that it fulfils this safety requirement required that the interlocking system fulfilled the signalling principles. This was achieved by modelling the interlocking systems, and formalising correctness for these systems. The interlocking systems considered in this thesis were implemented using ladder logic programs; hence ladder logic was formalised. Proof that a ladder logic program fulfilled the signalling principles was performed by induction and SAT solving. The interface to the SAT solver was by the (Oracle + Reflection) approach developed in Part I of this thesis.

The framework that was built-up in this part has been used for two case studies. The first was the Gwili Steam Railway interlocking system, and the second was a London Underground station. Both of these verifications were successful with respect to being feasible to perform from within Agda.

The issue of operational correctness was also explored; that is determining whether an interlocking system correctly refines a control table. This was a fairly simple process, provided the semantics of the control table is formalised. However, a generic framework for the operational correctness was not established due to its dependence on the underlying technology and signalling schemes. Instead, the two interlocking systems mentioned previously were shown to correctly refine their control tables.

12.1 Comparison

In Chapter 2, with respect to verifying high-level safety requirements of interlocking systems, four papers were discussed [Han98, HP00, Win02, SBRG12]. It is now considered how this thesis differs from these attempts.

These papers were selected as they discuss the following high-level safety requirements:

- Trains do not collide, and
- Trains do not detail.

Each of these papers is now compared in detail.

The first paper [Han98], Hansen does not perform verification but instead validates station topologies by simulation using the VDM. That is by an executable specification of railway topology. Hansen formalises the railway topology in a similar way to this thesis, i.e. track segments as a directed graph, and the edges are annotated by 2-aspect signals. However, provisions are made for the special track segments: sets of points and cross-overs. Specifically sets of points have their normal and reverse positions explicitly specified, and cross-overs are specified to simplify the models. The explicit formalisation of sets of points means that a set of points cannot be placed as a terminal segment on a line, because it must be connected to three other segments. The state space of the topology is given by the Cartesian product of the state spaces of the underlying components. That is, signals display stop or proceed, points are normal or reverse, and track segments know which trains are occupying them. A number of well-formedness constraints are given on the topology that ensures a train only occupies contiguous sections of track, and that the points are in a compatible position when occupied by a train. Furthermore, *areas* are defined. Areas are sets of contiguous track segments that are delimited by signals, an area can be thought of as a bidirectional route. These are computed to be maximal. Hansen provides the safety requirement for trains not colliding, which is given as the following VDM formula

$$\forall train_1\ train_2\ .\ train_1 \neq train_2 \rightarrow \text{area}(train_1) \cap \text{area}(train_2) = \varnothing$$

and proved it follow by the specification. The paper does not provide a formalisation of the other safety requirement; however, it claims that the work was undertaken. A number of Danish station topologies were then formalised and simulated. During the simulation, it was discovered by domain experts that the notion of an area was incorrect. This is because manual shunting moves required a dynamic area. This means the train driver could control

a set of points without first asking for the interlocking systems permission, resulting in a new area. However, this is only allowed to be done while shunting trains, and hence more information was added to the specification that identified whether the train was timetabled or shunting, and whether a set of points was being controlled by the interlocking or the driver. The result of the simulations was that a number of errors in the topologies of existing stations were identified.

In this thesis, the models are at a higher-level of abstraction than Hansen's models. The motivation for not fixing the underlying choice of track segments is to focus on the core problem without too many technical details. This is apparent when looking at Hansen's well-formdness conditions that specifically make use of the normal and reverse positions of a set of points. This fixing of track segments requires that every new (even yet to be defined) type of segment would manually require adding to the models. Whereas in our approach it is possible to model a turntable without changing the specification. It is also noted that Hansen's work differs by not attempting to verify any concrete interlocking system.

Haxthausen and Peleska's paper [HP00] makes use of the RAISE specification language. In the paper, they define an abstract model of a (German) distributed railway control system, then make 5 refinements to this model to obtain a specification that has an explicit state space. For which it is possible to formalise a concrete control system, and show that it fulfils the required properties. Notably the signalling scheme used in this paper does not have any explicit signals; instead, there are a number of control systems situated along the railway which are communicated with via radio link when the train is within a few meters proximity. The topology model consists of trains, track segments and sets of points. The state of each component is given, notably the state of a set of points is a pair of two adjacent track segments that are currently connected, and the state of a train is its position and direction. The position of a train is either the track segments that it is occupying, or an *error* position. The error position identifies when a train has run off the end of a line or when it traverses a set of points that are in the wrong position. The models are event based, and formalise two events: train moving, and switching a set of points. These events are then axiomatised. Most importantly are the axioms that detail how a train correctly moves, i.e. between two connected segments, such that the states of the segments are compatible. The high-level safety is formulated as the following RAISE formulæ:

$$\forall\, train_1\ train_2 \,.\, train_1 \neq train_2 \rightarrow \mathrm{position}(train_1) \cap \mathrm{position}(train_2) = \varnothing$$

and

$$\forall\, train \;.\; \text{position}(train) \neq \text{error}$$

These requirements are proved (by hand) to follow by the specification. The specification is then refined, and each refinement is verified. Furthermore, the architecture of the control systems are formalised, to allow for the verification of an actual control system.

Its clear that the models used by Haxthausen and Peleska are substantially different that those used in this thesis. For example, they do not formalise signals, to compensate the track segments are required to contain more information about the direction of allowed travel, and trains have an error position. Moreover, the refinement based approach is not used in this thesis.

The paper [Win02] by Winter is based on the process calculus CSP. In this paper Winter defines the most general, safe railway network. In the spirit of process calculus, each component is modelled as a process, e.g. trains, sets of points, signals and routes. Further processes are defined that formalise the correct behaviour of trains, signals and sets of points. For example, correct trains are modelled by a process that ensures that trains only move between connected segments in the forward direction, among other constraints. Notably, Winter models the front and rear of the trains separately so it is easier to determine which direction the train is moving.

The safety of the system is formulated by defining two processes, one process for no collision and one for no derailment. The definitions are not repeated here. The safety requirements are not proved to follow from the specification, instead the safety requirements form part of the specification of a rail network, i.e. it is assumed that the safety requirements are correctly formulated. What is done is to outline the methodology of checking whether a concrete control system refines the specification, using the model-checker FDR. Winter suggests that the approach is well-suited to determining the correctness of the design of an interlocking system.

The final paper [SBRG12] is by Sabatier. He performs the same type of high-level verification as done in this thesis, however, no specific control systems are considered. In the paper, steps towards a full, formal proof of a New York City subway line are presented. The high-level safety requirements that were verified are that trains do not collide, derail or break the speed limit. No formalisation of these requirements, or of the models was contained in the paper. The underlying logic is Event-B, and the automation is provided by the Atelier-B prover. However, no explicit reference is made to signalling principles, but instead they are generalised as assumptions from the target domain, and how to determine if they are good/bad assumptions. A large

portion of the paper is dedicated to the importance of relating the domain specifications into formal specifications, and most importantly whether assumptions made are valid. It is suggested that an assumption should be easy to validate, and determine whether it holds, or not. The paper also describes the author's experiences while proving the safety requirements. Specifically during a proof it becomes clear when (and why) an assumption is needed, if it is needed at all. This is what has been described in this thesis as verifying domain knowledge.

Generally, compared to this thesis Sabatier's work is similar in that the issues of assumptions from the railway domain and abstract proofs are carried out. However, the paper is only 4 pages, and reports about an on-going project, so it is not clear what exactly was accomplished.

12.1.1 Remarks

These papers are all similar. They create an abstract model of the railway, then directly verify (interactively or automatically) that this model is safe with respect to trains colliding and derailing. Haxthausen and Peleska go further by refining the specification to an implementation.

In contrast, this thesis introduces intermediate lemmata (signalling principles) that are obtained from the railway domain. These intermediate lemmata are already heuristics used in industry by the designers and developers of these systems. Then these intermediate lemmata are proved to be sufficient to guarantee the safety. In doing so, there is a possibility of detecting missing, redundant, or conflicting signalling principles, which would have a benefit for the whole railway network, possibly the whole industry.

Then for a concrete, hand-crafted control system, safety is shown by proving that it fulfils these intermediate lemmata. Hence the framework explained in this thesis will also verify domain knowledge, as well as concrete control systems. This has the added advantage that the approach builds upon domain knowledge that has taken over 150 years to accumulate, which should also be intuitive for engineers to validate.

12.2 Future Work

The abstract verification methodology built-up during this thesis is ideal to explore safe weakenings of signalling principles. This would be of interest to determining whether a selection of signalling principles optimised for throughput is safe. However, explorations of this type were not undertaken as part of this thesis and have been left as future work.

It would be desirable to extend the framework to allow liveware components to be formalised. In both the verifications performed, it was identified that a complete proof could not be obtained due to the human component. In the case of GSR, two of the signalling principles were assumed as the interlocking system was not technologically advanced enough to fulfil them; it could not take train detections inputs. This implies that it is assumed that the signalman never violated these two signalling principles. In the case of the underground station, it was assumed that the interlocking system was not in maintenance mode for one of the signalling principles to be fulfilled. The maintenance mode is activated by a human, and they only do so in response to specific events, e.g. hardware failure. Formalising the liveware would further reduce the amount of validation required. It would also increase safety as it is clear what the requirements on liveware are, i.e. formalise the job descriptions.

An idea that was not explored in-depth during this thesis was to perform safety verification directly upon the control tables. This has been studied by other researches [TRN02, RN03]. In principle, the direct verification would simplify the process by only having to verify that the interlocking system correctly refines the control table, and not that it fulfils the signalling principles. Some of the signalling principles discussed during this thesis follow by the semantics of a control table, for instance, the opposing signals principle.

The formalisations built-up in this part, i.e. topologies, control tables, signalling principles and safety requirements have not been validated by domain experts. This was done in-order to focus on the issue of identifying the framework without getting overwhelmed by the technical details that inevitably occur, from our experience, when the models are validated. For example, only one type of signal is considered in the models, but, in practice, there are many different types. Furthermore, there are exceptions to the rules such as in maintenance or emergency situations. It would be intriguing to attempt to validate these models to obtain a more accurate model. One important improvement that would be desirable would be to formalise the notion of a partial route release. These partial route releases are required to formalise when it is safe for a train to reverse direction, e.g. at a station or at the end of a line.

The methodology of building a framework for verifying systems in a specific domain is of interest. As future work, the framework should be adapted to other domains in-order to aid the development of verified critical systems. This issue has only been hinted at with the introduction of the Pelicon crossing example, but it should be possible for more complicated situations, such as the automotive domain or nuclear power-plants. In situations where there is a significant amount of arithmetic reasoning it would be desirable

to integrate Agda with an SMT solver instead of a SAT solver. However, particular attention would need to be payed to the integration as SMT has an undecidable fragment.

12.2.1 Institution Conjecture

During the work documented in this thesis, it was realised that train control systems could be described by an institution. An institution, as it occurs in [GB84], is defined as:

- Category Sign of Signatures
- Functor $\mathit{sen} : \mathit{Sign} \to \text{Set}$ mapping a signature onto a set of sentences.
- Functor $\mathit{Mod} : \mathit{Sign}^{op} \to \text{Cat}$ mapping a signature onto a category of models.
- For a signature Σ, satisfaction relation $\_\models_{\Sigma}\_ : \mathit{Mod}(\Sigma) \to \mathit{sen}(\Sigma) \to \text{Set}$

However, this idea was never explored in depth and is left as future work. It is conjectured that an institution can be defined, where the signatures are physical layouts (plus a chosen logic signature), the sentences are control tables (moreover a sentence is a formula built using the chosen logic), the models are ladder logic programs, and the satisfaction relation is derived from the correctness proof. That is,

Signatures are given by sets of signals, track segments, points, routes, function symbols and relations. E.g. $\mathit{connected} \subseteq \mathit{track}^2$, $\mathit{isBefore} \subseteq \mathit{signal} \times \mathit{track}$, $\mathit{isProceed} : \mathit{time} \times \mathit{route} \to \text{Bool}$, $\mathit{Occupied} : \mathit{time} \times \mathit{track} \to \text{Bool}$ and an equality: $\equiv$.

Sentences are control tables built by formulæ over the signature. E.g. a portion of a control table could be as follows:

$$\forall t \in \mathit{time} \;.\; \mathit{isProceed}\; t\; \mathit{rt}_1 \to \neg \mathit{Occupied}\; t\; \mathit{ts}_1$$

The category of models are given by ladder logic programs, and the satisfaction relation means that the ladder logic program correctly implements the control table.

Such a framework would allow theorems from one layout to be translated into another layout, and more generally translate theorems between the railway domain and another logic (possibly first-order). See Figure 12.1.

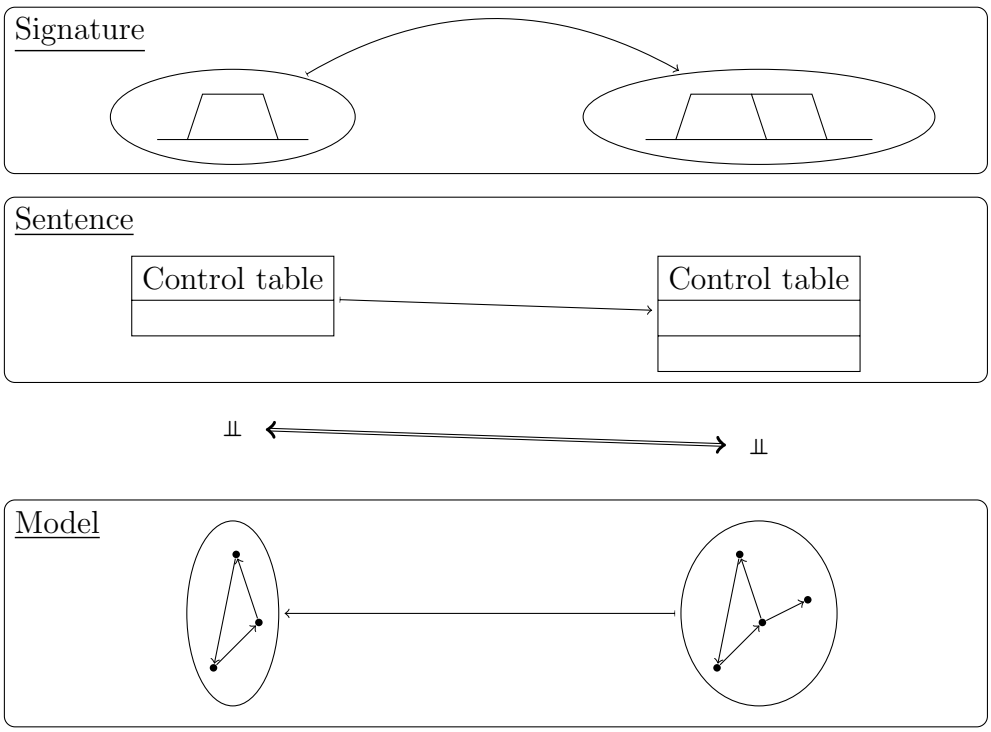


Figure 12.1: Railyard Institution. The maps show how the signatures, sentences and models are translated between rail yards. The equivalence shows how theorems are translated between rail yards using the satisfaction relation.

Appendices

Appendix A
Railway Terminology

In this appendix a selection of railway terminology relevant to this thesis is explained, together with page numbers to where they are introduced.

ATO Acronym for Automatic Train Operation.

ATP Acronym for Automatic Train Protection.

automatic train operation A system that will automatically (without input from the driver), drive the train. This is usually referred to as ATO. 34

automatic train protection A system that is responsible for protecting the train, typically by preventing the train from exceeding the maximum authorised speed (possibly 0). This is usually referred to as ATP. 34

berth segment The segment directly preceeding the main signal of a route, it is where trains wait while the signal displays a danger aspect. 171

block Block signalling is when a railway line is devided up into a number of blocks, such that only one train is allowed to enter each block. 187

call-on A Special type of shunting signal that allows for trains to be signalled into occupied routes. Used when coupling two trains. 162

caution A signal aspect. Indicates that the line ahead (of the signal) is safe to traverse at a reduced speed. This could be the case to warn the train that the succeeding signal along the line is at danger. 164

clear Synonymous with proceed. 178

danger A signal aspect. Indicates that the line ahead (of the signal) would be dangerous to enter. E.g. it is occupied by another train or the sets of points are not in the correct positions. 164

distant signal A signal that gives advanced warning of a main signal, this is because trains require sufficient warning if they must reduce their speed. 163

dog See tappet. 261

facing To describe the direction of travel from the perspective of the train driver, two terms are used: facing and trailing. These terms are typically used to refer to signals and sets of points. A facing signal is the direction when a train must obey it, and a facing move over a set of points is when the train is on the main line and travelling onto one of the (two) branch lines. Conversely a tailing signal is when the front of the signal is not visible and a trailing move over a set of points is when travelling from one of the branch lines on to the main line. 164

facing-point lock Locks a set of points into position when a train travels over them in the facing direction. It is often abbreviated to FPL. 174

FPL Acronym for facing-point lock. 293

main signal Signal at the beginning of a route. 163

normal Refers to the default position of a set of points as determined by the scheme plan. A set of points can be in one of two positions, the other position is called reverse. Historically these terms originated from mechanical interlocking systems, where they referenced the position of a lever. That is, normal is when the lever is in the default position, and reverse is when the lever was in the opposite position. 163

proceed A signal aspect. Indicates that the line ahead (of the signal) is safe to traverse (at maximum speed). E.g. it is not occupied by another train and the sets of points are in the correct positions. 164

railyard Part of a railway that consists of more than linear lines. Typically a railyard is a station, junction or deport. 163

reverse Opposite of normal. 163

rolling stock An individual vehicle on the railway, could either be a carriage or engine. 186

route A sequence of connected track segments that are protected by a signal. 167

route indicator When a signal protects multiple routes, a route indicator is also present that indicates which route the proceed aspect refers to. 227

scheme plan A graphical representation of the layout, of part of a railway. 251

set of points Special track segment that connects to three track segments. One of these segments is designated a main line, and the other two are designated branch lines, it allows travel between the main line and one of the branch lines, the choice of which branch line depends upon the position of the set of points. Also see normal and revers. 163

shunt Special type of signal that is used to manually signal trains around a railyard. They must not be obeyed by passenger trains, therefore safety requirements are not as strict. 162

stretcher bar Part of a set of points. A metal bar that keeps the point blades the correct distance apart. It is usually directly connected to the points machine and it is where the facing-point lock is located. 256

tappet A special metal blade that forms part of a mechanical interlocking. They have notiches cut on the side that allow for dogs to fit in and lock the tappet in place. The dogs are special nuts and bolts that the tappets can move. 260

track circuit A system used to detect discrete train positions, typically each track segment has one track circuit. 255

track segment Smallest unit of indistinguishable track. Often corresponds to a track circuit. 163

trailing Opposite of facing. 173

Appendix B
Mechanical Ladder

In Chapter 11 it was shown how to translate mechanical interlocking systems (lever frames) defined by locking tables into ladder logic programs. In this appendix it is considered how interlocking systems defined by locking tables and ladder logic differ. This is achieved by showing how to simulate ladder logic programs with a lever frame. As lever frames are constrained by allowing only one variable to change at a time, there is a protocol to use these resulting programs such that they are equivalent to ladder logic programs.

B.1 Prerequisites

The lever frames consist of a number of levers that are in either normal or reverse positions. When in normal position the lever is said to be false, and when in reverse position the lever is said to be true. Constraints between the levers are formed by the tappets and dogs. A full explanation of the operation of these interlocking systems is given in Section 11.1.2. In this appendix it is only required to consider the constraints: *locks normal* and *released by*. The constraint l 'locks normal' m, means that when lever l is in the reverse position, then lever m is constrained to the normal position. The constraint is formalised as follows:

$$l \rightarrow \neg m$$

The other constraint: l 'released by' m. Means that when l is reversed, then m is also reversed. The constraint is formalised as follows:

$$l \rightarrow m$$

The state of an interlocking with n levers, is given by a vector of Boolean values of length n. The inputs are given by the levers being moved. Note that each lever is an input, as well as an output. By convention the initial

state is given by all levers being in the normal position, this is possible due to the implementation (see below) of the two mentioned constraints.

These interlocking systems are specified by locking tables. A locking table simply lists the locks, see Figure 11.14 for an example table, however ignore the *both ways* lock as it is not required here.

Ladder Logic. For the purposes of this appendix, ladder logic is a linear system of Boolean valued equations that specify a transition function. The states and inputs of a ladder logic program are given by vectors of Boolean values. A ladder logic program can be though of as the following imperative program:

Initialise $(a_0, \ldots, a_n)$
Repeat {
  Read Inputs $(b_0, \ldots, b_m)$
  $a_0 \coloneqq \phi_0$
    $\vdots$
  $a_n \coloneqq \phi_n$
  Write Outputs
}

where a_i are Boolean valued state variables, b_i are Boolean valued input variables, and ϕ_i are propositional logic formulæ that depend on a_i and b_i. See Section 9.2 for a fully formal treatment of ladder logic.

It is clear that to encode ladder logic programs into lever frames, a method of evaluating Boolean valued equations is required. In the following it will be shown how to implement negation and conjunction using the 'released by' and 'locks normal' locks.

B.1.1 Negation

First it is demonstrated how the lock 'locks normal' is physically implemented. The 'locks normal' constraint is implemented by two tappets and one dog, see Figure B.1. See Chapter 11 for information about how these tappets and dogs interact. There are three possible states: $\neg m \wedge \neg l$, $m \wedge \neg l$, $\neg m \wedge l$. The final combination is not possible due to the dog interlocking the two levers, this is illustrated in the induced transition system in Figure B.2.

It is not possible to move more than one lever at a time, therefore it is not possible to enforce the constraint $l \leftrightarrow \neg m$ without any side conditions on the order that the levers are operated in, i.e. a protocol. For example in Figure B.1, consider treating m as a Boolean input, l as a Boolean output,

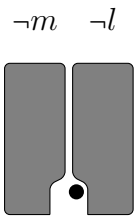


Figure B.1: Locks Normal

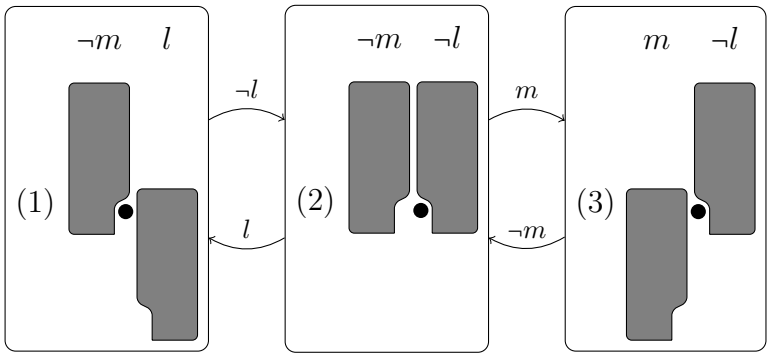


Figure B.2: Locks Normal Transition System

and the human operator pulling on the levers as a computation or evaluation. In such a situation, the following process is obtained to compute the negation:

1. Move all levers (m and l) into the normal position, state (2) in Figure B.2.
2. Move the input lever (m) to the desired configuration, normal or reverse; either stay in state (2) or move to state (3).
3. Attempt to move the output lever l into the reverse position, if it moves, then m is normal, otherwise it does not move because m is in the reverse position.

Thus provided the levers are initialised and pulled in the correct order, the constraint that $l \leftrightarrow \neg m$ holds, equivalently $l := \neg m$.

Remark

Later the issue of ordering the levers from left to right is discussed in depth.

B.1.2 Conjunction

The 'released by' lock is used to implemented conjunction. The 'released by' lock is also implemented by two tappets and one dog, see Figure B.3. There are three possible states: $\neg m \wedge \neg l$, $m \wedge \neg l$, $m \wedge l$. The final combination is not possible due to the dog interlocking the two levers. See Figure B.4 for an illustration of the induced transition system.

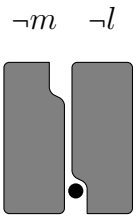


Figure B.3: Released By

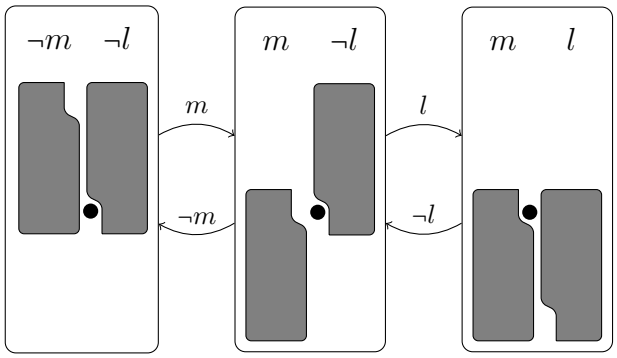


Figure B.4: Released By Transition System

A singular use of the 'released by' constraint is not particularly helpful with respect to this encoding of propositional logic. But two applications of this constraint allows for conjunction to be defined. That is, consider three levers l, m and n, such that $l \leftrightarrow m \wedge n$ is desired to hold. This is achieved by a protocol on the order the levers are moved in, and defining l 'released by' m, and l 'released by' n, i.e. $(l \rightarrow m) \wedge (l \rightarrow n)$. This is implemented by three tappets and two dogs, one for each constraint, see Figure B.5. The idea is similar to before:

1. Move all levers (n, m and l) into the normal position.

2. Configure the input levers (n and m) into normal or reverse positions.

3. Attempt to move l into the reverse position. Should l move, then n and m have been placed into the reverse positions, otherwise at least one of them is in the normal position.

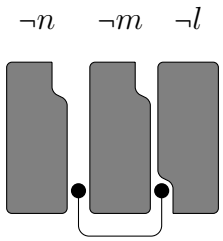


Figure B.5: Mechanical Conjunction

In the above it has been shown how to encode negation and conjunction. Note that Boolean formulæ are equivalent to formulæ formed from these two connectives, see next section. However there are side conditions as to how this embedding is done, these conditions were implicit in the orders that the levers are pulled in.

Protocol. Computation of negation and conjunction was the 3-step process: 1) initialise the levers by placing them in the normal position, 2) set the input levers into a desired position, 3) compute the result by starting on the left (after the rightmost input) and working rightwards, attempt to pull each lever in turn. This process generalises to arbitrary compositions of negation and conjunction.

Implicit in the above protocol is the assumption that all the input levers are assumed to be placed on the *left*, then proceeding on the right are combinations of levers that represent negation or conjunction. The inputs and results of previous operations can become inputs to the next operation. Thus the *furthest right* lever represents the result of the computation.

B.2 Encoding Ladder Logic

It is now shown how to encode ladder logic programs as lever frames. Consider a simple ladder with one rung of the form:

$$x := \varphi(i_1, \ldots, i_n)$$

where i_j are input variables that φ depends on. To encode this ladder, φ must be translated into a formula built out of negation and conjunction. For example consider disjunction, let $n = 2$ and

$$\varphi = \neg(\neg i_1 \wedge \neg i_2)$$

This is still too complicated to be represented directly on a lever frame, so it is broken down into its sub-formulæ. That is the following ladder is obtained:

$$\begin{aligned} x_1 &:= \neg i_1 \\ x_2 &:= \neg i_2 \\ x_3 &:= x_1 \wedge x_2 \\ x_4 &:= \neg x_3 \end{aligned}$$

which is simple enough to be directly represented on a lever frame. This is because each operation is computed individually, results from computed operations are fed into subsequent operations as inputs. This example will be returned to, but for now see Figure B.6 for the corresponding interlocking system.

B.2.1 Remarks

It has been shown how to encode propositional logic, and ladder logic into these lever frames, provided they are operated according to the identified protocol. This process would scale, it is noted again that as the results of intermediate steps in the computation are available for later parts of the computation, so there is no blow-up during the computation. Also, to simplify the translation it could be required that each of the inputs and conjunctions are immediately followed by their negation.

There is one large caveat here. The protocol requires that before each evaluation of φ that all levers are normalised. This means that all the internal state is forgotten. So for ladder logic programs, the output state must be manually fed back as inputs. This essentially means that the lever frames when combined with the protocol realise a decidable transition function, i.e.

$$\text{transition} : \text{Vec}_{\text{Bool}}\ n \to \text{Vec}_{\text{Bool}}\ m \to \text{Vec}_{\text{Bool}}\ n$$

One should note that there is little practical use for such an encoding, particularly as before each new input can be entered the levers are all normalised. So if it was used as an interlocking system, before every cycle,

all signals would be set to danger, all sets of points placed into the normal position and their facing-point locks unlocked. Interestingly, the underlying problem of not being able to formalise the simple ladder $a := \neg b$ without the protocol shows the limitation of lever frames, and why they are not able to support track circuits without supplementary electromechanical locks.

There is also an issue of time. For the translated ladder logic programs to be equivalent to ladder logic programs the human operator would need to complete the evaluation of a transition in milliseconds.

B.3 Disjunction

To conclude this appendix, and make the ideas introduced above clear. The continuation of the previous example of disjunction is fully worked out. There are two inputs i_1 and i_2; and four state variables $x_1 \ldots x_4$. By the protocol the levers are arranged from left to right as follows:

$$i_1, i_2, x_1, x_2, x_3, x_4$$

The constraints required to implement disjunction are given in the following locking table:

Lever	Locks Normal	Released By
i_1	–	–
i_2	–	–
x_1	i_1	–
x_2	i_2	–
x_3	–	x_1, x_2
x_4	x_3	–

Or equivalently formalised as follows:

$$x_1 \to \neg i_1 \qquad x_3 \to x_1 \wedge x_2 \qquad x_2 \to \neg i_2 \qquad x_4 \to \neg x_3$$

Here it is clear that $x_4 \to \neg(\neg i_1 \wedge \neg i_2)$ follows. Provided that the protocol is obeyed we also get $\neg(\neg i_1 \wedge \neg i_2) \to x_4$.

Consider the lever frame in Figure B.6, it is in the initial configuration. In the following, two cases of inputs are considered, first, when i_1 is set to true and i_2 is false, and secondly when i_1 and i_2 are both false. The first case, after setting the inputs, x_1 does not move as i_1 has moved, x_2 moves and pushes the dogs to the left. Then x_3 is constrained because x_1 is not reversed,

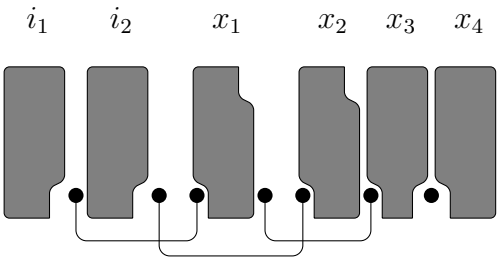


Figure B.6: Mechanical Disjunction

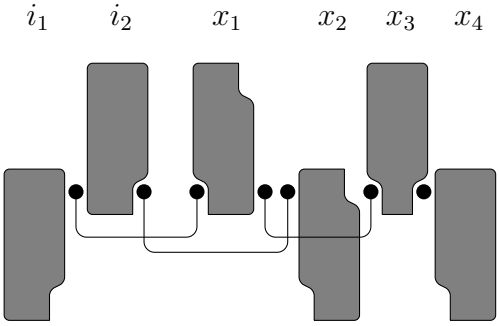


Figure B.7: Mechanical Disjunction. Shows the result of executing the disjunction where the input is $i_1 \wedge \neg i_2$.

therefore the output x_4 moves into the reverse position, representing true. See Figure B.7 for an illustration.

The case where the input is $\neg i_1 \wedge 1_2$ is symmetric. A more interesting case is when the input is $\neg i_1 \wedge \neg i_2$. As neither i_1 or i_2 were moved, then both x_1 and x_2 move to the reverse position. As x_1 and x_2 have been moved, x_3 also moves, therefore it locks the output x_4 in to the normal position. See Figure B.8 for an illustration.

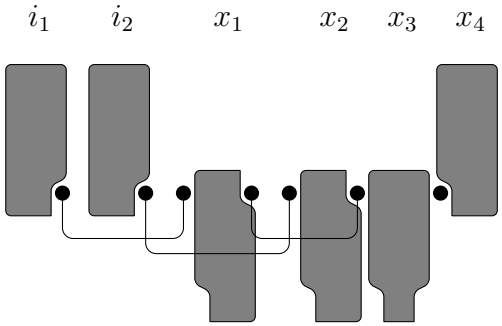


Figure B.8: Mechanical Disjunction. Shows the result of executing the disjunction where the input is $\neg i_1 \wedge \neg i_2$.

Appendix C

Pelicon Simulator Output

In this appendix there is a transcript of an execution of the pelicon simulator. The simulation outputs the state of the ladder and positions of the cars/pedestrians on each cycle, and takes as input one Boolean indicating if the crossing has been requested, and 4 numbers indicating the number of users approaching each of the 4 areas. Refer to Sections 1.1.1 and 10.4 for more information.

Remark

This simulation is slightly more complicated than the Gwili Steam Railway simulation in that it also simulates the architectural state.

The following simulation starts off in a state where all variables are false, then it requests a Boolean input that determines whether crossing has been requested by a pedestrian, also 4 numbers are requested that indicate the number of users approaching in each of the 4 perimeter areas: P1 P2 T1 T2. The ladder is executed once, and the position of the cars/pedestrians are evaluated using the functions defined in Section 10.4, which were proved to be correct. The process then repeats.

The simulation is given the following sequences of inputs:

Requested	T1	T2	P1	P2
no	5	4	2	67
yes	0	4	5	9
no	0	0	0	0
no	0	0	0	0

```
Pelicon Crossing Simulator 0.01
entering main loop...

Crossing: False
Requested: False
Car Green: False
Pedestrian Green: False

Area    People  Cars
P1      0       0
P2      0       0
T1      0       0
T2      0       0
MUX     0       0

P1->MUX: 0   P2->MUX: 0
MUX->P1: 0   MUX->P2: 0
T1->MUX: 0   T2->MUX: 0
MUX->T1: 0   MUX->T2: 0


Request Crossing [yes/no]
no
Enter no. cars embarking T1
5
Enter no. cars embarking T2
4
Enter no. people embarking P1
2
Enter no. people embarking P2
67
Crossing: False
Requested: False
Car Green: True
Pedestrian Green: False

Area    People  Cars
P1      2       0
P2      67      0
T1      0       5
T2      0       4
MUX     0       0

P1->MUX: 0   P2->MUX: 0
MUX->P1: 0   MUX->P2: 0
T1->MUX: 0   T2->MUX: 0
MUX->T1: 0   MUX->T2: 0


Request Crossing [yes/no]
yes
Enter no. cars embarking T1
0
Enter no. cars embarking T2
4
Enter no. people embarking P1
5
Enter no. people embarking P2
9
Crossing: False
Requested: True
Car Green: False
Pedestrian Green: False

Area    People  Cars
P1      7       0
P2      76      0
T1      0       0
T2      0       4
MUX     0       9

P1->MUX: 0   P2->MUX: 0
MUX->P1: 0   MUX->P2: 0
T1->MUX: 5   T2->MUX: 4
MUX->T1: 0   MUX->T2: 0


Request Crossing [yes/no]
no
Enter no. cars embarking T1
0
Enter no. cars embarking T2
0
Enter no. people embarking P1
0
Enter no. people embarking P2
```

```
0
Crossing: True
Requested: False
Car Green: False
Pedestrian Green: True

Area    People  Cars
P1      7       0
P2      76      0
T1      0       0
T2      0       4
MUX     0       0

P1->MUX: 0   P2->MUX: 0
MUX->P1: 0   MUX->P2: 0
T1->MUX: 0   T2->MUX: 0
MUX->T1: 4   MUX->T2: 5


Request Crossing [yes/no]
no
Enter no. cars embarking T1
0
Enter no. cars embarking T2
0
Enter no. people embarking P1
0
Enter no. people embarking P2
0
Crossing: False
Requested: False
Car Green: True
Pedestrian Green: False

Area    People  Cars
P1      0       0
P2      0       0
T1      0       0
T2      0       4
MUX     83      0

P1->MUX: 7   P2->MUX: 76
MUX->P1: 0   MUX->P2: 0
T1->MUX: 0   T2->MUX: 0
MUX->T1: 0   MUX->T2: 0
```

END SIMULATION

Appendix D

Agda How-To: Built-ins

In this user guide it is described how to add new built-in data-types and functions to Agda by hand, mainly by using examples. The built-in data-types are required when implementing the specific branches of the integration as described in Chapter 5. They are used to translate the formulæ between Agda and Haskell. These specific branches also require built-in functions are added that implement the decision procedure, and execute the external tool.

The generic integration (cf. Section 5.4) mitigates building-in data-types as it requires an Agda function is provided to generate a string in the tools input language from a formula. However, it still requires generic built-in functions to be added: One pseudo built-in that specifies the external tool, one pseudo built-in that specifies the Agda function that translates the input, and it requires a built-in function is added that will execute the external tool.

This guide is concluded with selected listings from Agda's source code that show the modifications made during this project.

It is assumed that the reader is a confident Haskell programmer.

D.1 Building-In Data-Types

Suppose we want to add the following built-in data-type to Agda:

$$\begin{aligned}&\text{data Form} : \text{Set where}\\ &\quad \text{Var} \;:\; \mathbb{N} \to \text{Form}\\ &\quad \text{Or} \;:\; \text{Form} \to \text{Form} \to \text{Form}\\ &\quad \text{Not} \;:\; \text{Form} \to \text{Form}\end{aligned}$$

The file `Agda/TypeChecking/Monad/Builtin.hs` defines all built-in tags, and projections from the type-checking monad that look-up the Agda term tagged by one of these tags. The definition of Form requires adding 4 tags, one for the type and one for each constructor.

So we need to add one entry for Form : Set,

```
builtinMyForm = "MYFORM"
```

and three entries for the three constructors

```
builtinMyFormVar = "MYFORMVAR"
builtinMyFormOr  = "MYFORMOR"
builtinMyFormNot = "MYFORMNOT"
```

Then the corresponding projections for looking-up the tagged terms for MYFORM inside the monad are as follows:

```
primMyForm    = getBuiltin builtinMyForm
primMyFormVar = getBuiltin builtinMyFormVar
primMyFormOr  = getBuiltin builtinMyFormOr
primMyFormNot = getBuiltin builtinMyFormNot
```

Here getBuiltin does the work of looking-up a tagged term from a map stored in the monad. Should no term be associated with the tag it results in a type-checking error.

The next step is to add rules to the type checker so that it knows what the types of the built-ins are. Open `Agda/TypeChecking/Rules/Builtin.hs`. This file specifies rules or checks that are enforced by the type-checker when binding a tag to a term. In this file there is a list coreBuiltins that defines a map between the tags and how to bind these tags to an Agda term. This map is used every time a built-in pragma is parsed. First add the following entry

```
(builtinMyForm |-> BuiltinData tset [builtinMyFormVar,
                                     builtinMyFormOr,
                                     builtinMyFormNot])
```

that will ensure that builtinMyForm is bound to a term that is a data-type of type Set that has three constructors. It does not provide information about the constructors, that is achieved by adding the following three entries to the list

```
(builtinMyFormVar |-> BuiltinDataCons (tnat --> tmyform))
(builtinMyFormOr  |-> BuiltinDataCons (tmyform --> tmyform
                                       --> tmyform))
(builtinMyFormNot |-> BuiltinDataCons (tmyform --> tmyform))
```

where tmyform = el primMyForm. These three entries associate a type to the constructors. A mini domain specific language is used to represent the types in a human readable way, notably the DSL provides two functions hPi (hidden) and nPi (normal) that produce Π-types. The hidden Π-type is used to represent hidden arguments, and the normal Π-type is used to represent unhidden arguments. It should be noted that when binding builtinMyFormVar

to a term, first the natural numbers must have been declared as built-ins as well.

Once the above has been implemented the built-ins have been defined, and the built-in pragmas

```
{-# BUILTIN MYFORM      Form #-}
{-# BUILTIN MYFORMVar Var   #-}
{-# BUILTIN MYFORMOr   Or    #-}
{-# BUILTIN MYFORMNot Not   #-}
```

will work. These built-ins are only of use if one defines built-in functions upon them.

D.2 Primitives

Before defining a built-in function over MYFORM, it is necessary to provide a primitive function that will become the Haskell implementation of the built-in function.

To create a new primitive function that uses MYFORM, the translations from Agda's internal syntax into Haskell definitions must be defined. An overview of the internal syntax can be found in [Tur10]. Also the inverse translation might be required depending whether the primitive function returns a MYFORM structure.

The file `Agda/TypeChecking/Primitive.hs` provides the definitions for the translations and primitive functions. To define the translations, first, a corresponding Haskell definition is given as follows:

```
data MyForm = MyVar Integer
            | MyAnd MyForm MyForm
            | MyNot MyForm
```

Then it is added to the PrimTerm type-class. The class is defined as:

```
class PrimTerm a where primTerm :: a -> TCM Term
```

and is added by

```
instance PrimTerm MyForm where primTerm _ = primMyForm
```

which indicates that the data-type MyForm has a corresponding Agda implementation.

Note the use of the monad TCM, this is the type-checking monad that forms the basis of the Agda program.

To translate from the Haskell term into an Agda term, the following function can be defined. The first three arguments are Agda terms that represent the constructors, these are the terms tagged by the built-in pragmas.

```
toterm :: Term -> Term -> Term -> MyForm -> Term
toterm var or not (MyVar n) =
  apply var [Arg NotHidden (Lit $ LitInt noRange n)]
toterm var or not (MyOr a b) =
  apply
   (apply or [Arg NotHidden (toterm var or not a)])
   [Arg NotHidden (toterm var or not b)]
toterm var or not (MyNot a) =
  apply not [Arg NotHidden (toterm var or not a)]
```

Then add it to the ToTerm type-class as:

```
instance ToTerm MyForm where
    toTerm = do
      var <- primMyFormVar
      or  <- primMyFormOr
      not <- primMyFormNot
      return $ toterm var or not
```

where the type-class is defined as:

```
class ToTerm a where toTerm :: TCM (a -> Term)
```

The inverse translation going from an Agda term into a Haskell term is a little messier because of the Agda representation (list of lists) of a term. The first three terms are the Agda representations of the constructors, and the fourth is the actual term to be translated. The translation can fail for a number of reasons, e.g. open terms, hence the result is wrapped by Maybe[1].

```
fromterm :: Term -> Term -> Term -> Term -> Maybe MyForm
fromterm var or not t
   | (t == var) = let varid = xvar var t in
      case varid of
        Just id -> case id of
                     Lit (LitInt _ n) -> Just $ MyFormVar n
                     _ -> Nothing
        _ -> Nothing
   | (t == or) = let oparands = xor or t in
      case oparands of
        Just (f,g) -> let f' = fromterm var or not f
                          g' = fromterm var or not g in
                case (f',g') of
                  (Just f'',Just g'') -> Just $ MyOr f'' g''
```

[1]This is not correct, Agda uses a special data-type that expands on Maybe by allowing for partial reductions to be returned in-situ of nothing. For simplicity this user guide only considers the use of Maybe.

```
                _ -> Nothing
            _ -> Nothing
  | (t == not) = let operand = xnot not t in
      case operand of
        Just f -> let f' = fromterm var or not f in
                  case f' of
                    Just f'' -> Just $ MyNot f''
        _ -> Nothing
  | otherwise      = Nothing
 where
  Def x [] == Def y []   = x == y
  Con x _ == Con y _   = x == y
  Var n [] == Var m []   = n == m
  _ == _ = False
```

where xvar, xor and xnot are definable by pattern matching on Agda's internal term syntax, as follows. However be warned that (at the time of writing) the internal syntax is not fixed, and it changes between releases as new features are added.

```
xvar :: Term -> Term -> Maybe Term
xvar (Con x _) (Con y [(Arg _ z)]) | x == y = Just z
xvar _ _ = Nothing

xor :: Term -> Term -> Maybe (Term,Term)
xor (Con x _) (Con y [(Arg _ f),(Arg _ g)]) | x == y
                                            = Just (f,g)
xor _ _ = Nothing

xnot :: Term -> Term -> Maybe (Term,Term)
xnot (Con x _) (Con y [(Arg _ f)]) | x == y = Just f
xnot _ _ = Nothing
```

Then is added to the FromTerm type-class.

```
instance FromTerm MyForm where
    fromTerm = do
      var <- primMyFormVar
      or  <- primMyFormOr
      not <- primMyFormNot
      fromReducedTerm $ fromterm var or not
```

Here the class FromTerm is defined as follows:

```
class FromTerm a where fromTerm :: TCM (Term -> Maybe a)
```

That concludes the hard part, the data-type has been added to the three type-classes that allow Agda to translate to/from the Haskell representation.

D.3 Built-In Functions

To add a primitive function to Agda, it is firstly defined in Haskell. For the purposes of this user guide the following, perhaps convoluted Haskell function is to be declared as a primitive.

```
flip :: MyForm -> MyForm
flip (Var n)  = Var n
flip (Not x)  = Not (flip x)
flip (Or x y) = Or (flip y) (flip x)
```

It recursively flips the operands of the Or constructor. To be able to reference flip from Agda it is added it to the map: primitiveFunctions of primitive functions (located at the end of the file). The following line will achieve this

```
"primMyFormFlip" |-> mkPrimFun1 flip
```

The string on the left provides a name for this function that will later be used from Agda to reference it. On the right-hand side of the |–> is the primitive implementation of the function. The function mkPrimFun1 constructs a primitive function out of a Haskell function, the number at the end indicates the number of arguments that the function has. There are other versions of this function that consider functions with up to 4 arguments. The function mkPrimFun1 requires that the type of the function provided is in the to/from term type-classes. This requirement means that it is possible to automatically deduce the corresponding Agda type of flip. It also automatically applies the to/from term functions to the arguments and result of the function.

Now it is possible to provide the primitive definition in Agda as:

```
primitive
  primMyFormFlip : Form → Form
```

From the perspective of Agda, primMyFormFlip is treated as a black-box, i.e. an atomic computation that when succeeds produces the required result, or when it fails, it does not normalise.

The final step to defining the built-in function is to also provide an implementation in Agda that is used when the primitive function fails. This has the effect that when the built-in function is normalised on a closed term, the computation is atomic and completes in one step. When normalised on an open term the Agda implementation is used to unfold the term to head normal-form.

To preserve consistency in Agda when creating a built-in function, it is necessary to axiomatise the function. The axioms prevent the incorrect Agda

implementation being replaced by a Haskell function. The axiom checks are defined in `Agda/TypeChecking/Rules/Builtin.hs`.

To build-in the flip function, first the function tag and corresponding primitive term are required to be added by the method described at the beginning of this user guide, see Section D.1. That is, add the following to `Agda/TypeChecking/Monad/Builtin.hs`

```
builtinFlip = "MYFLIP"
primFlip = getBuiltin builtinFlip
```

Then return to `Agda/TypeChecking/Rules/Builtin.hs`. Add the following

```
(builtinFlip |-> BuiltinPrim "primMyFormFlip" myCheck)
```

to the coreBuiltins list. The Agda type of the built-in is looked-up from the referenced primitive function. The function myCheck is defined as follows:

```
myCheck :: Term -> TCM ()
myCheck t = myform <- tmyform
            var'   <- primMyFormVar
            or'    <- primMyFormOr
            not'   <- primMyFormNot
            let x @@ y = x `apply` [defaultArg y]
                x == y = noConstraints $ equalTerm myform x y
                flip' a = t @@ a
                var n   = var' @@ n
                or  a b = or' @@ a @@ b
                not a   = not' @@ a
            xs <- mapM freshName_ ["a","b","n"]
            addCtxs xs (defaultArg myform) $ do
              flip' (var n)  == var n
              flip' (not a)  == not (flip' a)
              flip' (or a b) == or (flip' b) (flip' a)
```

The interesting part of myCheck are the last 3 lines, there the function flip is fully axiomatised. The double equals is Agda's internal equality test. It is possible to define significantly more complicated checks, the interested reader is directed to the Agda source code for examples.

Providing all the steps were successfully completed it is now possible to enter an equivalent definition of flip into an Agda module and then the built-in pragma. I.e.

$$
\begin{aligned}
&\text{flip} : \text{Form} \to \text{Form}\\
&\text{flip}\ (\text{Var}\ n) \ = \ \text{Var}\ n\\
&\text{flip}\ (\text{Not}\ x) \ = \ \text{Not}\ (\text{flip}\ x)\\
&\text{flip}\ (\text{Or}\ x\ y) \ = \ \text{Or}\ (\text{flip}\ y)(\text{flip}\ x)\\
&\\
&\{\text{-\# BUILTIN MYFLIP flip \#-}\}
\end{aligned}
$$

The above will attempt to bind the Agda implementation of flip to the Haskell implementation, provided the Agda version fulfils the constraints in myCheck, and that Form and Nat are built-in types.

D.3.1 External Programs

To execute an external program, a small amount of hackery is required. The primitive function provide must call the external program, but this requires that the function is defined as an IO monad. For this reason the mkPrimFun* functions are unusable. To remedy this, it is noted that the TCM monad is also an IO monad, so it is possible to write a custom implementation directly.

To explain the custom implementation, consider the following function that will create the implementation of a primitive function, which executes an external command. The primitive function will take as input a MyForm structure, and pass it to the external tool (after turning it into a string). The result of the function is a Boolean value that reflects whether the command terminated successfully. The function is similar to the functions that are used for the specific branches of the integration to call an external tool.

```
mkExecute :: TCM PrimitiveImpl
mkExecute = do
 toForm <- fromTerm
 bool   <- fmap (El $ mkType 0) primBool
 t <- el primMyForm --> el primBool
 return $ PrimImpl t $ PrimFun __IMPOSSIBLE__ 1 $ \p_t ->
   case p_t of
     [p_t] -> liftTCM $
                redBind (toForm p_t) (\_ -> [notReduced p_t]) $
       \prob -> do
         liftIO $ Sys.getEnv "AGDA_EXECUTE_PERMISSION"
         let
          atp_p :: CreateProcess
          atp_p = CreateProcess (RawCommand "/path/to/tool" [])
                                Nothing Nothing
                                CreatePipe Inherit Inherit
                                True
         (inp,out,err,pid) <- liftIO $ createProcess atp_p
         liftIO $ hPutStrLn (fromJust inp) (show prob)
         liftIO $ hClose (fromJust inp)
         exitcode <- liftIO $ waitForProcess pid
         case exitcode of
          ExitFailure 1 -> primFalse >>= redReturn
          ExitSuccess   -> primTrue >>= redReturn
          ExitFailure n -> typeError $ GenericError $ "Error"
     _ -> __IMPOSSIBLE__
```

The interesting part of the function start at the first liftIO function. The first thing that happens is a check is made to ensure that an environment variable is set, if not then an error is raised. Then a process is created, and the passed in formula is turned into a string and written to the programs standard input stream. It is then waited for this process to terminate, upon so, the exit code of the program is analysed.

The generic version of the integration extends upon this definition by allowing the path to the tool to be extracted from the Agda module, also instead of translating the formula into a string in Haskell it is translated in Agda. The code of the generic version is found in the next section.

D.4 Haskell Code Listings

This section lists a number code snippets, each snippet relates to part of the work that was undertaken to integrate Agda to external tools. Both the generic (cf. Section 5.4) and proof reconstruction (cf. Chapter 6) code is listed. This is by no means a full code listing, as a full code listing would obscure the modifications as some of the files are hundreds of lines. The listings are similar to the code snippets previously shown for adding built-in data types and functions. Please note that the full source code is available in digital format.

The remainder of this section is split into the three source files previously identified. Each subsection then lists the snippets. Of most interest are the two functions defined in Section D.4.3, these are called mkATPDecProc and mkExternal. The first function is used for the generic plug-in interface, and the second function is used for the proof reconstruction interface.

D.4.1 Agda/TypeChecking/Monad/Builtin.hs

```
builtinMaybe        = "MAYBE"
builtinJust         = "JUST"
builtinNothing      = "NOTHING"

builtinUnit         = "UNIT"
builtinTriv         = "TRIV"
builtinEmpty        = "EMPTY"
builtinAtom         = "ATOM"

builtinATPProblem   = "ATPPROBLEM"
builtinATPInput     = "ATPINPUT"
builtinATPDecProc   = "ATPDECPROC"
builtinATPSemantics = "ATPSEMANTICS"
```

```
builtinATPSound      = "ATPSOUND"
builtinATPComplete   = "ATPCOMPLETE"
builtinATPTool       = "ATPTOOL"

primMaybe   = getBuiltin builtinMaybe
primJust    = getBuiltin builtinJust
primNothing = getBuiltin builtinNothing

primUnit = getBuiltin builtinUnit
primTriv = getBuiltin builtinTriv

primEmpty  = getBuiltin builtinEmpty
primAtom   = getBuiltin builtinAtom

primATPProblem   = getBuiltin builtinATPProblem
primATPInput     = getBuiltin builtinATPInput
primATPDecProc   = getBuiltin builtinATPDecProc
primATPSemantics = getBuiltin builtinATPSemantics
primATPSound     = getBuiltin builtinATPSound
primATPComplete  = getBuiltin builtinATPComplete
primATPTool      = getBuiltin builtinATPTool
```

D.4.2 Agda/TypeChecking/Rules/Builtin.hs

```
  , (builtinMaybe              |-> BuiltinData (tset --> tset) [
    ↳builtinJust, builtinNothing])
  , (builtinJust               |-> BuiltinDataCons (hPi "A" tset
    ↳ (tv0 --> tmaybe v0)))
  , (builtinNothing            |-> BuiltinDataCons (hPi "A" tset
    ↳ (tmaybe v0)))
  , (builtinUnit               |-> BuiltinData tset [builtinTriv
    ↳])
  , (builtinTriv               |-> BuiltinDataCons tunit)
  , (builtinEmpty              |-> BuiltinData tset [])
  , (builtinAtom               |-> BuiltinUnknown (Just $ tbool
    ↳--> tset) verifyAtom)
  , (builtinATPProblem         |-> BuiltinUnknown (Just tset) (
    ↳const $ return ()))
  , (builtinATPInput           |-> BuiltinUnknown (Just $
    ↳tproblem --> tstring) (const $ return ()))
  , (builtinATPDecProc         |-> BuiltinPrim "primATPDecProc"
    ↳(const $ primATPTool >> return ()))
  , (builtinATPTool            |-> BuiltinUnknown (Just $
    ↳tstring) (const $ return ()))
  , (builtinATPSemantics       |-> BuiltinUnknown (Just $
    ↳tproblem --> tset) (const $ return ()))
  , (builtinATPSound           |-> BuiltinUnknown (Just $ nPi "q
    ↳" tproblem $ (tatom $ decproc v0) --> tsemantics v0) (
    ↳const $ return ()))
```

```
  , (builtinATPComplete          |-> BuiltinUnknown (Just $ nPi "q
    ↳" tproblem $ tsemantics v0 --> tatom (decproc v0)) (const
    ↳$ return ()))
```

D.4.3 Agda/TypeChecking/Primitive.hs

```
instance PrimTerm a => PrimTerm (Maybe a) where
    primTerm _ = do
      x  <- primMaybe
      a' <- primTerm (undefined :: a)
      return $ x `apply` [defaultArg a']

instance (FromTerm a , ToTerm a) => ToTerm (Maybe a) where
    toTerm = do
      just <- fmap (\ x -> \ a -> x `apply` [defaultArg a])
          ↳primJust
      nothing <- primNothing
      (fromA :: a -> Term) <- toTerm
      return $ \ t -> case t of
                        (Just a) -> just (fromA a)
                        Nothing  -> nothing

instance (ToTerm a , FromTerm a) => FromTerm (Maybe a) where
  fromTerm = do
    just <- isCon =<< primJust
    nothing <- isCon =<< primNothing
    (toA :: FromTermFunction a) <- fromTerm
    return $ \ t -> do
      b <- reduceB t
      let t = ignoreBlocking b
          arg = Arg (argHiding t) (argRelevance t)
      case unArg t of
        Con c [a]
          | c == just -> redBind (toA a) (\ a' -> notReduced $
              ↳arg $ Con c [ignoreReduced a']) $ \ a' ->
              ↳redReturn $ Just a'
        Con c []
          | c == nothing -> redReturn $ Nothing
        _ -> return $ NoReduction (reduced b)

agdaExecutePermission = catchError (liftIO $ Sys.getEnv "
    ↳AGDA_EXECUTE_PERMISSION") (\ _ -> typeError $ GenericError "
    ↳Environment␣variable␣'AGDA_EXECUTE_PERMISSION'␣has␣not␣been␣
    ↳set,␣not␣executing␣tool.")

lookupToolPath :: String -> TCM String
lookupToolPath tool = do
  s <- catchError (liftIO $ Sys.getEnv "AGDA_EXTERNAL_TOOLS") (\
      ↳ _ -> typeError $ GenericError "Environment␣variable␣'
```

```
      ↪AGDA_EXTERNAL_TOOLS' has not been set, it should contain a
      ↪ list of tools and paths or the form tool1=path1;...;tooln
      ↪=pathn")
  case look tool s of
    "" -> typeError $ GenericError $ "Could not find the tool: "
      ↪ ++ tool ++ " in '" ++ s ++ "', please set '
      ↪AGDA_EXTERNAL_TOOLS' to include the tool."
    s -> return s
  where
    look :: String -> String -> String
    look tool [] = []
    look tool s =
      let (a,s') = case List.findIndex (== ';') s of
            Nothing -> (s,"")
            (Just n) -> splitAt n s
          (t',p') = case List.findIndex (== '=') a of
            Nothing -> ("","")
            (Just n) -> splitAt n a
      in if t' == tool then (drop 1 p') else look tool (drop 1 s
         ↪')

mkATPDecProc :: TCM PrimitiveImpl
mkATPDecProc = do
  (toStr :: FromTermFunction Str)       <- fromTerm
  atp          <- fmap (El $ mkType 0) primATPProblem
  input_t      <- primATPInput
  bool         <- fmap (El $ mkType 0) primBool
  tool_t       <- primATPTool
  let input p = input_t `apply` [p]
  t <- el primATPProblem --> el primBool
  return $ PrimImpl t $ PrimFun __IMPOSSIBLE__ 1 $ \p_t ->
    case p_t of
      [p_t] -> liftTCM $ redBind (toStr $ defaultArg $ input p_t
          ↪) (\_ -> [notReduced p_t]) $ \prob -> redBind (toStr $
          ↪ defaultArg tool_t) (\_ -> [notReduced p_t]) $ \tool'
          ↪->
        do
          tool <- lookupToolPath $ unStr tool'
          primATPSound
          primATPComplete
          agdaExecutePermission
          let
            atp_cp :: CreateProcess
            atp_cp = CreateProcess (RawCommand tool []) Nothing
                ↪Nothing CreatePipe Inherit Inherit True
          reportSLn "prim.mkatpdecproc" 2 "Executing ATP Tool"
          reportSLn "prim.mkatpdecproc" 99 $ "Formula for tool:
              ↪" ++ unStr prob
          (inp,out,err,pid) <- liftIO $ createProcess atp_cp
```

```
          ec <- liftIO $ getProcessExitCode pid
          Maybe.maybe (return ()) (\ _ -> typeError $
              ↳GenericError $ "Problem executing external tool, 
              ↳possibly the tool '" ++ unStr tool' ++ "' is 
              ↳wrongly set.") ec
          liftIO $ hPutStrLn (Maybe.fromJust inp) (unStr prob)
          liftIO $ hClose (Maybe.fromJust inp)
          exitcode <- liftIO $ waitForProcess pid
          case exitcode of
            ExitFailure 1 -> primFalse >>= redReturn
            ExitSuccess   -> primTrue >>= redReturn
            ExitFailure n -> typeError $ GenericError $ "ATP 
               ↳Error, exit code: " ++ show n ++ ", see ghci 
               ↳buffer"
      _ -> __IMPOSSIBLE__

mkExternal :: TCM PrimitiveImpl
mkExternal = do
  (toStr :: FromTermFunction Str)       <- fromTerm
  t <- hPi "A" tset $  el primString --> el primString --> el (
     ↳primMaybe <@> varM 0)
  return $ PrimImpl t $ PrimFun __IMPOSSIBLE__ 3 $ \p_t ->
    case p_t of
      [ty_t,tool_t,str_t] -> redBind (toStr $ tool_t) (\s -> [
          ↳notReduced ty_t,s,notReduced str_t]) $ \tool ->
          ↳redBind (toStr $ str_t) (\s -> [notReduced ty_t,
          ↳notReduced tool_t,s]) $ \prob ->
        do
          tool <- lookupToolPath $ unStr tool
          agdaExecutePermission
          let
             atp_cp :: CreateProcess
             atp_cp = CreateProcess (RawCommand tool []) Nothing
                 ↳ Nothing CreatePipe CreatePipe Inherit True
          (inp,out,err,pid) <- liftIO $ createProcess atp_cp
          ec <- liftIO $ getProcessExitCode pid
          Maybe.maybe (return ()) (\ _ -> typeError $
              ↳GenericError $ "Problem executing external tool: "
              ↳ ++ tool ++ ", possibly environment variable 
              ↳AGDA_EXTERNAL_TOOLS is wrongly set.") ec
          result <- liftIO $ do
            hPutStrLn (fromJust inp) (unStr prob)
            hClose (fromJust inp)
            hGetLine (fromJust out)
          reportSLn "prim.mkexternal2" 2 result
          catchError
            (do e <- liftIO $ parse exprParser $ result
                Just s <- getPScope
                e' <- concreteToAbstract s e
```

```
            (primJust <@> checkExpr e' (El (mkType 0) (unArg
              ↳ ty_t))) >>= redReturn) (\ _ -> primNothing
              ↳>>= redReturn)
    _ -> __IMPOSSIBLE__

  , "primATPDecProc"        |-> mkATPDecProc
  , "primExternal"          |-> mkExternal
```

Appendix E

eProver Wrapper Program

This appendix shows the task undertaken by the eProver wrapper script, translating from a list of TSTP inferences into a list of Agda compatible inferences. As an example of the wrapper programs function, consider the following TSTP input

```
fof(ax1,axiom,~(('a' => 'b') <=> (~ 'b' => ~ 'a'))).
```

that declares a first-order formula, with the name 'ax1' that is an axiom. The formula is the theorem that is to be proved. Executing eProver on this input (or more correctly eproof, an eProver script that outputs level 4 derivations), results in the following list of derivations

```
fof(1, axiom,~((('a'=>'b')<=>(~('b')=>~('a')))),file('sample.tptp', ax1)).
fof(2, plain,~((('a'=>'b')<=>(~('b')=>~('a')))),inference(fof_simplification,[status(thm)
    ↳],[1,theory(equality)])).
fof(3, plain,((('a'&~('b'))|(~('b')&'a'))&((~('a')|'b')|('b'|~('a')))),inference(fof_nnf,[status(
    ↳thm)],[2])).
fof(4, plain,((((~('b')|'a')&('a'|'a'))&((~('b')|~('b'))&('a'|~('b'))))&((~('a')|'b')|('b'|~('a'))))
    ↳,inference(distribute,[status(thm)],[3])).
cnf(5,plain,('b'|'b'|~'a'|~'a'),inference(split_conjunct,[status(thm)],[4])).
cnf(7,plain,(~'b'|~'b'),inference(split_conjunct,[status(thm)],[4])).
cnf(8,plain,('a'|'a'),inference(split_conjunct,[status(thm)],[4])).
cnf(12,plain,('b'|$false),inference(rw,[status(thm)],[5,8,theory(equality)])).
cnf(13,plain,('b'),inference(cn,[status(thm)],[12,theory(equality)])).
cnf(14,plain,($false),inference(sr,[status(thm)],[13,7,theory(equality)])).
cnf(15,plain,($false),14,['proof']).
```

Once processed by the wrapper script the following derivation (ProofList) is obtained. In the following the yen symbol (¥) is used to represent a variable, and Boolean connectives are given as &&, ||, and =>. See the definition of a ProofList on page 128 for more information.

```
(node axiom (((~ ((( ¥ 0) => ( ¥ 1)) => ((~ ( ¥ 1)) => (~ ( ¥ 0)))) && (((~ ( ¥ 1))
    ↳=> (~ ( ¥ 0))) => (( ¥ 0) => ( ¥ 1))))) ::[]) ⇒(~ ((( ¥ 0) => ( ¥ 1)) => ((~ (
    ↳ ¥ 1)) => (~ ( ¥ 0)))) && (((~ ( ¥ 1)) => (~ ( ¥ 0))) => (( ¥ 0) => ( ¥ 1))))))
    ↳ []) ::
```

```
(node fof_simplification (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~
    ↳ (¥1)) => (~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒(~ ((((¥0) => (¥1))
    ↳=> ((~ (¥1)) => (~ (¥0)))) && (((~ (¥1)) => (~ (¥0))) => ((¥0) => (
    ↳¥1)))))) (0 ::[])) ::
(node fof_nnf (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (¥1))
    ↳=> (~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒((((¥0) && (~ (¥1))) || ((~ (
    ↳¥1)) && (¥0))) && (((~ (¥0)) || (¥1)) || ((¥1) || (~ (¥0)))))) (1 ::[])) ::
(node distribute (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (¥1)
    ↳) => (~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒(((((~ (¥1)) || (¥0)) && ((¥
    ↳0) || (¥0))) && (((~ (¥1)) || (~ (¥1))) && ((¥0) || (~ (¥1))))) && (((~ (
    ↳¥0)) || (¥1)) || ((¥1) || (~ (¥0)))))) (2 ::[])) ::
(node split_conjunct (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (
    ↳¥1)) => (~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒((((¥1) || (¥1)) || (~ (¥
    ↳0))) || (~ (¥0)))) (3 ::[])) ::
(node split_conjunct (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (
    ↳¥1)) => (~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒((~ (¥1)) || (~ (¥1))))
    ↳(3 ::[])) ::
(node split_conjunct (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (
    ↳¥1)) => (~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒((¥0) || (¥0))) (3 ::[]))
    ↳::
(node rw (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (¥1)) =>
    ↳(~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒((¥1) || ¥false)) (4 ::(6 ::[]))) ::
(node cn (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (¥1)) =>
    ↳(~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒(¥1)) (7 ::[])) ::
(node sr (((~ ((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (¥1)) =>
    ↳(~ (¥0))) => ((¥0) => (¥1))))) ::[]) ⇒ ¥false) (8 ::(5 ::[]))) ::
(node unsat ([] ⇒((((¥0) => (¥1)) => ((~ (¥1)) => (~ (¥0)))) && (((~ (¥1))
    ↳=> (~ (¥0))) => ((¥0) => (¥1))))) (9 ::[])) ::[]
```

E.1 Code Listing

The following is a listing of the Haskell code for the wrapper script.

```
module Main where

import Codec.TPTP.Import
import Codec.TPTP.Base
import Control.Monad.Identity
import qualified Data.Map as Map
import Data.Maybe
import qualified Data.List as List
import qualified System.Directory as Dir
import System.IO
import System.Process
import Data.Generics
import System
import FOLFormula
```

```
{-
Wraps up eProver, parses the output and produces an Agda term.

essentially the process is:
 [TPTP_Input] -> [Record] -> [PNode] -> String
-}

main :: IO ()
main = do
  file <- inputfile
  let atp_cp :: CreateProcess
      atp_cp = CreateProcess (RawCommand ("eproof")
                                         ["--tstp-format",file])
                             Nothing Nothing
                             Inherit CreatePipe Inherit True
  (inp,Just out,err,pid) <- createProcess atp_cp
  rawlines <- fmap parse $ hGetContents out
  if (Comment "# SZS status Unsatisfiable") `elem` rawlines then
      putStrLn $ showAgda $ construct $ step1 rawlines
  else do
    putStrLn "Problem set un-provable."
    hClose out
    exitFailure
  hClose out

type Record = (Int , (PL_FormulaCode,ERuleCode))

data PNode = N { rule :: String
               , pnodeform :: (Proves FOLFormula FOLFormula)
               , seq :: [Int]}
             deriving (Show)

step1 :: [TPTP_Input] -> [Record]
step1 = (map input2record) . (filter filt_AFormula)

input2record x = (input2id x , (input2formula x , input2rule x))

inputfile :: IO String
inputfile = do
  args <- getArgs
  case List.find (\ s -> not $ List.isPrefixOf "--" s) args of
    Nothing -> do
      tmpdir <- Dir.getTemporaryDirectory
      (tmpfile , htmp) <- openTempFile tmpdir "eproof-input"
      hGetContents stdin >>= hPutStr htmp
      hFlush htmp >> hClose htmp >> return tmpfile
    Just s -> return s

construct :: [Record] -> [PNode]
```

```
construct l = let x = buildvarmap (getvars' l) (getstrings' l)
              in discharge $ builds [] l x (Map.empty) []

data PL_FormulaCode =
  Or_ PL_FormulaCode PL_FormulaCode | Var_ (Either String Int) |
  And_ PL_FormulaCode PL_FormulaCode | Neg_ PL_FormulaCode |
  Imp_ PL_FormulaCode PL_FormulaCode | True_ | False_
  deriving (Show,Eq)

gamma :: Proves f f' -> [f]
gamma (Proves a b) = a

phi :: Proves f f' -> f'
phi (Proves a b) = b

data ERuleCode =
  NNF_ Int | Simplify_ Int  | SplitConj_ Int | PM_ Int Int |
  CN_ Int | Axiom_ | UnSat_ Int | Distribute_ Int |
  Apply_ Int [Int] | RW_ Int Int | Void_ Int | SR_ Int Int

binop2formula :: BinOp -> PL_FormulaCode -> PL_FormulaCode ->
   ↳PL_FormulaCode
-- in agda <-> is not implemented
binop2formula (:<=>:) a b = And_ (Imp_ a b) (Imp_ b a)
binop2formula (:=>:) a b  = Imp_ a b
binop2formula (:<=:) a b  = Imp_ b a
binop2formula (:&:) a b   = And_ a b
binop2formula (:|:) a b   = Or_ a b
binop2formula _ a b       = undefined

readatom :: String -> Either String Int
readatom s = case ((reads :: ReadS Int) s) of
  [(n,[])] -> Right n
  _        -> Left s

atom2formula :: AtomicWord -> PL_FormulaCode
atom2formula (AtomicWord atom) | atom == "$true" = True_
                               | atom == "$false" = False_
                               | True = Var_ $ readatom atom

formula02formula :: Formula0 (T Identity) (F Identity) ->
   ↳PL_FormulaCode
formula02formula (BinOp a op b) = binop2formula op (
   ↳formula2formula a) (formula2formula b)
formula02formula (PredApp atom []) = atom2formula atom -- all
   ↳variables and constants are here
formula02formula ((:~:) a) = let a' = (formula2formula a)
                             in case a' of
                                  True_ -> False_
```

```
                                    False_ -> True_
                                    _ -> Neg_ a'
formula02formula _ = undefined

formula2formula :: F Identity -> PL_FormulaCode
formula2formula = formula02formula . runIdentity . runF

input2formula :: TPTP_Input -> PL_FormulaCode
input2formula = formula2formula . formula

tonum :: GTerm -> Int
tonum (GTerm (GNumber n)) = floor n
tonum x = undefined

applydef2rule :: [GTerm] -> [Int]
applydef2rule = let f = (const True) :: Double -> Bool
                in map floor . listify f

inference2rule :: [GTerm] -> ERuleCode
inference2rule [GTerm (GWord (AtomicWord a)) , _ , GList b]
  | a == "fof_nnf" = NNF_ (tonum (head b))
  | a == "fof_simplification" = Simplify_ (tonum (head b))
  | a == "split_equiv" = SplitConj_ (tonum (head b))
  | a == "split_conjunct" = SplitConj_ (tonum (head b))
  | a == "cn" = CN_ (tonum (head b))
  | a == "distribute" = Distribute_ (tonum (head b))
  | a == "apply_def" = let indices = applydef2rule b
                       in Apply_ (head indices) $ tail indices
  | a == "rw" = RW_ (tonum (head b)) (tonum (head (tail b)))
  | a == "eval_answer_literal" = Void_ (tonum (head b))
  | a == "sr" = SR_ (tonum (head b)) (tonum (head (tail b)))
  | a == "pm" = PM_ (tonum (head b)) (tonum (head (tail b)))
inference2rule _ = undefined

annotation2rule :: Annotations -> ERuleCode
annotation2rule NoAnnotations = undefined
annotation2rule (Annotations (GTerm (GApp (AtomicWord n) args))
   ↳NoUsefulInfo)
  | n == "introduced" = Axiom_
  | n == "inference" = inference2rule args
  | n == "file" = Axiom_
annotation2rule (Annotations (GTerm (GNumber n)) (UsefulInfo [(
   ↳GTerm (GWord (AtomicWord m)))])) | m == "proof" = UnSat_ (
   ↳floor n)
annotation2rule (Annotations gt _) = undefined

input2rule :: TPTP_Input -> ERuleCode
input2rule = annotation2rule . annotations
```

```
input2id :: TPTP_Input -> Int
input2id = (\ (AtomicWord x) -> read x) . name

filt_AFormula :: TPTP_Input -> Bool
filt_AFormula (AFormula _ _ _ _) = Prelude.True
filt_AFormula _                  = Prelude.False

filt_Comment :: TPTP_Input -> Bool
filt_Comment (Comment _) = Prelude.True
filt_Comment _ = Prelude.False

instance ShowAgda Int where showAgda = show

instance ShowAgda PNode where
  showAgda (N s f i) = "(node " ++ s ++ " " ++ showAgda f
                       ++ " " ++ showAgda i ++ ")"

next :: [Int] -> Int
next xs = f xs 0
  where f :: [Int] -> Int -> Int
        f xs n | n `elem` xs = f xs (n + 1)
               | otherwise = n

getvars :: PL_FormulaCode -> [Int]
getvars True_      = []
getvars False_     = []
getvars (Or_ a b)  = getvars a ++ getvars b
getvars (And_ a b) = getvars a ++ getvars b
getvars (Imp_ a b) = getvars a ++ getvars b
getvars (Var_ x)   = either (\ _ -> []) (:[]) x
getvars (Neg_ a)   = getvars a

getvars' :: [Record] -> [Int]
getvars' = let f (_ , (formula , _)) = (++) (getvars formula)
           in foldr f []

getstrings :: PL_FormulaCode -> [String]
getstrings True_      = []
getstrings False_     = []
getstrings (Or_ a b)  = getstrings a ++ getstrings b
getstrings (And_ a b) = getstrings a ++ getstrings b
getstrings (Imp_ a b) = getstrings a ++ getstrings b
getstrings (Var_ x)   = either (:[]) (\ _ -> []) x
getstrings (Neg_ a)   = getstrings a

getstrings' :: [Record] -> [String]
getstrings' = let f (_ , (form , _)) = (++) $ getstrings form
              in foldr f []
```

```
buildvarmap :: [Int] -> [String] -> [(String , Int)]
buildvarmap ids []                      = []
buildvarmap ids (s : ss) | s `elem` ss = buildvarmap ids ss
                         | True        =
  let n = Main.next ids in (s , n) : buildvarmap (n : ids) ss

buildvarmap' :: [Int] -> [String] -> Map.Map String Int
buildvarmap' ids ss = Map.fromList $ buildvarmap ids ss

toform :: [(String,Int)] -> PL_FormulaCode -> FOLFormula
toform env True_      = FOLTrue
toform env False_     = FOLFalse
toform env (Or_ a b)  = FOLOr (toform env a) (toform env b)
toform env (And_ a b) = FOLAnd (toform env a) (toform env b)
toform env (Imp_ a b) = FOLImp (toform env a) (toform env b)
toform env (Var_ x)   = FOLVar $ fromIntegral $ either (\ s ->
   ↳maybe (-1) id $ lookup s env) id x
toform env (Neg_ a)   = FOLNeg $ toform env a

unnegate :: FOLFormula -> FOLFormula
unnegate (FOLNeg a)                              = a
unnegate (FOLImp a FOLFalse)                     = a
unnegate (FOLImp a (FOLImp FOLTrue FOLFalse)) = a
unnegate a                                       = a

build :: [PNode] -> [(String,Int)] -> Map.Map Int Int ->
   ↳PL_FormulaCode -> ERuleCode -> [PNode]
build done env m f (NNF_ n)        =
  let a = fromJust (Map.lookup n m)
  in [N "fof_nnf" (Proves (gamma $ pnodeform $ done !! a) $
     ↳toform env f) [a]]
build done env m f (Simplify_ n)   =
  let a = fromJust (Map.lookup n m)
  in [N "fof_simplification" (Proves (gamma $ pnodeform $ done
     ↳!! a) $ toform env f) [a]]
build done env m f (SplitConj_ n)  =
  let a = fromJust (Map.lookup n m)
  in [N "split_conjunct" (Proves (gamma $ pnodeform $ done !! a)
     ↳ $ toform env f) [a]]
build done env m f (CN_ n)         =
  let a = fromJust (Map.lookup n m)
  in [N "cn" (Proves (gamma $ pnodeform $ done !! a) $ toform
     ↳env f) [a]]
build done env m f Axiom_          = [N "axiom" (Proves [toform
   ↳env f] $ toform env f) []]
build done env m f (UnSat_ n)      =
  let f1 = phi $ pnodeform $ head done
  in [N "unsat" (Proves (filter (\ a -> not $ f1 == a) (gamma $
     ↳pnodeform $ done !! fromJust (Map.lookup n m))) $ unnegate
```

```
      ↳ f1) [fromJust (Map.lookup n m)]]
build done env m f (Distribute_ n) =
  let a = fromJust (Map.lookup n m)
  in [N "distribute" (Proves (gamma $ pnodeform $ done !! a) $
      ↳toform env f) [a]]
build done env m f (Apply_ n n')   = buildapply done (fromJust $
    ↳ Map.lookup n m) (map (\ x -> fromJust $ Map.lookup x m) n')
build done env m f (RW_ n n')      =
  let a = fromJust $ Map.lookup n m
      b = fromJust $ Map.lookup n' m
  in [N "rw" (Proves (List.nub ((gamma $ pnodeform $ done !! a)
      ↳++ (gamma $ pnodeform $ done !! b))) $ toform env f) [a,b
      ↳]]
build done env m f (Void_ n)       = [done !! fromJust (Map.
    ↳lookup n m)]
build done env m f (SR_ n n')      =
  let a = fromJust $ Map.lookup n m
      b = fromJust $ Map.lookup n' m
  in [N "sr" (Proves (List.nub ((gamma $ pnodeform $ done !! a)
      ↳++ (gamma $ pnodeform $ done !! b))) $ toform env f) [a,b
      ↳]]
build done env m f (PM_ n n')      =
  let a = fromJust $ Map.lookup n m
      b = fromJust $ Map.lookup n' m
  in [N "pm" (Proves (List.nub ((gamma $ pnodeform $ done !! a)
      ↳++ (gamma $ pnodeform $ done !! b))) $ toform env f) [a,b
      ↳]]

buildapply :: [PNode] -> Int -> [Int] -> [PNode]
buildapply done f [def] | isEquiv $ phi $ pnodeform $ done !!
    ↳def =
  let f'  = pnodeform $ done !! f
      eqv = pnodeform $ done !! def
  in [N "apply_def" (Proves (List.nub ((gamma f') ++ (gamma eqv)
      ↳)) $ subst (phi f') (equivLeft $ phi eqv) (equivRight $
      ↳phi eqv)) [ f , def ]]
buildapply done f (def : defs) | isEquiv $ phi $ pnodeform $
    ↳done !! def =
  let f'  = buildapply done f defs
      eqv = pnodeform $ done !! def
  in f' ++ [N "apply_def" (Proves (List.nub ((gamma $ pnodeform
      ↳$ last f') ++ (gamma eqv))) $ subst (phi $ pnodeform $
      ↳last f') (equivLeft $ phi eqv) (equivRight $ phi eqv)) [
      ↳length (done ++ f') -1 , def ]]

builds :: [Record] -> [Record] -> [(String,Int)] -> Map.Map Int
    ↳Int -> [PNode] -> [PNode]
builds r1 [] env m a = a
```

```
builds r1 (n@(i, (f , r)) : r2) env m a = builds (r1 ++ [n]) r2
    ↪env (Map.insert i (length r1) m) (a ++ build a env m f r)

-- replace all occourances of psi1 in phi by psi2
subst :: FOLFormula -> FOLFormula -> FOLFormula -> FOLFormula
subst phi psi1 psi2 | phi == psi1 = psi2
subst (FOLOr a b) psi1 psi2 =
  FOLOr (subst a psi1 psi2) (subst b psi1 psi2)
subst (FOLAnd a b) psi1 psi2 =
  FOLAnd (subst a psi1 psi2) (subst b psi1 psi2)
subst (FOLImp a b) psi1 psi2 =
  FOLImp (subst a psi1 psi2) (subst b psi1 psi2)
subst (FOLNeg a) psi1 psi2 = FOLNeg (subst a psi1 psi2)
subst x _ _ = x

isEquiv :: FOLFormula -> Bool
isEquiv (FOLAnd (FOLImp (FOLVar a) b) (FOLImp c (FOLVar d))) =
  a == d && b == c
isEquiv _ = False

equivLeft :: FOLFormula -> FOLFormula
equivLeft (FOLAnd (FOLImp a b) (FOLImp c d)) = a

equivRight :: FOLFormula -> FOLFormula
equivRight (FOLAnd (FOLImp a b) (FOLImp c d)) = b

discharge :: [PNode] -> [PNode]
discharge t | (gamma $ pnodeform $ last t) == [] = t
discharge t | isEquiv $ head $ gamma $ pnodeform $ last t =
  discharge $ t ++ [N "fresh" (Proves (tail $ gamma $ pnodeform
     ↪$ last t) $ phi $ pnodeform $ last t) [length t - 1]]
```

Appendix F

Agda Code

This appendix is an Agda code listing for all parts of the thesis. It is split into modules, sorted alphabetically. See the next page for the table of contents, note the different page numbering.

The module hierarchy provides an intuitive guide to which part of the thesis the module relates. Below is a short description of the hierarchy.

Boolean Propositional logic and Boolean formulæ. The naïve SAT solver and built-in bindings. See Sections 4.1 and 5.4.

- **PL-Formula** Modules related to integrating eProver with Agda, also see Proof below.

CTL All variants of CTL model-checking discussed in this thesis. Symbolic model-checking is located in RecordSystem, and LadderCTL is located in Ladder. See Sections 4.2, 4.3 and 9.4.1.

Data.Fin Proof of pigeonhole principle and pairing finite numbers.

Gwili Gwili Steam Railway verification, Chapter 11.

Ladder Ladder logic definition, decidability and correctness. See Chapter 9

Pelicon Fully worked example of pelicon crossing.

Proof Rule systems and proof reconstruction. The eProver interface is located here. See Chapter 6.

RDM Railway domain model, see Chapter 8.

Table of Agda Modules

```
{-# OPTIONS --universe-polymorphism #-}

module PropIso where

open import Data.Bool renaming (_∧_ to _∧ᵇ_; _∨_ to _∨ᵇ_)
open import Data.Unit public using (⊤ ; tt)
open import Data.Empty public using (⊥ ; ⊥-elim)
open import Data.Sum
open import Data.Product

open import Relation.Nullary public using (¬_)

open import Function public using (id ; _∘_ ; _$_ ; const)

{- Propositional logic isomorphisim -}

lem-bool-neg-c : (b : Bool) → ¬ T b → T (not b)
lem-bool-neg-c true p  = p tt
lem-bool-neg-c false p = tt

lem-bool-neg-s : (b : Bool) → T (not b) → ¬ T b
lem-bool-neg-s true  ()
lem-bool-neg-s false p = λ x → x

lem-bool-∧-s : (b c : Bool) → T (b ∧ᵇ c) → T b × T c
lem-bool-∧-s true  _ x = tt , x
lem-bool-∧-s false _ ()

lem-bool-∧-c : (b c : Bool) → T b × T c → T (b ∧ᵇ c)
lem-bool-∧-c true  _ (_ , x') = x'
lem-bool-∧-c false _ (() , _)

lem-bool-∨-s : (b c : Bool) → T (b ∨ᵇ c) → T b ⊎ T c
lem-bool-∨-s true  _ _ = inj₁ tt
lem-bool-∨-s false _ x = inj₂ x

lem-bool-∨-c : (b c : Bool) → T b ⊎ T c → T (b ∨ᵇ c)
lem-bool-∨-c true  _ _          = tt
lem-bool-∨-c false b' (inj₁ ())
lem-bool-∨-c false b' (inj₂ y)  = y

{- Propositional logic introduction / elimination rules -}

∨-introl : (a b : Bool) → T a → T (a ∨ᵇ b)
∨-introl true  _ p = p
∨-introl false _ ()

∨-intror : (a b : Bool) → T b → T (a ∨ᵇ b)
∨-intror true  true  p = p
∨-intror false true  p = p
∨-intror _     false ()

∨-elim : ∀ {C : Set} → {a b : Bool} → (T a → C) → (T b → C) → T (a ∨ᵇ b) → C
∨-elim {_} {a} {b} f g p = [ f , g ]' (lem-bool-∨-s a b p)

uncurry' : ∀ {A : Set} {B : A → Set} {C : Set} → ((x : A) → (y : B x) → C) → ((p : Σ A B) → C)
uncurry' = uncurry

∧-elim : {C : Set} → {a b : Bool} → (T a → T b → C) → T (a ∧ᵇ b) → C
∧-elim {C} {a} {b} f p = uncurry' f (lem-bool-∧-s a b p)

#uncurry : ∀ {a b c} {A : Set a} {B : A → Set b} {C : Σ A B → Set c}
         → ((x : A) → (y : B x) → C (x , y)) → ((p : Σ A B) → C p)
#uncurry f π = f (proj₁ π) (proj₂ π)

∧-swap : ∀ a b → T (a ∧ᵇ b) → T (b ∧ᵇ a)
∧-swap true  true  = id
∧-swap true  false = id
∧-swap false true  = id
∧-swap false false = id

∨-swap : ∀ a b → T (a ∨ᵇ b) → T (b ∨ᵇ a)
∨-swap true  true  = id
∨-swap true  false = id
∨-swap false true  = id
∨-swap false false = id

f∨g : {a b c d : Bool} → (T a → T c) → (T b → T d) → T (a ∨ᵇ b) → T (c ∨ᵇ d)
f∨g {a} {b} {c} {d} f g = ∨-elim (λ p → ∨-introl c d (f p)) (λ p → ∨-intror c d (g p))

∧-intro : (a b : Bool) → T a → T b → T (a ∧ᵇ b)
∧-intro true  true  c  d = tt
∧-intro true  false c  ()
∧-intro false b     () d

∧-eliml : {a b : Bool} → T (a ∧ᵇ b) → T a
```

```
∧-eliml {true}  p = tt
∧-eliml {false} ()

∧-elimr : (a : Bool) → {b : Bool} → T (a ∧♭ b) → T b
∧-elimr a     {true}  p = tt
∧-elimr true  {false} ()
∧-elimr false {false} ()

f∧g : {a b c d : Bool} → (T a → T c) → (T b → T d) → T (a ∧♭ b) → T (c ∧♭ d)
f∧g {a} {b} {c} {d} f g p = ∧-intro c d (f (∧-eliml p)) (g (∧-elimr a p))

¬[p∧¬p] : (b : Bool) → ¬ (T b × T (not b))
¬[p∧¬p] true  = proj₂
¬[p∧¬p] false = proj₁

ex-mid : (a : Bool) → T a ⊎ ¬ T a
ex-mid true  = inj₁ tt
ex-mid false = inj₂ id

_→_ : Bool → Bool → Bool
true  → b = b
false → b = true

lem-→-s : (a b : Bool) → T (a → b) → T a → T b
lem-→-s true  b p pa = p
lem-→-s false b p pa = ⊥-elim pa

lem-→-c : (a b : Bool) → (T a → T b) → T (a → b)
lem-→-c true  b p = p tt
lem-→-c false b p = tt

lem-→-intro : (a b : Bool) → T (not a ∨♭ b) → T a → T b
lem-→-intro a b p ta = ∨-elim (\ x → ⊥-elim (¬[p∧¬p] a (ta , x))) id p

lem-→-elim : (a b : Bool) → (T a → T b) → T (not a ∨♭ b)
lem-→-elim true  b f = f tt
lem-→-elim false b f = tt

lem-→ : {A B : Set} → ¬ A ⊎ B → A → B
lem-→ = [ (λ x x' → ⊥-elim (x x')) , (λ z _ → z) ]'

lem-⊎ : {A : Set} {B : Set} {C : Set} → (A ⊎ B) ⊎ C → A ⊎ (B ⊎ C)
lem-⊎ (inj₁ (inj₁ x)) = inj₁ x
lem-⊎ (inj₁ (inj₂ y)) = inj₂ (inj₁ y)
lem-⊎ (inj₂ y)        = inj₂ (inj₂ y)

demorg1 : ∀ a b → T (not (a ∨♭ b)) → T ((not a) ∧♭ (not b))
demorg1 true  b p = p
demorg1 false b p = p

demorg2 : ∀ a b → T ((not a) ∧♭ (not b)) → T (not (a ∨♭ b))
demorg2 true  b ¬ab = ¬ab
demorg2 false b ¬ab = ¬ab

¬∃ : ∀ {A : Set} → (P : A → Set) → ¬ ∃ P → ∀ x → ¬ P x
¬∃ P ¬ep x px = ¬ep (, px)

open import Relation.Binary.PropositionalEquality

Tb : ∀ {b} → T b → b ≡ true
Tb {true}  tt = refl
Tb {false} ()

¬Tb : ∀ {b} → ¬ T b → b ≡ false
¬Tb {true}  x = ⊥-elim (x tt)
¬Tb {false} x = refl

--- these functions should be moved into a nat properties module
open import Data.Nat hiding (_<_)

_==_ : ℕ → ℕ → Bool
zero    == zero    = true
(suc n) == (suc m) = n == m
_       == _       = false

{-# BUILTIN NATEQUALS _==_ #-}

trans-== : ∀ k l m → T (k == l) → T (l == m) → T (k == m)
trans-== zero    l       zero     k=l l=m = tt
trans-== zero    zero    (suc n)  k=l l=m = l=m
trans-== zero    (suc n) (suc n') k=l l=m = k=l
trans-== (suc n) zero    zero     k=l l=m = k=l
trans-== (suc n) (suc n') zero    k=l l=m = l=m
trans-== (suc n) zero    (suc n') k=l l=m = ⊥-elim l=m
trans-== (suc n) (suc n') (suc n0) k=l l=m = trans-== n n' n0 k=l l=m

sym-== : ∀ k l → T (k == l) → T (l == k)
```

```
sym-== zero    zero    = id
sym-== zero    (suc l) = id
sym-== (suc k) zero    = id
sym-== (suc k) (suc l) = sym-== k l

lift-== : ∀ n m → T (n == m) → n ≡ m
lift-== zero    zero    p = refl
lift-== (suc n) (suc m) p = cong suc (lift-== n m p)
lift-== zero    (suc m) ()
lift-== (suc n) zero    ()

id-== : ∀ n → T (n == n)
id-== zero    = tt
id-== (suc n) = id-== n

+r-== : ∀ n m → ¬ T (n == (n + (suc m)))
+r-== zero    m = id
+r-== (suc n) m = +r-== n m

open import Data.Nat.Properties using (isCommutativeSemiring)
import Algebra.Structures as Alg
open Alg.IsCommutativeSemiring using (+-isCommutativeMonoid)
open Alg.IsCommutativeMonoid using (comm)

+-comm : ∀ n m → n + m ≡ m + n
+-comm n m = comm (+-isCommutativeMonoid isCommutativeSemiring) n m

+l-== : ∀ n m → ¬ T (n == ((suc m) + n))
+l-== zero    m = id
+l-== (suc n) m rewrite +-comm m (suc n)
                      | +-comm n m
  = +l-== n m

_<_ : ℕ → ℕ → Bool
n     < zero   = false
zero  < suc n' = true
suc n < suc n' = n < n'

{-# BUILTIN NATLESS _<_ #-}

<-ord : ∀ n → T (n < suc n)
<-ord zero    = tt
<-ord (suc n) = <-ord n

<-trans : ∀ k l m → T (k < suc l) → T (l < suc m) → T (k < suc m)
<-trans zero    l       m       k<l l<m = tt
<-trans (suc n) zero    zero    k<l l<m = ⊥-elim k<l
<-trans (suc n) (suc n') zero    k<l l<m = ⊥-elim l<m
<-trans (suc n) zero    (suc n') k<l l<m = ⊥-elim k<l
<-trans (suc n) (suc n') (suc n0) k<l l<m = <-trans n n' n0 k<l l<m

<-lsuc : ∀ n m → T (suc n < m) → T (n < m)
<-lsuc zero    zero    p = ⊥-elim p
<-lsuc zero    (suc n) p = tt
<-lsuc (suc n) zero    p = ⊥-elim p
<-lsuc (suc n) (suc n') p = <-lsuc n n' p

<-rsuc : ∀ n m → T (n < m) → T (n < suc m)
<-rsuc zero    zero    p = ⊥-elim p
<-rsuc zero    (suc n) p = tt
<-rsuc (suc n) zero    p = ⊥-elim p
<-rsuc (suc n) (suc n') p = <-rsuc n n' p

<-trans' : ∀ k l m → T (k < l) → T (l < m) → T (suc k < m)
<-trans' k       l       zero          p q = ⊥-elim q
<-trans' k       zero    (suc m)       p q = ⊥-elim p
<-trans' zero    (suc l) (suc zero)    p q = q
<-trans' zero    (suc l) (suc (suc m)) p q = tt
<-trans' (suc k) (suc l) (suc m)       p q = <-trans' k l m p q

<-¬ : ∀ n m → ¬ T (n < m) → T (m < suc n)
<-¬ zero    zero    p = tt
<-¬ zero    (suc n) p = p tt
<-¬ (suc n) zero    p = tt
<-¬ (suc n) (suc n') p = <-¬ n n' p

<-¬' : ∀ n m → T (n < m) → ¬ T (m < suc n)
<-¬' n       zero    p = const p
<-¬' zero    (suc m) p = id
<-¬' (suc n) (suc m) p = <-¬' n m p

<-≠ : ∀ n m → T (n < m) → ¬ T (n == m)
<-≠ n       zero    p = const p
<-≠ zero    (suc m) p = λ ()
<-≠ (suc n) (suc m) p = <-≠ n m p

<-≢ : ∀ n m → T (n < m) → n ≢ m
```

```
<→≠ n zero p = const p
<→≠ zero (suc m) p = λ ()
<→≠ (suc n) (suc m) p = <→≠ n m p ∘ (cong pred)

<-weaken : ∀ n m l → T (n < m) → T (n < (m + l))
<-weaken n       zero    l = ⊥-elim
<-weaken zero    (suc m) l = id
<-weaken (suc n) (suc m) l = <-weaken n m l

<-¬refl : ∀ n → ¬ T (n < n)
<-¬refl zero    ()
<-¬refl (suc n) p = <-¬refl n p

<¬-weaken : ∀ n m k → ¬ T (n < m) → ¬ T ((n + k) < m)
<¬-weaken n       zero    k ¬n<m n+k<m = n+k<m
<¬-weaken zero    (suc m) k ¬n<m n+k<m = ¬n<m tt
<¬-weaken (suc n) (suc m) k ¬n<m n+k<m = <¬-weaken n m k ¬n<m n+k<m


<-+-rsuc : ∀ n m l → T ((n + suc m) < suc l) → T ((n + m) < l)
<-+-rsuc zero    m l       p = p
<-+-rsuc (suc n) m zero    p = p
<-+-rsuc (suc n) m (suc l) p = <-+-rsuc n m l p

max : ℕ → ℕ → ℕ
max n m with n < m
...| true = m
...| false = n

elim-max : ∀ n m → (P : ℕ → Set) → (fn : ¬ T (n < m) → P n) → (fm : T (n < m) → P m) →  P (max n m)
elim-max n m P fn fm with n < m
...| true = fm tt
...| false = fn id

cong' : ∀ {k m l} {A : Set k} (B : A → Set l) {C : Set m}
      → (F : (a : A) → B a → C)
      → {a a' : A}
      → (eq : a ≡ a')
      → (b : B a)
      → F a b ≡ F a' (subst B eq b)
cong' B F refl b = refl

assembleℕ : (f g : ℕ → Set) → (f 0 → g 0) → ({n : ℕ} → f (suc n) → g (suc n)) → {m : ℕ} → f m → g m
assembleℕ f g bc ih {zero}  = bc
assembleℕ f g bc ih {suc m} = ih
```

```
module Boolean.Formula where

open import Data.Nat hiding (_<_)
open import Data.Bool
open import Data.Unit
open import Data.Empty
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.List

open import Algebra

open import PropIso

open import Relation.Binary.PropositionalEquality

infixr 8 _&&_
infixr 7 _||_

data PL-Formula : Set where
  ¥true ¥false : PL-Formula
  _||_ _&&_ _=>_ : PL-Formula → PL-Formula → PL-Formula
  ¥ : ℕ → PL-Formula

~ : PL-Formula → PL-Formula
~ φ = φ => ¥false

_<=>_ : PL-Formula → PL-Formula → PL-Formula
φ <=> ψ = (φ => ψ) && (ψ => φ)

Env : Set
Env = ℕ → Bool

⟦_⊧_⟧pl : (ξ : Env) → PL-Formula → Set
⟦ ξ ⊧ ¥true ⟧pl = ⊤
⟦ ξ ⊧ ¥false ⟧pl = ⊥
⟦ ξ ⊧ φ || ψ ⟧pl = ⟦ ξ ⊧ φ ⟧pl ⊎ ⟦ ξ ⊧ ψ ⟧pl
⟦ ξ ⊧ φ && ψ ⟧pl = ⟦ ξ ⊧ φ ⟧pl × ⟦ ξ ⊧ ψ ⟧pl
⟦ ξ ⊧ φ => ψ ⟧pl = ⟦ ξ ⊧ φ ⟧pl → ⟦ ξ ⊧ ψ ⟧pl
⟦ ξ ⊧ ¥ v ⟧pl = T (ξ v)

Taut-pl : PL-Formula → Set
Taut-pl φ = ∀ ξ → ⟦ ξ ⊧ φ ⟧pl

elim-pl : {A : Set} → (t f : A) → (v : ℕ → A) → (or and iff : A → A → A) → PL-Formula → A
elim-pl t f v or and iff ¥true     = t
elim-pl t f v or and iff ¥false    = f
elim-pl t f v or and iff (y || y') = or (elim-pl t f v or and iff y) (elim-pl t f v or and iff y')
elim-pl t f v or and iff (y && y') = and (elim-pl t f v or and iff y) (elim-pl t f v or and iff y')
elim-pl t f v or and iff (y => y') = iff (elim-pl t f v or and iff y) (elim-pl t f v or and iff y')
elim-pl t f v or and iff (¥ y)     = v y

_≡pl_ : PL-Formula → PL-Formula → Bool
¥true     ≡pl ¥true     = true
¥true     ≡pl _         = false
¥false    ≡pl ¥false    = true
¥false    ≡pl _         = false
(y || y') ≡pl (z || z') = y ≡pl z ∧ y' ≡pl z'
(y || y') ≡pl _         = false
(y && y') ≡pl (z && z') = y ≡pl z ∧ y' ≡pl z'
(y && y') ≡pl _         = false
(y => y') ≡pl (z => z') = y ≡pl z ∧ y' ≡pl z'
(y => y') ≡pl _         = false
¥ y       ≡pl ¥ z       = y == z
¥ y       ≡pl _         = false

lift-≡pl : ∀ φ ψ → T (φ ≡pl ψ) → φ ≡ ψ
lift-≡pl ¥true     ¥true p       = refl
lift-≡pl ¥true     ¥false    ()
lift-≡pl ¥true     (y || y') ()
lift-≡pl ¥true     (y && y') ()
lift-≡pl ¥true     (y => y') ()
lift-≡pl ¥true     (¥ y)     ()
lift-≡pl ¥false    ¥true     ()
lift-≡pl ¥false    ¥false p      = refl
lift-≡pl ¥false    (y || y') ()
lift-≡pl ¥false    (y && y') ()
lift-≡pl ¥false    (y => y') ()
lift-≡pl ¥false    (¥ y)     ()
lift-≡pl (y || y') ¥true     ()
lift-≡pl (y || y') ¥false    ()
lift-≡pl (y || y') (y0 || y1) p rewrite lift-≡pl y y0 $ proj₁ $ lem-bool-∧-s (y ≡pl y0) _ p
                                      | lift-≡pl y' y1 $ proj₂ $ lem-bool-∧-s (y ≡pl y0) _ p = refl
lift-≡pl (y || y') (y0 && y1) ()
lift-≡pl (y || y') (y0 => y1) ()
lift-≡pl (y || y') (¥ y0)     ()
lift-≡pl (y && y') ¥true      ()
```

```
lift-≡pl (y && y') ¥false      ()
lift-≡pl (y && y') (y0 || y1)  ()
lift-≡pl (y && y') (y0 && y1) p rewrite lift-≡pl y y0 $ proj₁ $ lem-bool-∧-s (y ≡pl y0) _ p
                                      | lift-≡pl y' y1 $ proj₂ $ lem-bool-∧-s (y ≡pl y0) _ p = refl
lift-≡pl (y && y') (y0 => y1)  ()
lift-≡pl (y && y') (¥ y0)      ()
lift-≡pl (y => y') ¥true       ()
lift-≡pl (y => y') ¥false      ()
lift-≡pl (y => y') (y0 || y1)  ()
lift-≡pl (y => y') (y0 && y1)  ()
lift-≡pl (y => y') (y0 => y1) p rewrite lift-≡pl y y0 $ proj₁ $ lem-bool-∧-s (y ≡pl y0) _ p
                                      | lift-≡pl y' y1 $ proj₂ $ lem-bool-∧-s (y ≡pl y0) _ p = refl
lift-≡pl (y => y') (¥ y0)      ()
lift-≡pl (¥ y)     ¥true       ()
lift-≡pl (¥ y)     ¥false      ()
lift-≡pl (¥ y)     (y' || y0)  ()
lift-≡pl (¥ y)     (y' && y0)  ()
lift-≡pl (¥ y)     (y' => y0)  ()
lift-≡pl (¥ y)     (¥ y') p       = cong ¥ (lift-== y y' p)

id-≡pl : ∀ φ → T (φ ≡pl φ)
id-≡pl ¥true     = tt
id-≡pl ¥false    = tt
id-≡pl (y || y') = lem-bool-∧-c (y ≡pl y) _ ((id-≡pl y) , (id-≡pl y'))
id-≡pl (y && y') = lem-bool-∧-c (y ≡pl y) _ (id-≡pl y , id-≡pl y')
id-≡pl (y => y') = lem-bool-∧-c (y ≡pl y) _ (id-≡pl y , id-≡pl y')
id-≡pl (¥ y)     = id-== y

lower-≡pl : ∀ φ ψ → φ ≡ ψ → T (φ ≡pl ψ)
lower-≡pl φ ._ refl = id-≡pl φ

_isSubFormula_ : PL-Formula → PL-Formula → Bool
_isSubFormula_ φ ψ with φ ≡pl ψ
...| true = true
φ isSubFormula ¥true     | false = false
φ isSubFormula ¥false    | false = false
φ isSubFormula (y || y') | false = φ isSubFormula y ∨ φ isSubFormula y'
φ isSubFormula (y && y') | false = φ isSubFormula y ∨ φ isSubFormula y'
φ isSubFormula (y => y') | false = φ isSubFormula y ∨ φ isSubFormula y'
φ isSubFormula ¥ y       | false = false

id-isSubFormula : ∀ φ → T (φ isSubFormula φ)
id-isSubFormula φ rewrite Tb (id-≡pl φ) = tt

envupdate : Env → ℕ → Bool → Env
envupdate ξ n b n' with n == n'
...| true  = b
...| false = ξ n'

lem-envupdate : ∀ ξ n b → envupdate ξ n b n ≡ b
lem-envupdate ξ n b rewrite Tb (id-== n) = refl

eval-pl : Env → PL-Formula → Bool
eval-pl ξ ¥true     = true
eval-pl ξ ¥false    = false
eval-pl ξ (y || y') = eval-pl ξ y ∨ eval-pl ξ y'
eval-pl ξ (y && y') = eval-pl ξ y ∧ eval-pl ξ y'
eval-pl ξ (y => y') = not (eval-pl ξ y) ∨ eval-pl ξ y'
eval-pl ξ (¥ y)     = ξ y

mutual
  lem-eval : ∀ ξ φ → ⟦ ξ ⊨ φ ⟧pl → T (eval-pl ξ φ)
  lem-eval ξ ¥true     = id
  lem-eval ξ ¥false    = id
  lem-eval ξ (y || y') = [ ∨-introl (eval-pl ξ y) _ ∘ lem-eval ξ y ,
                           ∨-intror (eval-pl ξ y) _ ∘ lem-eval ξ y' ]'
  lem-eval ξ (y && y') = lem-bool-∧-c (eval-pl ξ y) _ ∘ Prod.map (lem-eval ξ y) (lem-eval ξ y')
  lem-eval ξ (y => y') = λ p → lem-→-elim (eval-pl ξ y) _ (lem-eval ξ y' ∘ p ∘ lem-eval' ξ y)
  lem-eval ξ (¥ y)     = id

  lem-eval' : ∀ ξ φ → T (eval-pl ξ φ) → ⟦ ξ ⊨ φ ⟧pl
  lem-eval' ξ ¥true     = id
  lem-eval' ξ ¥false    = id
  lem-eval' ξ (y || y') = ∨-elim (inj₁ ∘ lem-eval' ξ y) (inj₂ ∘ lem-eval' ξ y')
  lem-eval' ξ (y && y') = ∧-elim (curry $ Prod.map (lem-eval' ξ y) (lem-eval' ξ y'))
  lem-eval' ξ (y => y') = (λ p → lem-eval' ξ y' ∘ p ∘ lem-eval ξ y) ∘ lem-→-intro (eval-pl ξ y) _
  lem-eval' ξ (¥ y)     = id

exmid-or : {A : Set} → {B : Set} → (A ⊎ ¬ A) × (B ⊎ ¬ B) → (A ⊎ B) ⊎ ¬ (A ⊎ B)
exmid-or (inj₁ x , y)       = inj₁ (inj₁ x)
exmid-or (inj₂ y , inj₁ x)  = inj₁ (inj₂ x)
exmid-or (inj₂ y , inj₂ y') = inj₂ [ y , y' ]'

exmid-and : {A : Set} → {B : Set} → (A ⊎ ¬ A) × (B ⊎ ¬ B) → (A × B) ⊎ ¬ (A × B)
exmid-and (inj₁ x , inj₁ x') = inj₁ (x , x')
exmid-and (inj₁ x , inj₂ y)  = inj₂ (y ∘ proj₂)
exmid-and (inj₂ y , x)       = inj₂ (y ∘ proj₁)
```

```
exmid-fun : {A B : Set} → (A ⊎ ¬ A) × (B ⊎ ¬ B) → (A → B) ⊎ ¬ (A → B)
exmid-fun (a , inj₁ x) = inj₁ (const x)
exmid-fun (inj₁ x , inj₂ y)  = inj₂ (λ x' → y (x' x))
exmid-fun (inj₂ y , inj₂ y') = inj₁ (⊥-elim ∘ y)

ex-mid-pl : (ξ : Env) → (φ : PL-Formula) → ⟦ ξ ⊨ φ || ~ φ ⟧pl
ex-mid-pl ξ ¥true    = inj₁ tt
ex-mid-pl ξ ¥false   = inj₂ id
ex-mid-pl ξ (φ || ψ) = exmid-or  (ex-mid-pl ξ φ , ex-mid-pl ξ ψ)
ex-mid-pl ξ (φ && ψ) = exmid-and (ex-mid-pl ξ φ , ex-mid-pl ξ ψ)
ex-mid-pl ξ (φ => ψ) = exmid-fun (ex-mid-pl ξ φ , ex-mid-pl ξ ψ)
ex-mid-pl ξ (¥ v)    = ex-mid (ξ v)

stbl-pl : (ξ : Env) → (φ : PL-Formula) → ⟦ ξ ⊨ ~ (~ φ) ⟧pl → ⟦ ξ ⊨ φ ⟧pl
stbl-pl ξ φ p = [ id , ⊥-elim ∘ p ]' (ex-mid-pl ξ φ)

demorg : ∀ ξ φ ψ → ⟦ ξ ⊨ ~ (φ && ψ) ⟧pl → ⟦ ξ ⊨ ~ φ || ~ ψ ⟧pl
demorg ξ φ ψ p = stbl-pl ξ (~ φ || ~ ψ) (λ x → p $ stbl-pl ξ φ (x ∘ inj₁) , stbl-pl ξ ψ (x ∘ inj₂))

material-pl : ∀ ξ φ ψ → ⟦ ξ ⊨ φ => ψ ⟧pl → ⟦ ξ ⊨ ~ φ || ψ ⟧pl
material-pl ξ φ ψ f = [ inj₂ ∘ f , inj₁ ]' (ex-mid-pl ξ φ)

material-¬pl : ∀ ξ φ ψ → ⟦ ξ ⊨ ~ (φ => ψ) ⟧pl → ⟦ ξ ⊨ φ && (~ ψ) ⟧pl
material-¬pl ξ φ ψ p = [ ⊥-elim ∘ p ∘ const ,
                         _,_ $ [ id , ⊥-elim ∘ p ∘ (λ ~φ → ⊥-elim ∘ ~φ) ]' (ex-mid-pl ξ φ)
                       ]' (ex-mid-pl ξ ψ)

mkenv : List Bool → Env
mkenv []       n       = false
mkenv (x ∷ xs) 0       = x
mkenv (x ∷ xs) (suc n) = mkenv xs n

lem-mkenv-++-eq : ∀ ξ ξ' ζ n → ξ ++ ζ ≡ ξ' → T (n < length ξ) → mkenv ξ n ≡ mkenv ξ' n
lem-mkenv-++-eq []       ξ'            ζ n       eq   n<ξ = ⊥-elim n<ξ
lem-mkenv-++-eq (x ∷ ξ) .(x ∷ ξ ++ ζ) ζ zero    refl n<ξ = refl
lem-mkenv-++-eq (x ∷ ξ) .(x ∷ ξ ++ ζ) ζ (suc n) refl n<ξ = lem-mkenv-++-eq ξ _ ζ n refl n<ξ

lem-mkenv-++ : ∀ ξ ζ n → T (n < length ξ) → T (mkenv ξ n) → T (mkenv (ξ ++ ζ) n)
lem-mkenv-++ []       ζ n       () q
lem-mkenv-++ (x ∷ ξ) ζ zero    p  q = q
lem-mkenv-++ (x ∷ ξ) ζ (suc n) p  q = lem-mkenv-++ ξ ζ n p q

lem-mkenv-++' : ∀ ξ ζ n → T (n < length ξ) → T (mkenv (ξ ++ ζ) n) → T (mkenv ξ n)
lem-mkenv-++' []       ζ n       () q
lem-mkenv-++' (x ∷ ξ) ζ zero    p  q = q
lem-mkenv-++' (x ∷ ξ) ζ (suc n) p  q = lem-mkenv-++' ξ ζ n p q

bound : ℕ → PL-Formula → Bool
bound n ¥true     = true
bound n ¥false    = true
bound n (y || y') = bound n y ∧ bound n y'
bound n (y && y') = bound n y ∧ bound n y'
bound n (y => y') = bound n y ∧ bound n y'
bound n (¥ y)     = y < n

injbound : ∀ φ n m → T (n < suc m) → T (bound n φ) → T (bound m φ)
injbound ¥true     n m n<m p = tt
injbound ¥false    n m n<m p = tt
injbound (y || y') n m n<m p = f∧g {a = bound n y} (injbound y n m n<m) (injbound y' n m n<m) p
injbound (y && y') n m n<m p = f∧g {a = bound n y} (injbound y n m n<m) (injbound y' n m n<m) p
injbound (y => y') n m n<m p = f∧g {a = bound n y} (injbound y n m n<m) (injbound y' n m n<m) p
injbound (¥ y)     n m n<m p = <-trans (suc y) n m p n<m

env-subst : ∀ ξ₁ ξ₂ φ → (∀ n → ξ₁ n ≡ ξ₂ n) → ⟦ ξ₁ ⊨ φ ⟧pl → ⟦ ξ₂ ⊨ φ ⟧pl
env-subst ξ₁ ξ₂ ¥true    ext = id
env-subst ξ₁ ξ₂ ¥false   ext = id
env-subst ξ₁ ξ₂ (φ || ψ) ext = Sum.map (env-subst ξ₁ ξ₂ φ ext) (env-subst ξ₁ ξ₂ ψ ext)
env-subst ξ₁ ξ₂ (φ && ψ) ext = Prod.map (env-subst ξ₁ ξ₂ φ ext) (env-subst ξ₁ ξ₂ ψ ext)
env-subst ξ₁ ξ₂ (φ => ψ) ext = λ x → env-subst ξ₁ ξ₂ ψ ext ∘ x ∘ env-subst ξ₂ ξ₁ φ (sym ∘ ext)
env-subst ξ₁ ξ₂ (¥ v)    ext rewrite ext v = id

env-eq-guard : ∀ ξ₁ ξ₂ φ → (∀ n → T (¥ n isSubFormula φ) → ξ₁ n ≡ ξ₂ n)
             → ⟦ ξ₁ ⊨ φ ⟧pl ≡ ⟦ ξ₂ ⊨ φ ⟧pl
env-eq-guard ξ₁ ξ₂ ¥true ext  = refl
env-eq-guard ξ₁ ξ₂ ¥false ext = refl
env-eq-guard ξ₁ ξ₂ (φ || ψ) ext
  = cong₂ _⊎_ (env-eq-guard ξ₁ ξ₂ φ (λ n x → ext n (V-introl _ _ x)))
              (env-eq-guard ξ₁ ξ₂ ψ (λ n x → ext n (V-intror (¥ n isSubFormula φ) _ x)))
env-eq-guard ξ₁ ξ₂ (φ && ψ) ext
  = cong₂ _×_ (env-eq-guard ξ₁ ξ₂ φ (λ n x → ext n (V-introl _ _ x)))
              (env-eq-guard ξ₁ ξ₂ ψ (λ n x → ext n (V-intror (¥ n isSubFormula φ) _ x)))
env-eq-guard ξ₁ ξ₂ (φ => ψ) ext
  = cong₂ (\ a b → a → b) (env-eq-guard ξ₁ ξ₂ φ (λ n x → ext n (V-introl _ _ x)))
                          (env-eq-guard ξ₁ ξ₂ ψ (λ n x → ext n (V-intror (¥ n isSubFormula φ) _ x)))
env-eq-guard ξ₁ ξ₂ (¥ v) ext rewrite ext v (id-isSubFormula (¥ v)) = refl

env-subst-guard : ∀ ξ₁ ξ₂ φ → (∀ n → T (¥ n isSubFormula φ) → ξ₁ n ≡ ξ₂ n)
                → ⟦ ξ₁ ⊨ φ ⟧pl → ⟦ ξ₂ ⊨ φ ⟧pl
```

```
env-subst-guard ξ₁ ξ₂ ¥true ext = id
env-subst-guard ξ₁ ξ₂ ¥false ext = id
env-subst-guard ξ₁ ξ₂ (φ || ψ) ext
  = Sum.map (env-subst-guard ξ₁ ξ₂ φ (λ n x → ext n (V-introl _ _ x)))
            (env-subst-guard ξ₁ ξ₂ ψ (λ n x → ext n (V-intror (¥ n isSubFormula φ) _ x)))
env-subst-guard ξ₁ ξ₂ (φ && ψ) ext
  = Prod.map (env-subst-guard ξ₁ ξ₂ φ (λ n x → ext n (V-introl _ _ x)))
             (env-subst-guard ξ₁ ξ₂ ψ (λ n x → ext n (V-intror (¥ n isSubFormula φ) _ x)))
env-subst-guard ξ₁ ξ₂ (φ => ψ) ext
  = λ x → env-subst-guard ξ₁ ξ₂ ψ (λ n x' → ext n (V-intror (¥ n isSubFormula φ) _ x')) ∘
          x ∘ env-subst-guard ξ₂ ξ₁ φ (λ n x' → sym (ext n (V-introl _ _ x')))
env-subst-guard ξ₁ ξ₂ (¥ v) ext rewrite ext v (id-isSubFormula (¥ v)) = id

subform-¥-elim : ∀ k y → T (¥ (suc k) isSubFormula ¥ (suc y)) → T (¥ k isSubFormula ¥ y)
subform-¥-elim k y p with k == y
...| true = tt
...| false = p

lem-mkenv-++-pl-eq : ∀ φ ξ ζ → T (bound (length ξ) φ) → ∀ k → T (¥ k isSubFormula φ)
                   → mkenv ξ k ≡ mkenv (ξ ++ ζ) k
lem-mkenv-++-pl-eq ¥true ξ ζ b k kinφ = ⊥-elim kinφ
lem-mkenv-++-pl-eq ¥false ξ ζ b k kinφ = ⊥-elim kinφ
lem-mkenv-++-pl-eq (y || y') ξ ζ b k kinφ
  = V-elim (lem-mkenv-++-pl-eq y ξ ζ (∧-eliml b) k)
           (lem-mkenv-++-pl-eq y' ξ ζ (∧-elimr (bound (length ξ) y) b) k) kinφ
lem-mkenv-++-pl-eq (y && y') ξ ζ b k kinφ
  = V-elim (lem-mkenv-++-pl-eq y ξ ζ (∧-eliml b) k)
           (lem-mkenv-++-pl-eq y' ξ ζ (∧-elimr (bound (length ξ) y) b) k) kinφ
lem-mkenv-++-pl-eq (y => y') ξ ζ b k kinφ
  = V-elim (lem-mkenv-++-pl-eq y ξ ζ (∧-eliml b) k)
           (lem-mkenv-++-pl-eq y' ξ ζ (∧-elimr (bound (length ξ) y) b) k) kinφ
lem-mkenv-++-pl-eq (¥ y) [] ζ b k kinφ = ⊥-elim b
lem-mkenv-++-pl-eq (¥ y) (x :: ξ) ζ b zero kinφ = refl
lem-mkenv-++-pl-eq (¥ zero) (x :: ξ) ζ b (suc k) kinφ = ⊥-elim kinφ
lem-mkenv-++-pl-eq (¥ (suc y)) (x :: ξ) ζ b (suc k) kinφ
  = lem-mkenv-++-pl-eq (¥ y) ξ ζ b k (subform-¥-elim k y kinφ)

lem-mkenv-++-pl-eq' : ∀ φ ξ ξ' ζ → ξ' ≡ ξ ++ ζ → T (bound (length ξ) φ) → ∀ k
                    → T (¥ k isSubFormula φ) → mkenv ξ k ≡ mkenv ξ' k
lem-mkenv-++-pl-eq' ¥true ξ ξ' ζ eq b k kinφ = ⊥-elim kinφ
lem-mkenv-++-pl-eq' ¥false ξ ξ' ζ eq b k kinφ = ⊥-elim kinφ
lem-mkenv-++-pl-eq' (y || y') ξ ξ' ζ ξ≡ξ' b k kinφ
  = V-elim (lem-mkenv-++-pl-eq' y ξ ξ' ζ ξ≡ξ' (∧-eliml b) k)
           (lem-mkenv-++-pl-eq' y' ξ ξ' ζ ξ≡ξ' (∧-elimr (bound (length ξ) y) b) k) kinφ
lem-mkenv-++-pl-eq' (y && y') ξ ξ' ζ ξ≡ξ' b k kinφ
  = V-elim (lem-mkenv-++-pl-eq' y ξ ξ' ζ ξ≡ξ' (∧-eliml b) k)
           (lem-mkenv-++-pl-eq' y' ξ ξ' ζ ξ≡ξ' (∧-elimr (bound (length ξ) y) b) k) kinφ
lem-mkenv-++-pl-eq' (y => y') ξ ξ' ζ ξ≡ξ' b k kinφ
  = V-elim (lem-mkenv-++-pl-eq' y ξ ξ' ζ ξ≡ξ' (∧-eliml b) k)
           (lem-mkenv-++-pl-eq' y' ξ ξ' ζ ξ≡ξ' (∧-elimr (bound (length ξ) y) b) k) kinφ
lem-mkenv-++-pl-eq' (¥ y) [] ξ' ζ ξ≡ξ' b k kinφ = ⊥-elim b
lem-mkenv-++-pl-eq' (¥ y) (x :: ξ) ._ ζ refl b zero kinφ = refl
lem-mkenv-++-pl-eq' (¥ zero) (x :: ξ) ξ' ζ ξ≡ξ' b (suc k) kinφ = ⊥-elim kinφ
lem-mkenv-++-pl-eq' (¥ (suc y)) (x :: ξ) ._ ζ refl b (suc k) kinφ
  = lem-mkenv-++-pl-eq' (¥ y) ξ _ ζ refl b k (subform-¥-elim k y kinφ)

lem-mkenv-++-pl : ∀ φ ξ ζ → T (bound (length ξ) φ) → ⟦ mkenv ξ ⊨ φ ⟧pl → ⟦ mkenv (ξ ++ ζ) ⊨ φ ⟧pl
lem-mkenv-++-pl φ ξ ζ b = env-subst-guard (mkenv ξ) (mkenv (ξ ++ ζ)) φ (lem-mkenv-++-pl-eq φ ξ ζ b)

lem-mkenv-++-pl' : ∀ φ ξ ζ → T (bound (length ξ) φ) → ⟦ mkenv (ξ ++ ζ) ⊨ φ ⟧pl → ⟦ mkenv ξ ⊨ φ ⟧pl
lem-mkenv-++-pl' φ ξ ζ b = env-subst-guard (mkenv (ξ ++ ζ)) (mkenv ξ) φ
                                           (λ k p → sym (lem-mkenv-++-pl-eq φ ξ ζ b k p))

lem-length : {a : Set} → (l m : List a) → length l + length m ≡ length (l ++ m)
lem-length [] m = refl
lem-length (x :: l) m = cong suc (lem-length l m)

lem-length² : {a : Set} → (l m n : List a) → length l + length m + length n ≡ length (l ++ m ++ n)
lem-length² [] m n = lem-length m n
lem-length² (x :: l) m n = cong suc (lem-length² l m n)

lem-mkenv-++-pl-eq² : ∀ φ ξ₁ ξ₂ ζ → T (bound (length ξ₁ + length ξ₂) φ)→ _
lem-mkenv-++-pl-eq² φ ξ₁ ξ₂ ζ rewrite lem-length ξ₁ ξ₂
  = lem-mkenv-++-pl-eq' φ (ξ₁ ++ ξ₂) (ξ₁ ++ ξ₂ ++ ζ) ζ (sym (Monoid.assoc (monoid Bool) ξ₁ ξ₂ ζ))

lem-mkenv-++-pl² : ∀ φ ξ₁ ξ₂ ζ → T (bound (length ξ₁ + length ξ₂) φ)
                 → ⟦ mkenv (ξ₁ ++ ξ₂) ⊨ φ ⟧pl → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ζ) ⊨ φ ⟧pl
lem-mkenv-++-pl² φ ξ₁ ξ₂ ζ b
  = env-subst-guard (mkenv (ξ₁ ++ ξ₂)) (mkenv (ξ₁ ++ ξ₂ ++ ζ)) φ (lem-mkenv-++-pl-eq² φ ξ₁ ξ₂ ζ b)

lem-mkenv-++-pl²' : ∀ φ ξ₁ ξ₂ ζ → T (bound (length ξ₁ + length ξ₂) φ)
                  → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ζ) ⊨ φ ⟧pl → ⟦ mkenv (ξ₁ ++ ξ₂) ⊨ φ ⟧pl
lem-mkenv-++-pl²' φ ξ₁ ξ₂ ζ b
  = env-subst-guard (mkenv (ξ₁ ++ ξ₂ ++ ζ)) (mkenv (ξ₁ ++ ξ₂)) φ
                    (\ k p → sym (lem-mkenv-++-pl-eq² φ ξ₁ ξ₂ ζ b k p))

lem-mkenv-++-pl-eq³ : ∀ φ ξ₁ ξ₂ ξ₃ ζ
```

```
                    → T (bound (length ξ₁ + length ξ₂ + length ξ₃) φ)
                    → (n : ℕ) → _ → mkenv (ξ₁ ++ ξ₂ ++ ξ₃) n ≡ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ζ) n
lem-mkenv-++-pl-eq³ φ ξ₁ ξ₂ ξ₃ ζ b rewrite lem-length ξ₁ ξ₂
                                         | sym (Monoid.assoc (monoid Bool)  ξ₁ ξ₂ ξ₃)
                                         | sym (Monoid.assoc (monoid Bool)  ξ₁ ξ₂ (ξ₃ ++ ζ))
 = lem-mkenv-++-pl-eq² φ (ξ₁ ++ ξ₂) ξ₃ ζ b

lem-mkenv-++-pl³ : ∀ φ ξ₁ ξ₂ ξ₃ ζ → T (bound (length ξ₁ + length ξ₂ + length ξ₃) φ)
                 → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃) ⊨ φ ⟧pl → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ζ) ⊨ φ ⟧pl
lem-mkenv-++-pl³ φ ξ₁ ξ₂ ξ₃ ζ p
  = env-subst-guard (mkenv (ξ₁ ++ ξ₂ ++ ξ₃)) (mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ζ)) φ
                    (lem-mkenv-++-pl-eq³ φ ξ₁ ξ₂ ξ₃ ζ p)

lem-mkenv-++-pl³' : ∀ φ ξ₁ ξ₂ ξ₃ ζ → T (bound (length ξ₁ + length ξ₂ + length ξ₃) φ)
                  → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ζ) ⊨ φ ⟧pl → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃) ⊨ φ ⟧pl
lem-mkenv-++-pl³' φ ξ₁ ξ₂ ξ₃ ζ p
  = env-subst-guard (mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ζ)) (mkenv (ξ₁ ++ ξ₂ ++ ξ₃)) φ
                    (\ a b → sym (lem-mkenv-++-pl-eq³ φ ξ₁ ξ₂ ξ₃ ζ p a b))

lem-mkenv-++-pl-eq⁴ : ∀ φ ξ₁ ξ₂ ξ₃ ξ₄ ζ
                    → T (bound (length ξ₁ + length ξ₂ + length ξ₃ + length ξ₄) φ)
                    → ∀ n → _
                    → mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄) n ≡ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄ ++ ζ) n
lem-mkenv-++-pl-eq⁴ φ ξ₁ ξ₂ ξ₃ ξ₄ ζ b rewrite lem-length ξ₁ ξ₂
                                           | sym (Monoid.assoc (monoid Bool) ξ₁ ξ₂ (ξ₃ ++ ξ₄))
                                           | sym (Monoid.assoc (monoid Bool) ξ₁ ξ₂ (ξ₃ ++ ξ₄ ++ ζ))
 = lem-mkenv-++-pl-eq³ φ (ξ₁ ++ ξ₂) ξ₃ ξ₄ ζ b

lem-mkenv-++-pl⁴ : ∀ φ ξ₁ ξ₂ ξ₃ ξ₄ ζ → T (bound (length ξ₁ + length ξ₂ + length ξ₃ + length ξ₄) φ)
                 → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄) ⊨ φ ⟧pl
                 → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄ ++ ζ) ⊨ φ ⟧pl
lem-mkenv-++-pl⁴ φ ξ₁ ξ₂ ξ₃ ξ₄ ζ p
  = env-subst-guard (mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄)) (mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄ ++ ζ)) φ
                    (lem-mkenv-++-pl-eq⁴ φ ξ₁ ξ₂ ξ₃ ξ₄ ζ p)

lem-mkenv-++-pl⁴' : ∀ φ ξ₁ ξ₂ ξ₃ ξ₄ ζ
                  → T (bound (length ξ₁ + length ξ₂ + length ξ₃ + length ξ₄) φ)
                  → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄ ++ ζ) ⊨ φ ⟧pl
                  → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄) ⊨ φ ⟧pl
lem-mkenv-++-pl⁴' φ ξ₁ ξ₂ ξ₃ ξ₄ ζ p
  = env-subst-guard (mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄ ++ ζ)) (mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄)) φ
                    (\ a b → sym (lem-mkenv-++-pl-eq⁴ φ ξ₁ ξ₂ ξ₃ ξ₄ ζ p a b))

extendenv : ∀ ξ ξ' ζ → (P : ℕ → Set) → length ξ ≡ length ξ' → (∀ k → P k → mkenv ξ k ≡ mkenv ξ' k)
          → ∀ j → P j → mkenv (ξ ++ ζ) j ≡ mkenv (ξ' ++ ζ) j
extendenv [] [] ζ P lp ∀k j pj = refl
extendenv [] (x' :: ξ') ζ P () ∀k j pj
extendenv (x :: ξ) [] ζ P () ∀k j pj
extendenv (x :: ξ) (x' :: ξ') ζ P lp ∀k zero pj = ∀k 0 pj
extendenv (x :: ξ) (x' :: ξ') ζ P lp ∀k (suc n) pj = extendenv ξ ξ' ζ (P ∘ suc)
                                                      (cong pred lp) (∀k ∘ suc) n pj

shiftpl : PL-Formula → ℕ → PL-Formula
shiftpl φ n = elim-pl ¥true ¥false (¥ ∘ (_+_ n)) _||_ _&&_ _=>_ φ

mutual
  lem-shift1-pl : ∀ m b φ ξ → ⟦ mkenv ξ ⊨ shiftpl φ m ⟧pl → ⟦ mkenv (b :: ξ) ⊨ shiftpl φ (suc m) ⟧pl
  lem-shift1-pl m b ¥true ξ     = id
  lem-shift1-pl m b ¥false ξ    = id
  lem-shift1-pl m b (y || y') ξ = Sum.map  (lem-shift1-pl m b y ξ) (lem-shift1-pl m b y' ξ)
  lem-shift1-pl m b (y && y') ξ = Prod.map (lem-shift1-pl m b y ξ) (lem-shift1-pl m b y' ξ)
  lem-shift1-pl m b (y => y') ξ = \ x → (lem-shift1-pl m b y' ξ) ∘ x ∘ (lem-shift1-pl' m b y ξ)
  lem-shift1-pl m b (¥ y) ξ     = id

  lem-shift1-pl' : ∀ m b φ ξ → ⟦ mkenv (b :: ξ) ⊨ shiftpl φ (suc m) ⟧pl → ⟦ mkenv ξ ⊨ shiftpl φ m ⟧pl
  lem-shift1-pl' m b ¥true ξ     = id
  lem-shift1-pl' m b ¥false ξ    = id
  lem-shift1-pl' m b (y || y') ξ = Sum.map  (lem-shift1-pl' m b y ξ) (lem-shift1-pl' m b y' ξ)
  lem-shift1-pl' m b (y && y') ξ = Prod.map (lem-shift1-pl' m b y ξ) (lem-shift1-pl' m b y' ξ)
  lem-shift1-pl' m b (y => y') ξ = \ x → (lem-shift1-pl' m b y' ξ) ∘ x ∘ (lem-shift1-pl m b y ξ)
  lem-shift1-pl' m b (¥ y) ξ     = id

id-elim-pl : ∀ φ → elim-pl ¥true ¥false ¥ _||_ _&&_ _=>_ φ ≡ φ
id-elim-pl ¥true     = refl
id-elim-pl ¥false    = refl
id-elim-pl (y || y') = cong₂ _||_ (id-elim-pl y) (id-elim-pl y')
id-elim-pl (y && y') = cong₂ _&&_ (id-elim-pl y) (id-elim-pl y')
id-elim-pl (y => y') = cong₂ _=>_ (id-elim-pl y) (id-elim-pl y')
id-elim-pl (¥ y)     = refl

lem-shift-pl : (φ : PL-Formula) → ∀ ξ ζ → ⟦ mkenv ζ ⊨ φ ⟧pl
             → ⟦ mkenv (ξ ++ ζ) ⊨ shiftpl φ (length ξ) ⟧pl
lem-shift-pl φ [] ζ p rewrite id-elim-pl φ = p
lem-shift-pl φ (x :: ξ) ζ p = lem-shift1-pl (length ξ) x φ (ξ ++ ζ) (lem-shift-pl φ ξ ζ p)

lem-shift-pl' : (φ : PL-Formula) → ∀ ξ ζ → ⟦ mkenv (ξ ++ ζ) ⊨ shiftpl φ (length ξ) ⟧pl
              → ⟦ mkenv ζ ⊨ φ ⟧pl
lem-shift-pl' φ [] ζ p rewrite (id-elim-pl φ) = p
```

```
lem-shift-pl' φ (x :: ξ) ζ p = lem-shift-pl' φ ξ ζ (lem-shift1-pl' (length ξ) x φ (ξ ++ ζ) p)

lem-shift-pl² : (φ : PL-Formula) → (ξ₁ ξ₂ ξ₃ : List Bool) → ⟦ mkenv ξ₃ ⊨ φ ⟧pl
              → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃) ⊨ shiftpl φ (length ξ₁ + length ξ₂) ⟧pl
lem-shift-pl² φ [] ξ₂ ξ₃ p = lem-shift-pl φ ξ₂ ξ₃ p
lem-shift-pl² φ (x :: ξ₁) ξ₂ ξ₃ p = lem-shift1-pl (length ξ₁ + length ξ₂) x φ (ξ₁ ++ ξ₂ ++ ξ₃)
                                                  (lem-shift-pl² φ ξ₁ ξ₂ ξ₃ p)

lem-shift-pl²' : (φ : PL-Formula) → (ξ₁ ξ₂ ξ₃ : List Bool)
               → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃) ⊨ shiftpl φ (length ξ₁ + length ξ₂) ⟧pl
               → ⟦ mkenv ξ₃ ⊨ φ ⟧pl
lem-shift-pl²' φ [] ξ₂ ξ₃ p = lem-shift-pl' φ ξ₂ ξ₃ p
lem-shift-pl²' φ (x :: ξ₁) ξ₂ ξ₃ p
  = lem-shift-pl²' φ ξ₁ ξ₂ ξ₃ (lem-shift1-pl' (length ξ₁ + length ξ₂) x φ (ξ₁ ++ ξ₂ ++ ξ₃) p)

lem-shift-pl³ : (φ : PL-Formula)
              → (ξ₁ ξ₂ ξ₃ ξ₄ : List Bool)
              → ⟦ mkenv ξ₄ ⊨ φ ⟧pl
              → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄) ⊨ shiftpl φ (length ξ₁ + length ξ₂ + length ξ₃) ⟧pl
lem-shift-pl³ φ [] ξ₂ ξ₃ ξ₄ p = lem-shift-pl² φ ξ₂ ξ₃ ξ₄ p
lem-shift-pl³ φ (x :: ξ₁) ξ₂ ξ₃ ξ₄ p = lem-shift1-pl (length ξ₁ + length ξ₂ + length ξ₃)
                                                     x φ (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄)
                                                     (lem-shift-pl³ φ ξ₁ ξ₂ ξ₃ ξ₄ p)

lem-shift-pl³' : (φ : PL-Formula) → (ξ₁ ξ₂ ξ₃ ξ₄ : List Bool)
               → ⟦ mkenv (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄) ⊨ shiftpl φ (length ξ₁ + length ξ₂ + length ξ₃)⟧pl
               → ⟦ mkenv ξ₄ ⊨ φ ⟧pl
lem-shift-pl³' φ [] ξ₂ ξ₃ ξ₄ p = lem-shift-pl²' φ ξ₂ ξ₃ ξ₄ p
lem-shift-pl³' φ (x :: ξ₁) ξ₂ ξ₃ ξ₄ p
  = lem-shift-pl³' φ ξ₁ ξ₂ ξ₃ ξ₄ (lem-shift1-pl' (length ξ₁ + length ξ₂ + length ξ₃)
                                                 x φ (ξ₁ ++ ξ₂ ++ ξ₃ ++ ξ₄) p)

lem-bound : ∀ φ n k → T (bound n φ) → T (¥ k isSubFormula φ) → T (k < n)
lem-bound ¥true     n k bn sub = ⊥-elim sub
lem-bound ¥false    n k bn sub = ⊥-elim sub
lem-bound (y || y') n k bn sub = ∨-elim (lem-bound y n k (∧-eliml bn))
                                        (lem-bound y' n k (∧-elimr (bound n y) bn)) sub
lem-bound (y && y') n k bn sub = ∨-elim (lem-bound y n k (∧-eliml bn))
                                        (lem-bound y' n k (∧-elimr (bound n y) bn)) sub
lem-bound (y => y') n k bn sub = ∨-elim (lem-bound y n k (∧-eliml bn))
                                        (lem-bound y' n k (∧-elimr (bound n y) bn)) sub
lem-bound (¥ y)     n k bn sub with ex-mid (k == y)
...| inj₁ x rewrite lift-== k y x = bn
...| inj₂ x rewrite ¬Tb x = ⊥-elim sub

-- another variant of subst
env-eq-bound : ∀ ξ₁ ξ₂ φ n → T (bound n φ) → (∀ m → T (m < n) → ξ₁ m ≡ ξ₂ m)
             → ⟦ ξ₁ ⊨ φ ⟧pl ≡ ⟦ ξ₂ ⊨ φ ⟧pl
env-eq-bound ξ₁ ξ₂ ¥true     n p q = refl
env-eq-bound ξ₁ ξ₂ ¥false    n p q = refl
env-eq-bound ξ₁ ξ₂ (φ || φ₁) n p q = cong₂ _∪_ (env-eq-bound ξ₁ ξ₂ φ n (∧-eliml p) q)
                                               (env-eq-bound ξ₁ ξ₂ φ₁ n (∧-elimr (bound n φ) p) q)
env-eq-bound ξ₁ ξ₂ (φ && φ₁) n p q = cong₂ _×_ (env-eq-bound ξ₁ ξ₂ φ n (∧-eliml p) q)
                                               (env-eq-bound ξ₁ ξ₂ φ₁ n (∧-elimr (bound n φ) p) q)
env-eq-bound ξ₁ ξ₂ (φ => φ₁) n p q = cong₂ (λ a b → a → b)
                                           (env-eq-bound ξ₁ ξ₂ φ n (∧-eliml p) q)
                                           (env-eq-bound ξ₁ ξ₂ φ₁ n (∧-elimr (bound n φ) p) q)
env-eq-bound ξ₁ ξ₂ (¥ x)     n p q = cong T (q x p)

env-eq-bound-subst : ∀ ξ₁ ξ₂ φ n → T (bound n φ) → (∀ m → T (m < n) → ξ₁ m ≡ ξ₂ m)
                   → ⟦ ξ₁ ⊨ φ ⟧pl → ⟦ ξ₂ ⊨ φ ⟧pl
env-eq-bound-subst ξ₁ ξ₂ φ n p q r rewrite env-eq-bound ξ₁ ξ₂ φ n p q = r

injbool : Bool → PL-Formula
injbool true = ¥true
injbool false = ¥false

andpl : List PL-Formula → PL-Formula
andpl = foldr _&&_ ¥true
```

```
module Boolean.TPTP where

open import Data.Bool
open import Data.Nat
open import Data.String

open import Boolean.Formula

private primitive primShowNat : ℕ → String

tptpformat : PL-Formula → String
tptpformat ¥true    = "$true"
tptpformat ¥false   = "$false"
tptpformat (¥ vid)  = "'" ++ (primShowNat vid)  ++ "'"
tptpformat (φ && ψ) = "(" ++ tptpformat φ ++ " & "  ++ tptpformat ψ ++ ")"
tptpformat (φ || ψ) = "(" ++ tptpformat φ ++ " | "  ++ tptpformat ψ ++ ")"
tptpformat (φ => ψ) = "(" ++ tptpformat φ ++ " => " ++ tptpformat ψ ++ ")"

tptp : PL-Formula → String
tptp φ = "fof(ax1,axiom," ++ tptpformat (~ φ) ++ ")."
```

```
module Boolean.CommonBinding where

open import Boolean.Formula
open import Boolean.TPTP

open import Data.Bool

open import PropIso

{-# BUILTIN UNIT ⊤ #-}

{-# BUILTIN TRIV tt #-}

{-# BUILTIN EMPTY ⊥ #-}

{-# BUILTIN ATOM T #-}

{-# BUILTIN ATPPROBLEM PL-Formula #-}

{-# BUILTIN ATPINPUT tptp #-}
```

```
module Boolean.SatSolver where

open import Data.Bool renaming (_∧_ to _∧ᵇ_; _∨_ to _∨ᵇ_; not to ¬ᵇ_)
open import Data.Nat hiding (_<_ ; _≤_)
open import Data.Product as Prod
open import Data.Sum as Sum

open import PropIso renaming (_→_ to _⇒ᵇ_)
open import Boolean.Formula

varbound : PL-Formula → ℕ
varbound ¥true     = 0
varbound ¥false    = 0
varbound (y || y') = max (varbound y) (varbound y')
varbound (y && y') = max (varbound y) (varbound y')
varbound (y => y') = max (varbound y) (varbound y')
varbound (¥ y)     = suc y

lem-varbound : ∀ φ → T (bound (varbound φ) φ)
lem-varbound ¥true  = tt
lem-varbound ¥false = tt
lem-varbound (y || y')
  = elim-max _ _ (λ k → T (bound k (y || y')))
      (λ p → ∧-intro _ _ (lem-varbound y) (injbound y' (varbound y') _ (<-¬ _ (varbound y') p)
                                                      (lem-varbound y')))
      (λ p → ∧-intro _ _ (injbound y (varbound y) _ (<-rsuc (varbound y) _ p) (lem-varbound y))
                         (lem-varbound y'))
lem-varbound (y && y')
  = elim-max _ _ (λ k → T (bound k (y && y')))
      (λ p → ∧-intro _ _ (lem-varbound y) (injbound y' (varbound y') _
                                                      (<-¬ _ (varbound y') p) (lem-varbound y')))
      (λ p → ∧-intro _ _ (injbound y (varbound y) _ (<-rsuc (varbound y) _ p) (lem-varbound y))
                         (lem-varbound y'))
lem-varbound (y => y')
  = elim-max _ _ (λ k → T (bound k (y => y')))
      (λ p → ∧-intro _ _ (lem-varbound y) (injbound y' (varbound y') _
                                                      (<-¬ _ (varbound y') p) (lem-varbound y')))
      (λ p → ∧-intro _ _ (injbound y (varbound y) _ (<-rsuc (varbound y) _ p) (lem-varbound y))
                         (lem-varbound y'))
lem-varbound (¥ y) = <-ord y

Boolean : ℕ → Set -- formulae bounded by n variables
Boolean n = Σ PL-Formula (T ∘ bound n)

⟦_⊨_⟧b : ∀ {n} → Env → Boolean n → Set
⟦ ξ ⊨ φ ⟧b = ⟦ ξ ⊨ proj₁ φ ⟧pl

BooleanFormula : Set
BooleanFormula = Σ ℕ Boolean

mkbooleanformula : PL-Formula → BooleanFormula
mkbooleanformula φ = varbound φ , φ , lem-varbound φ

Taut : BooleanFormula → Set
Taut φ = ∀ ξ → ⟦ ξ ⊨ proj₂ φ ⟧b

inst : ∀ {n} → Boolean n → Boolean (suc n)→ Boolean n
inst b (¥true   , proj₂) = ¥true , proj₂
inst b (¥false  , proj₂) = ¥false , proj₂
inst b (y || y' , proj₂)
  = uncurry' (λ x y0 → uncurry' (λ x' y1 → x || x' , ∧-intro (bound _ x) _ y0 y1)
                                (inst b (y' , ∧-elimr (bound (suc _) y) proj₂)))
             (inst b (y , ∧-eliml proj₂))
inst b (y && y' , proj₂)
  =  uncurry' (λ x y0 → uncurry' (λ x' y1 → x && x' , ∧-intro (bound _ x) _ y0 y1)
                                 (inst b (y' , ∧-elimr (bound (suc _) y) proj₂)))
              (inst b (y , ∧-eliml proj₂))
inst b (y => y' , proj₂)
  =  uncurry' (λ x y0 → uncurry' (λ x' y1 → x => x' , ∧-intro (bound _ x) _ y0 y1)
                                 (inst b (y' , ∧-elimr (bound (suc _) y) proj₂)))
              (inst b (y , ∧-eliml proj₂))
inst b (¥ zero     , proj₂) = b
inst b (¥ (suc n') , proj₂) = (¥ n') , proj₂

abstract
  taut : BooleanFormula → Bool
  taut (zero  , ¥true   , p) = true
  taut (zero  , ¥false  , p) = false
  taut (zero  , y || y' , p) = taut (0 , y , ∧-eliml p) ∨ᵇ taut (0 , y' , ∧-elimr (bound 0 y) p)
  taut (zero  , y && y' , p) = taut (0 , y , ∧-eliml p) ∧ᵇ taut (0 , y' , ∧-elimr (bound 0 y) p)
  taut (zero  , y => y' , p) = taut (0 , y , ∧-eliml p) ⇒ᵇ taut (0 , y' , ∧-elimr (bound 0 y) p)
  taut (zero  , ¥ y     , ())
  taut (suc n , φ       , p)
    = uncurry' (λ x y0 → uncurry' (λ x' y1 → taut (n , x , y0) ∧ᵇ taut (n , x' , y1))
                                  (inst (¥true , tt) (φ , p)))
               (inst (¥false , tt) (φ , p))
```

```
const-pl : Bool → ∀ n → Boolean n
const-pl true n  = ¥true , tt
const-pl false n = ¥false , tt

mutual
  lem-inst : ∀ {n : ℕ} (φ : Boolean (suc n)) (ξ : Env) → ⟦ ξ ⊨ φ ⟧b
           → ⟦ ξ ∘ suc ⊨ inst (const-pl (ξ 0) n) φ ⟧b
  lem-inst (¥true   , b) ξ p = p
  lem-inst (¥false  , b) ξ p = p
  lem-inst (y || y' , b) ξ p = Sum.map  (lem-inst (y , ∧-eliml b) ξ)
                                        (lem-inst (y' , ∧-elimr (bound _ y) b) ξ) p
  lem-inst (y && y' , b) ξ p = Prod.map (lem-inst (y , ∧-eliml b) ξ)
                                        (lem-inst (y' , ∧-elimr (bound _ y) b) ξ) p
  lem-inst (y => y' , b) ξ p = lem-inst (y' , ∧-elimr (bound _ y) b) ξ ∘ p ∘
                               lem-inst' (y , ∧-eliml b) ξ
  lem-inst (¥ zero  , b) ξ p with ξ 0
  ...| true = p
  ...| false = p
  lem-inst (¥ (suc n') , b) ξ p = p

  lem-inst' : ∀ {n : ℕ} (φ : Boolean (suc n)) (ξ : Env) → ⟦ ξ ∘ suc ⊨ inst (const-pl (ξ 0) n) φ ⟧b
            → ⟦ ξ ⊨ φ ⟧b
  lem-inst' (¥true   , b) ξ p = p
  lem-inst' (¥false  , b) ξ p = p
  lem-inst' (y || y' , b) ξ p = Sum.map  (lem-inst' (y , ∧-eliml b) ξ)
                                         (lem-inst' (y' , ∧-elimr (bound _ y) b) ξ) p
  lem-inst' (y && y' , b) ξ p = Prod.map (lem-inst' (y , ∧-eliml b) ξ)
                                         (lem-inst' (y' , ∧-elimr (bound _ y) b) ξ) p
  lem-inst' (y => y' , b) ξ p = lem-inst' (y' , ∧-elimr (bound _ y) b) ξ ∘ p ∘
                                lem-inst (y , ∧-eliml b) ξ
  lem-inst' (¥ zero  , b) ξ p with ξ 0
  ...| true = p
  ...| false = p
  lem-inst' (¥ (suc n') , b) ξ p = p

lem-ξ-zero : (φ : Boolean 0) → (ξ ζ : Env) → ⟦ ξ ⊨ φ ⟧b → ⟦ ζ ⊨ φ ⟧b
lem-ξ-zero (¥true   , b) ξ ζ p = p
lem-ξ-zero (¥false  , b) ξ ζ p = p
lem-ξ-zero (y || y' , b) ξ ζ p = Sum.map  (lem-ξ-zero (y , ∧-eliml b) ξ ζ)
                                          (lem-ξ-zero (y' , ∧-elimr (bound _ y) b) ξ ζ) p
lem-ξ-zero (y && y' , b) ξ ζ p = Prod.map (lem-ξ-zero (y , ∧-eliml b) ξ ζ)
                                          (lem-ξ-zero (y' , ∧-elimr (bound _ y) b) ξ ζ) p
lem-ξ-zero (y => y' , b) ξ ζ p = lem-ξ-zero (y' , ∧-elimr (bound _ y) b) ξ ζ ∘ p ∘
                                 lem-ξ-zero (y , ∧-eliml b) ζ ξ
lem-ξ-zero (¥ y     , b) ξ ζ p = ⊥-elim b

private
  trivenv : Env
  trivenv _ = true

mutual
  abstract
    sound : (φ : BooleanFormula) → T (taut φ) → Taut φ
    sound (zero , ¥true , b) p ξ   = p
    sound (zero , ¥false , b) p ξ  = p
    sound (zero , y || y' , b) p ξ = ∨-elim (λ x → inj₁ (sound (0 , y , _) x ξ))
                                            (λ x → inj₂ (sound (0 , y' , _) x ξ)) p
    sound (zero , y && y' , b) p ξ = ∧-elim (λ x x' → (sound (0 , y , _) x ξ) ,
                                                      (sound (0 , y' , _) x' ξ)) p
    sound (zero , y => y' , b) p ξ = λ x → sound (0 , y' , _) (lem→→-s (taut (0 , y , _)) _ p
                                                 (comp (0 , y , _)
                                                   (λ ξ' → lem-ξ-zero (y , ∧-eliml b) ξ ξ' x))) ξ
    sound (zero , ¥ y , b) p ξ     = ⊥-elim b
    sound (suc n , φ , b) p ξ      = lem-inst' (φ , b) ξ (sound (n , inst (const-pl (ξ 0) n)(φ , b))
                                                          (aux p) (λ _ → ξ(suc _)))
      where
        aux : T (taut (n , inst (¥false , tt) (φ , b)) ∧ᵇ taut (n , inst (¥true , tt) (φ , b)))
            → T (taut (n , inst (const-pl (ξ 0) n) (φ , b)))
        aux p with ξ 0
        ...| true  = ∧-elimr (taut (n , _ , _)) p
        ...| false = ∧-eliml p

    comp : (φ : BooleanFormula) → Taut φ → T (taut φ)
    comp (zero , ¥true   , b) p = tt
    comp (zero , ¥false  , b) p = p trivenv
    comp (zero , y || y' , b) p
      = lem-bool-∨-c (taut (0 , y , _)) _
                     (Sum.map (λ x → comp (0 , y , _)
                                          (λ ξ → lem-ξ-zero (y , ∧-eliml b) trivenv ξ x))
                              (λ x → comp (0 , y' , _)
                                          (λ ξ → lem-ξ-zero (y' , ∧-elimr(bound 0 y)b) trivenv ξ x))
                              (p trivenv))
    comp (zero , y && y' , b) p = ∧-intro (taut (0 , y , _)) _
                                          (comp (0 , y , _) (λ ξ → proj₁ (p ξ)))
                                          (comp (0 , y' , _) (λ ξ → proj₂ (p ξ)))
    comp (zero , y => y' , b) p = lem→→-c (taut (0 , y , _)) _
                                          (λ x → comp (0 , y' , _)
                                                 (λ ξ → p ξ (sound (0 , y , _) x ξ)))
```

```
    comp (zero , ¥ y , b) p      = ⊥-elim b
    comp (suc n , φ , b) p
      = ∧-intro (taut (n , inst (¥false , tt) (φ , b))) _
                (comp (n , inst (¥false , tt) (φ , b))
                      (λ ξ → lem-inst (φ , b) (extendξ ξ false) $ p $ extendξ ξ false))
                (comp (n , inst (¥true , tt) (φ , b))
                      (λ ξ → lem-inst (φ , b) (extendξ ξ true) $ p $ extendξ ξ true))
      where
       extendξ : Env → Bool → Env
       extendξ ξ b zero     = b
       extendξ ξ b (suc x) = ξ x

{- External Interface -}
open import Boolean.CommonBinding
open import Data.String

atptool : String
atptool = "z3"

{-# BUILTIN ATPTOOL atptool #-}

decproc : PL-Formula → Bool
decproc = taut ∘ mkbooleanformula

{-# BUILTIN ATPDECPROC decproc #-}

{-# BUILTIN ATPSEMANTICS Taut-pl #-}

sound' : (φ : PL-Formula) → T (decproc φ) → Taut-pl φ
sound' = sound ∘ mkbooleanformula

{-# BUILTIN ATPSOUND sound' #-}

comp' :  (φ : PL-Formula) → Taut-pl φ → T (decproc φ)
comp' = comp ∘ mkbooleanformula

{-# BUILTIN ATPCOMPLETE comp' #-}
```

```
module Data.Fin.Arithmetic where

open import Data.Fin hiding (_<_;inject≤;_+_;inject+;inject₁)
open import Data.Nat hiding (_<_)
open import Data.Bool
open import Data.Product as Prod

open import Relation.Binary.PropositionalEquality

open import PropIso

finpred : ∀ {n} → Fin (suc (suc n)) → Fin (suc n)
finpred zero    = zero
finpred (suc i) = i

finpred' : ∀ {n} → Fin n → Fin n
finpred' {zero}        x             = x
finpred' {suc zero}    x             = zero
finpred' {suc (suc n)} zero          = zero -- for loop put finmax here
finpred' {suc (suc n)} (suc zero)    = zero
finpred' {suc (suc n)} (suc (suc i)) = suc (finpred' (suc i))

finsuc : ∀ {n} → Fin n → Fin n
finsuc {zero}        ()
finsuc {suc zero}    zero    = zero
finsuc {suc (suc n)} zero    = suc zero
finsuc {suc n}       (suc i) = suc (finsuc i)

finprop-aux : ∀ {n} → (Fin (suc n) → Bool) → Bool
finprop-aux {zero}  f = f zero
finprop-aux {suc n} f = f zero ∨ finprop-aux {n} (f ∘ suc)

finprop : ∀ {n} → (Fin n → Bool) → Bool
finprop {zero}  f = false
finprop {suc n} f = finprop-aux f

finpred-eq : {n : ℕ} → (i i' : Fin n) → _≡_ {A = Fin (suc n)} (suc i) (suc i') → i ≡ i'
finpred-eq _ ._ refl = refl

flt : {n m : ℕ} → (x : Fin n) → (y : Fin m) → Bool
flt x y = toℕ x < toℕ y

lem-flt : {n : ℕ} → (x : Fin n) → T (toℕ x < n)
lem-flt zero    = tt
lem-flt (suc i) = lem-flt i

inject≤ : {m n : ℕ} → T (m < (suc n)) → Fin m → Fin n
inject≤ {zero}  p       ()
inject≤ {suc m} {zero}  () i
inject≤ {suc m} {suc n} p  zero    = zero
inject≤ {suc m} {suc n} p  (suc i) = suc (inject≤ p i)

fin-remainder : {n : ℕ} → Fin (suc n) → ℕ
fin-remainder {n}     zero    = n
fin-remainder {zero}  (suc ())
fin-remainder {suc n} (suc i) = fin-remainder i

lem-fin-remainder-lt : {n : ℕ} → (a : Fin n) → T (fin-remainder (suc a) < n)
lem-fin-remainder-lt (zero {n}) = <-ord n
lem-fin-remainder-lt (suc a)    = <-rsuc (fin-remainder (suc a)) _ (lem-fin-remainder-lt a)

lem-fin-remainder-plus-lt' : {n : ℕ} (x : Fin (suc n)) (y : ℕ)
                           → (y<lfmx : T (y < (suc (fin-remainder x))))
                           → T ((toℕ x + suc y) < suc (suc n))
lem-fin-remainder-plus-lt'          zero    y p = p
lem-fin-remainder-plus-lt' {zero}   (suc ()) y p
lem-fin-remainder-plus-lt' {suc n} (suc x)  y p = lem-fin-remainder-plus-lt' x y p

lem-fin-remainder-plus-lt : ∀ {n} x → T ((toℕ x + suc (fin-remainder x)) < suc (suc n))
lem-fin-remainder-plus-lt {n} x
  = lem-fin-remainder-plus-lt' x (fin-remainder x) (<-ord (fin-remainder x))

-- (a - b) - 1
fminus : {n : ℕ} → (a b : Fin (suc n)) → T (flt b a) → Fin (fin-remainder b {- n - b -})
fminus zero          b       ()
fminus (suc a)       zero    p  = a
fminus (suc zero)    (suc b) ()
fminus (suc (suc a)) (suc b) p  = fminus (suc a) b p

<-fin-remainder : {m n : ℕ} → (k : Fin m) → (k' : Fin (suc n)) → T (flt k k')
                  → T ((toℕ k + fin-remainder k') < n)
<-fin-remainder                 zero    zero     ()
<-fin-remainder                 zero    (suc k') k<k' = lem-fin-remainder-lt k'
<-fin-remainder {suc m} {zero}  (suc k) zero     ()
<-fin-remainder {suc m} {zero}  (suc k) (suc ()) k<k'
<-fin-remainder {suc m} {suc n} (suc k) zero     ()
<-fin-remainder {suc m} {suc n} (suc k) (suc k') k<k' = <-fin-remainder k k' k<k'
```

```
maxfin : ∀{n} → (Fin n → ℕ) → ℕ
maxfin {zero}  f = 0
maxfin {suc n} f = max (f zero) (maxfin (λ x → f (suc x)))

_suc^_ : {n : ℕ} → Fin n → (m : ℕ) → Fin (m + n)
x suc^ zero  = x
x suc^ suc n = suc (x suc^ n)

fromℕ< : ∀ {n} m → T (m < n) → Fin n
fromℕ< {zero}  _       ()
fromℕ< {suc n} zero    p  = zero
fromℕ< {suc n} (suc m) p  = suc (fromℕ< m p)

toℕ< : ∀ {n} → Fin n → Σ ℕ (\ k → T (k < n))
toℕ< zero    = zero , tt
toℕ< (suc i) = Prod.map suc id (toℕ< i)

lem-tofromℕ : ∀ {n} m (x : T (m < n)) → toℕ< {n} (fromℕ< m x) ≡ (m , x)
lem-tofromℕ {zero}  _       ()
lem-tofromℕ {suc n} zero    tt = refl
lem-tofromℕ {suc n} (suc m) x
  = cong (Prod.map suc id) (subst (λ k → k ≡ (m , x)) refl (lem-tofromℕ {n} m x))

lem-fromtoℕ : {n : ℕ} → (m : Fin n) → fromℕ< {n} (proj₁ (toℕ< m)) (proj₂ (toℕ< m)) ≡ m
lem-fromtoℕ {zero}  ()
lem-fromtoℕ {suc n} zero    = refl
lem-fromtoℕ {suc n} (suc i) = cong suc (lem-fromtoℕ i)

lem-tofromtoℕ : {n m : ℕ} (x : Fin n) (y : T (toℕ x < m)) → toℕ x ≡ toℕ {m} (fromℕ< (toℕ x) y)
lem-tofromtoℕ {zero}  ()       y
lem-tofromtoℕ {suc n} {zero}   zero    ()
lem-tofromtoℕ {suc n} {suc m}  zero    y  = refl
lem-tofromtoℕ {suc n} {zero}   (suc x) ()
lem-tofromtoℕ {suc n} {suc n'} (suc x) y  = cong suc (lem-tofromtoℕ x y)

-- maybe remove this as its only an application of cong'
lem-fromtofromtoℕ : {n m : ℕ} → (i : Fin n) → (x : T (toℕ i < n)) → (z : T (toℕ i < m))
                  → fromℕ< {n} (toℕ i) x ≡ fromℕ< {n} (toℕ (fromℕ< {m} (toℕ i) z))
                                          (subst (λ k → T (k < n)) (lem-tofromtoℕ i z) x)
lem-fromtofromtoℕ {n} i x z = cong' (λ k → T (k < n)) (fromℕ< {n}) (lem-tofromtoℕ i z) x

finminus : {n m : ℕ} → (x : Fin (n + m)) → ¬ T (toℕ x < n) → Fin m
finminus {zero}   x       ¬x<n = x
finminus {suc n'} zero    ¬x<n = ⊥-elim (¬x<n tt)
finminus {suc n}  (suc x) ¬x<n = finminus {n} x ¬x<n

lem-finminus-id : ∀ {n} m (x : Fin n) (z : ¬ T (toℕ (x suc^ m) < m)) → x ≡ finminus {m} (x suc^ m) z
lem-finminus-id zero    x z = refl
lem-finminus-id (suc m) x z = lem-finminus-id m x z

lem-finminus-id' : ∀ {n} m (x : Fin (m + n)) (z : ¬ T (toℕ x < m)) → x ≡ (finminus {m} x z) suc^ m
lem-finminus-id' zero    x       z = refl
lem-finminus-id' (suc m) zero    z = ⊥-elim (z tt)
lem-finminus-id' (suc m) (suc i) z = cong suc (lem-finminus-id' m i z)

inject+' : {n : ℕ} → (m o : ℕ) → o ≡ m + n → Fin n → Fin o
inject+' {n} zero .n refl x = x
inject+' (suc m) .(suc (m + suc n)) refl (zero {n}) = zero
inject+' {suc n} (suc m) .(suc (m + suc n)) refl (suc i)
  = suc (inject+' {n} (suc m) (m + suc n) (trans (+-comm m (suc n)) (cong suc (+-comm n m))) (i))

inject+ : {n : ℕ} → (m : ℕ) → Fin n → Fin (m + n)
inject+ {n} m = inject+' m (m + n) refl

inject₁ : ∀ {m} → Fin m → Fin (suc m)
inject₁ {m} = inject+' 1 (suc m) refl

toℕ-inj-eq : ∀ {n} (x : Fin n) i i' ieq j j' jeq
           → toℕ (inject+' i i' ieq x) ≡ toℕ (inject+' j j' jeq x)
toℕ-inj-eq {n} x zero .n refl zero .n refl = refl
toℕ-inj-eq (zero {n}) zero .(suc n) refl (suc j) .(suc (j + suc n)) refl = refl
toℕ-inj-eq (suc {n} x) zero .(suc n) refl (suc j) .(suc (j + suc n)) refl
  = cong suc (toℕ-inj-eq x 0 n refl (suc j) (j + suc n)
                         (trans (+-comm j (suc n)) (cong suc (+-comm n j))))
toℕ-inj-eq (zero {n}) (suc i) .(suc (i + suc n)) refl zero .(suc n) refl = refl
toℕ-inj-eq (zero {n}) (suc i) .(suc (i + suc n)) refl (suc j) .(suc (j + suc n)) refl = refl
toℕ-inj-eq (suc {n} x) (suc i) .(suc (i + suc n)) refl zero .(suc n) refl
  = cong suc (toℕ-inj-eq x (suc i) (i + suc n)
                         (trans (+-comm i (suc n)) (cong suc (+-comm n i))) 0 n refl)
toℕ-inj-eq (suc {n} x) (suc i) .(suc (i + suc n)) refl (suc j) .(suc (j + suc n)) refl
  = cong suc (toℕ-inj-eq x (suc i) (i + suc n)
                         (trans (+-comm i (suc n)) (cong suc (+-comm n i))) (suc j)
                         (j + suc n) (trans (+-comm j (suc n)) (cong suc (+-comm n j))))

lem-fin-suc^ : {n : ℕ} → (m : ℕ) → (x : Fin n) → ¬ T (toℕ (x suc^ m) < m)
lem-fin-suc^ zero    x = id
lem-fin-suc^ (suc m) x = lem-fin-suc^ m x
```

```
lem-fromto𝕟-inj+' : ∀ n k o (eq : o ≡ k + n) (i : Fin o) (x : T (to𝕟 i < n))
                  → inject+' k o eq (from𝕟< {n} (to𝕟 i) x) ≡ i
lem-fromto𝕟-inj+' zero    k       ._ refl i       ()
lem-fromto𝕟-inj+' (suc n) zero    ._ refl zero    x  = refl
lem-fromto𝕟-inj+' (suc n) zero    ._ refl (suc i) x  = cong suc (lem-fromto𝕟-inj+' n 0 _ refl i x)
lem-fromto𝕟-inj+' (suc n) (suc k) ._ refl zero    x  = refl
lem-fromto𝕟-inj+' (suc n) (suc k) ._ refl (suc i) x
  = cong suc (lem-fromto𝕟-inj+' n (suc k) _ (trans (+-comm k (suc n)) (cong suc (+-comm n k))) i x)

lem-fromto𝕟-inj+ : ∀ n k (i : Fin (k + n))(x : T (to𝕟 i < n)) → inject+ k (from𝕟< {n} (to𝕟 i) x) ≡ i
lem-fromto𝕟-inj+ n k = lem-fromto𝕟-inj+' n k _ refl
```

```
module CTL.TransitionSystem where

open import Data.Nat hiding (_<_)
open import Data.Fin hiding (inject≤;_<_) renaming (_+_ to _F+_)
open import Data.Fin.Arithmetic hiding (inject₁)
open import Data.Fin.Pigeon
open import Data.List
open import Data.Bool
open import Data.Product as Prod
open import Data.Sum as Sum

open import Relation.Binary.PropositionalEquality

open import Coinduction

open import PropIso

-- finite state machine
record FSM : Set where
  constructor
    fsm
  field
    state atom : ℕ
    action : Fin state → ℕ
    initial : List (Fin state)
    transition : (s : Fin state) → (a : Fin (action s)) → Fin state
    label : Fin state → Fin atom → Bool

  State : Set
  State = Fin state

  Action : State → Set
  Action s = Fin (action s)

open FSM

data Transition (ts : FSM) (a : State ts) : State ts → Set where
  arrow : (su : Action ts a) → (b : State ts) → b ≡ (transition ts a su) → Transition ts a b

data Run (ts : FSM) (s : State ts) : Set where
  next : (s' : Action ts s) → ∞ (Run ts (transition ts s s')) → Run ts s

run-decomp : ∀ ts {s} → Run ts s → Σ[ x ∶ Action ts s ] (Run ts (transition ts s x))
run-decomp ts {s} (next s' y) = s' , ♭ y

run-head : (ts : FSM) → {s : State ts} → Run ts s → Fin (action ts s)
run-head ts r = proj₁ (run-decomp ts r)

run-tail : ∀ ts {s} → (r : Run ts s) → Run ts (transition ts s (run-head ts r))
run-tail ts r = proj₂ (run-decomp ts r)

nth : (ts : FSM) → {s : State ts} → (n : ℕ) → Run ts s → State ts
nth ts {s} zero r = s
nth ts (suc n) r  = nth ts n (run-tail ts r)

run-drop : (ts : FSM) → {s : Fin (state ts)} → (n : ℕ) → (r : Run ts s) → Run ts (nth ts n r)
run-drop ts zero r    = r
run-drop ts (suc n) r = run-drop ts n (run-tail ts r)

data FinRun (ts : FSM) (s : State ts) : ℕ → Set where
  end : FinRun ts s 0
  next : {n : ℕ} → (s' : Action ts s) → FinRun ts (transition ts s s') n → FinRun ts s (suc n)

last : (ts : FSM) → {n : ℕ} → {s : State ts} → FinRun ts s n → State ts
last ts {zero} {s} end          = s
last ts {suc n} {s} (next s' y) = last ts y

fnth : ∀ ts {n} {s} → FinRun ts s n → Fin (suc n) → State ts
fnth ts {_} {s} r zero = s
fnth ts end (suc ())
fnth ts {(suc n)} {s} (next s' y) (suc x) = fnth ts y x

fnth-suc : ∀ ts {n} {s} → (r : FinRun ts s n)  → (x : Fin n) → Action ts (fnth ts r (inject₁ x))
fnth-suc ts {zero} _ ()
fnth-suc ts {suc n} {s} (next s' y) zero    = s'
fnth-suc ts {suc n} {s} (next s' y) (suc x) = fnth-suc ts y x

lem-fnth-suc : ∀ ts {n} {s} → (r : FinRun ts s n) → (x : Fin n)
             → fnth ts r (suc x) ≡ transition ts (fnth ts r (inject₁ x)) (fnth-suc ts r x)
lem-fnth-suc ts {zero} r ()
lem-fnth-suc ts {suc n} {s} (next s' y) zero    = refl
lem-fnth-suc ts {suc n} {s} (next s' y) (suc x) = lem-fnth-suc ts y x

frconc : ∀ ts {s n m} → (r : FinRun ts s n) → (r' : FinRun ts (last ts r) m) → FinRun ts s (n + m)
frconc ts end r'             = r'
frconc ts {s} (next s' y) r' = next s' (frconc ts y r')
```

```
lem-frconc-last : ∀ ts {s n m} → (r : FinRun ts s n) → (r' : FinRun ts (last ts r) m)
                → last ts r' ≡ last ts (frconc ts r r')
lem-frconc-last ts end r' = refl
lem-frconc-last ts {s} (next s' y) r' = lem-frconc-last ts y r'

frtake : ∀ ts {n s} → (r : FinRun ts s n) → (x : Fin (suc n)) → FinRun ts s (toℕ x)
frtake ts end zero                          = end
frtake ts end (suc ())
frtake ts {.(suc n)} {s} (next {n} s' y) zero    = end
frtake ts {.(suc n)} {s} (next {n} s' y) (suc x) = next s' (frtake ts y x)

lem-frtake : ∀ ts {n s} → (r : FinRun ts s n) → (x : Fin (suc n))
           → fnth ts r x ≡ last ts (frtake ts r x)
lem-frtake ts end zero                          = refl
lem-frtake ts end (suc ())
lem-frtake ts {.(suc n)} {s} (next {n} s' y) zero    = refl
lem-frtake ts {.(suc n)} {s} (next {n} s' y) (suc x) = lem-frtake ts y x

lem-frtake-lift : ∀ ts {n s} → (r : FinRun ts s n) → (l : Fin (suc n)) → (x : Fin (suc (toℕ l)))
                → fnth ts r (inject≤ {suc _} (lem-flt l) x) ≡ fnth ts (frtake ts r l) x
lem-frtake-lift ts r zero zero                          = refl
lem-frtake-lift ts r zero (suc ())
lem-frtake-lift ts r (suc l) zero                       = refl
lem-frtake-lift ts end (suc ()) (suc x)
lem-frtake-lift ts {.(suc n)} {s} (next {n} s' y) (suc l) (suc x) = lem-frtake-lift ts y l x

frdrop : ∀ ts {n s} → (r : FinRun ts s n) → (x : Fin (suc n))
       → FinRun ts (fnth ts r x) (fin-remainder x)
frdrop ts r zero                          = r
frdrop ts end (suc ())
frdrop ts {.(suc n)} {s} (next {n} s' y) (suc x) = frdrop ts y x

lem-frdrop-last : ∀ ts {n s} → (r : FinRun ts s n) → (x : Fin (suc n))
                → last ts r ≡ last ts (frdrop ts r x)
lem-frdrop-last ts r zero = refl
lem-frdrop-last ts end (suc ())
lem-frdrop-last ts {.(suc n)} {s} (next {n} s' y) (suc x) = lem-frdrop-last ts y x

lem-frdrop-lift : ∀ ts {n s} → (r : FinRun ts s n) → (k : Fin (suc n))
                → (x : Fin (suc (fin-remainder k)))
                → fnth ts r (inject≤ (lem-fin-remainder-plus-lt k) (k F+ x))
                  ≡ fnth ts (frdrop ts r k) x
lem-frdrop-lift ts r zero zero                          = refl
lem-frdrop-lift ts end zero (suc ())
lem-frdrop-lift ts {.(suc n)} {s} (next {n} s' y) zero (suc zero) = refl
lem-frdrop-lift ts {.(suc n)} {s} (next {n} s' y) zero (suc (suc x))
  = lem-frdrop-lift ts y zero (suc x)
lem-frdrop-lift ts end (suc ()) x
lem-frdrop-lift ts {.(suc n)} {s} (next {n} s' y) (suc k) x       = lem-frdrop-lift ts y k x

-- ignores last element in run
lem-frdrop-lift-nolast : ∀ ts {n s} → (r : FinRun ts s n) → (k : Fin (suc n))
                       → (x : Fin (fin-remainder k))
                       → fnth ts r (inject₁ (inject≤ (<-+-rsuc (toℕ k) _ (suc n)
                                                     (lem-fin-remainder-plus-lt k)) (k F+ x)))
                         ≡ fnth ts (frdrop ts r k) (inject₁ x)
lem-frdrop-lift-nolast ts r zero zero = refl
lem-frdrop-lift-nolast ts {.(suc n)} {s} (next {n} s' y) zero (suc x)
  = lem-frdrop-lift-nolast ts y zero x
lem-frdrop-lift-nolast ts end (suc ()) x
lem-frdrop-lift-nolast ts {.(suc n)} {s} (next {n} s' y) (suc k) x
  = lem-frdrop-lift-nolast ts y k x

lem-frtake-frdrop : ∀ ts {n s} → (r : FinRun ts s n) → (k : Fin (suc n))→ (l : Fin n)
                  → (k<l : T (flt k (suc l)))
                  → fnth ts r (inject₁ l)
                    ≡ last ts (frtake ts (frdrop ts r k) (inject₁ (fminus (suc l) k k<l)))
lem-frtake-frdrop ts r zero zero k<l    = lem-frtake ts r (inject₁ zero)
lem-frtake-frdrop ts r zero (suc l) k<l = lem-frtake ts r (suc (inject₁ l))
lem-frtake-frdrop ts r (suc k) zero ()
lem-frtake-frdrop ts {.(suc n)} {s} (next {n} s' y) (suc k) (suc l) k<l
  = lem-frtake-frdrop ts y k l k<l

lem-frtake-frdrop-lift : ∀ ts {n s} → (r : FinRun ts s n) → (k : Fin (suc n))
                       → (l : Fin (suc (fin-remainder k))) → (x : Fin (suc (toℕ l)))
                       → fnth ts r (inject≤ (lem-fin-remainder-plus-lt' k (toℕ l)
                                            (lem-flt l)) (k F+ x))
                       ≡ fnth ts (frtake ts (frdrop ts r k) l) x
lem-frtake-frdrop-lift ts r zero l x                          = lem-frtake-lift ts r l x
lem-frtake-frdrop-lift ts end (suc ()) l x
lem-frtake-frdrop-lift ts {._} {s} (next {n} s' y) (suc k) l x = lem-frtake-frdrop-lift ts y k l x

data Lasso (ts : FSM) : State ts → Set where
  lasso : {s' : State ts}  -- initial state
        → (i j : ℕ)  -- prefix / loop length-1
        → (s : State ts)  -- loop state
        → (prefix : FinRun ts s i)
        → (con : Transition ts (last ts prefix) s')
```

```
        → (loop : FinRun ts s' j)
        → (loopproof : Transition ts (last ts loop) s')
        → Lasso ts s

getPrefixLen : (ts : FSM)
             → {s : State ts}
             → (l : Lasso ts s)
             → ℕ
getPrefixLen ts {s} (lasso i j .s prefix con loop loopproof) = i

getLoopLen : (ts : FSM)
           → {s : State ts}
           → (l : Lasso ts s)
           → ℕ
getLoopLen ts {s} (lasso i j .s prefix con loop loopproof) = j

getPrefix : (ts : FSM)
          → {s : State ts}
          → (l : Lasso ts s)
          → FinRun ts s (getPrefixLen ts l)
getPrefix ts {s} (lasso i j .s prefix con loop loopproof) = prefix

getLoopStart : (ts : FSM)
             → {s : State ts}
             → (l : Lasso ts s)
             → State ts
getLoopStart ts {s} (lasso {s'} i j .s prefix con loop loopproof) = s'

getLoop : (ts : FSM)
        → {s : State ts}
        → (l : Lasso ts s)
        → FinRun ts (getLoopStart ts l) (getLoopLen ts l)
getLoop ts {s} (lasso i j .s prefix con loop loopproof) = loop

-- canonical translation from lasso into run
lasso2run : (ts : FSM) → {s : State ts} → Lasso ts s → Run ts s
lasso2run ts {s} (lasso zero j .s end (arrow suc' ._ refl) loop loopproof)
  = next suc' (♯ lasso2run ts (lasso j j (transition ts s suc') loop loopproof loop loopproof))
lasso2run ts {s} (lasso (suc i) j .s (next suc' y) con loop loopproof)
  = next suc' (♯ lasso2run ts (lasso i j (transition ts s suc') y con loop loopproof))

-- maps a point from the resulting infinite run back to the prefix/loop
lem-lasso2run : (ts : FSM) → {s : State ts} → (l : Lasso ts s) → (n : ℕ)
              → (Σ[ pre ∶ Fin (suc (getPrefixLen ts l)) ]
                    (fnth ts (getPrefix ts l) pre ≡ nth ts n (lasso2run ts l)))
                ⊎ (Σ[ loop ∶ Fin (suc (getLoopLen ts l)) ]
                      (fnth ts (getLoop ts l) loop ≡ nth ts n (lasso2run ts l)))
lem-lasso2run ts {s} (lasso i j .s prefix con loop loopproof) zero = inj₁ (zero , refl)
lem-lasso2run ts {s} (lasso .0 j .s end (arrow suc' ._ refl) loop loopproof) (suc n)
  = inj₂ ([ id , id ]' (lem-lasso2run ts (lasso j j _ loop loopproof loop loopproof) n))
lem-lasso2run ts {s} (lasso .(suc n) j .s (next {n} s0 y) con loop loopproof) (suc n')
  = Sum.map (Prod.map suc id) id (lem-lasso2run ts (lasso n j _ y con loop loopproof) n')

{-
From this point on, functions/lemmas specific to CTL logic and should be relocated
-}

-- Globally

path2lasso-G : (ts : FSM) → {s : State ts} → FinRun ts s (state ts) → Lasso ts s
path2lasso-G ts r with pigeon (suc (state ts)) (state ts) (<-ord (state ts)) (fnth ts r)
path2lasso-G ts r | (k , zero) , () , eq
path2lasso-G ts {s} r | (zero , suc l) , k<l , eq = lasso (toℕ (inject₁ l))
                                                          (toℕ (inject₁ l))
                                                          s (frtake ts r (inject₁ l))
                                                          p (frtake ts r (inject₁ l)) p
  where
    p : Transition ts (last ts (frtake ts r (inject₁ l))) s
    p = subst (λ x → Transition ts (last ts (frtake ts r (inject₁ l))) x)
              (sym eq) (subst (λ x → Transition ts x (fnth ts r (suc l)))
                              (lem-frtake ts r (inject₁ l))
                              (arrow (fnth-suc ts r l) (fnth ts r (suc l)) (lem-fnth-suc ts r l)))
path2lasso-G ts {s} r | (suc k , suc l) , k<l , eq
  = lasso (toℕ (inject₁ k)) (toℕ (inject₁ (fminus (suc l) (suc k) k<l))) s (frtake ts r (inject₁ k))
          (subst (λ x → Transition ts x (fnth ts r (suc k))) (lem-frtake ts r (inject₁ k))
                 (arrow (fnth-suc ts r k) (fnth ts r (suc k)) (lem-fnth-suc ts r k)))
          (frtake ts (frdrop ts r (suc k)) (inject₁ (fminus (suc l) (suc k) k<l)))
          (subst (λ x → Transition ts x (fnth ts r (suc k))) (lem-frtake-frdrop ts r (suc k) l k<l)
                 (arrow (fnth-suc ts r l) (fnth ts r (suc k))
                        (subst (λ x → x ≡ transition ts (fnth ts r (inject₁ l)) (fnth-suc ts r l))
                               (sym eq) (lem-fnth-suc ts r l))))

lem-path2lasso-G : ∀ ts {s} → (r : FinRun ts s (state ts)) → (p : State ts → Bool)
                 → (q : (x : Fin (suc (state ts))) → T (p (fnth ts r x))) → (x : ℕ)
                 → T (p (nth ts x (lasso2run ts (path2lasso-G ts r))))
lem-path2lasso-G ts r p q x = [ (λ k → subst (T ∘ p) (proj₂ k) (proj₁ (f r p q) (proj₁ k))) ,
                                (λ k → subst (T ∘ p) (proj₂ k) (proj₂ (f r p q) (proj₁ k)))
                              ]' (lem-lasso2run ts (path2lasso-G ts r) x)
```

```
  where
    f : {s : State ts}
      → (r : FinRun ts s (state ts))
      → (p : State ts → Bool)
      → (q : (x : Fin (suc (state ts))) → T (p (fnth ts r x)))
      → (∀ x → T (p (fnth ts (getPrefix ts (path2lasso-G ts r)) x)))
        × (∀ x → T (p (fnth ts (getLoop ts (path2lasso-G ts r)) x)))
    f r p q with pigeon (suc (state ts)) (state ts) (<-ord (state ts)) (fnth ts r)
    f r' p' q' | (k , zero) , () , eq
    f r' p' q' | (zero , suc l) , k<l , eq = xx , xx
      where
        xx : (x' : Fin (suc (toℕ (inject₁ l)))) → T (p' (fnth ts (frtake ts r' (inject₁ l)) x'))
        xx x = subst (T ∘ p') (lem-frtake-lift ts r' (inject₁ l) x) (q' (inject≤ _ x))
    f r' p' q' | (suc k , suc l) , k<l , eq
      = (λ x' → subst (T ∘ p') (lem-frtake-lift ts r' (inject₁ k) x')
                      (q' (inject≤ {suc (toℕ (inject₁ k))} (lem-flt (inject₁ k)) x')))
        , λ x' → subst (T ∘ p') (lem-frtake-frdrop-lift ts r' (suc k)
                                                        (inject₁ (fminus (suc l) (suc k) k<l)) x')
                        (q' (inject≤ _ (suc (k F+ x'))))
```

```
module CTL.Definition where

open import CTL.TransitionSystem

open import Data.Fin hiding (_<_)
open import Data.Sum
open import Data.Product
open import Data.Bool using (T)
open import Data.Nat hiding (_<_)

open import PropIso

{- CTL formula -}
data CTL (n : ℕ) : Set where
  false : CTL n
  _∨_ _∧_ E[_U_] : (φ : CTL n) → (ψ : CTL n) → CTL n
  P[_] : (ap : Fin n) → CTL n
  ~ EX EG : (φ : CTL n) → CTL n

data CTLProblem : Set where
  _,_⊨_ : (M : FSM) → (Fin (FSM.state M)) → CTL (FSM.atom M) → CTLProblem

⟦_⟧ : CTLProblem → Set
⟦ M , s ⊨ false ⟧       = ⊥
⟦ M , s ⊨ ~ φ ⟧         = ¬ ⟦ M , s ⊨ φ ⟧
⟦ M , s ⊨ (φ ∨ ψ) ⟧     = ⟦ M , s ⊨ φ ⟧ ⊎ ⟦ M , s ⊨ ψ ⟧
⟦ M , s ⊨ (φ ∧ ψ) ⟧     = ⟦ M , s ⊨ φ ⟧ × ⟦ M , s ⊨ ψ ⟧
⟦ M , s ⊨ P[ ap ] ⟧     = T (FSM.label M s ap)
⟦ M , s ⊨ EX φ ⟧        = Σ[ run ∶ Run M s ] ⟦ M , nth M 1 run ⊨ φ ⟧
⟦ M , s ⊨ EG φ ⟧        = Σ[ run ∶ Run M s ] (∀ (n : ℕ) → ⟦ M , nth M n run ⊨ φ ⟧)
⟦ M , s ⊨ E[ φ U ψ ] ⟧ = Σ[ run ∶ Run M s ] Σ[ k ∶ ℕ ]
                           ((∀ (j : ℕ) → T (j < k) → ⟦ M , nth M j run ⊨ φ ⟧)
                             × ⟦ M , nth M k run ⊨ ψ ⟧)

--{-# BUILTIN ATPPROBLEM CTLProblem #-}
--{-# BUILTIN ATPSEMANTICS ⟦_⟧ #-}
```

```
{-# OPTIONS --universe-polymorphism #-}

module Data.Fin.EqReasoning where

open import Data.Fin
open import Data.Nat
open import Data.Bool
open import Data.Product as Prod
open import Data.Sum as Sum

open import Relation.Binary.PropositionalEquality

open import PropIso

fin-∨ : (n : ℕ) → Fin n → (Fin n → Bool) → Bool
fin-∨ zero () f
fin-∨ (suc zero)        zero     f = f zero ∨ false
fin-∨ (suc (suc n))     (zero)   f = f zero ∨ fin-∨ (suc n) zero (λ k → f (suc k))
fin-∨ (suc zero)        (suc ()) f
fin-∨ (suc (suc n))     (suc i)  f =  fin-∨ (suc n) i (λ k → f (suc k))

fin-∨-↔ : (n : ℕ) → (f : Fin n → Bool) → Bool
fin-∨-↔ zero f = false
fin-∨-↔ (suc n) f = fin-∨ (suc n) zero f

lem-fin-∨ : {n : ℕ} → (i : Fin n) → (f : Fin n → Bool) → T (f i) → T (fin-∨-↔ n f)
lem-fin-∨ {zero} () f p
lem-fin-∨ {suc zero} zero f p = ∨-introl (f zero) false p
lem-fin-∨ {suc (suc n)} zero f p = ∨-introl (f zero) (fin-∨ ((suc n)) zero (λ k → f (suc k))) p
lem-fin-∨ {suc zero} (suc ()) f p
lem-fin-∨ {suc (suc n)} (suc i) f p = ∨-intror (f zero) (fin-∨ ((suc n)) zero (λ k → f (suc k)))
                                               (lem-fin-∨ i (λ x → f (suc x)) p)

lem-fin-∨' : {n : ℕ} → (f : Fin n → Bool) → T (fin-∨-↔ n f) → Σ (Fin n) (λ i → T (f i))
lem-fin-∨' {zero} f ()
lem-fin-∨' {suc zero} f p = zero , [ id , ⊥-elim ]' (lem-bool-∨-s (f zero) false p)
lem-fin-∨' {suc (suc n)} f p = [ (λ x → zero , x) ,
                                 (λ x → Prod.map suc id (lem-fin-∨' {suc n} (λ k → f (suc k)) x))
                               ]' (lem-bool-∨-s (f zero) (fin-∨ ((suc n)) zero (λ k → f (suc k))) p)

toBool : Fin 2 → Bool
toBool zero = false
toBool (suc zero) = true
toBool (suc (suc ()))

fromBool : Bool → Fin 2
fromBool false = zero
fromBool true = suc zero

tobool-iso1 : ∀ b → toBool (fromBool b) ≡ b
tobool-iso1 true = refl
tobool-iso1 false = refl

tobool-iso2 : ∀ b → fromBool (toBool b) ≡ b
tobool-iso2 zero = refl
tobool-iso2 (suc zero) = refl
tobool-iso2 (suc (suc ()))

flip-toBool1 : ∀ b c → c ≡ toBool b → fromBool c ≡ b
flip-toBool1 zero .false refl = refl
flip-toBool1 (suc zero) .true refl = refl
flip-toBool1 (suc (suc ())) ._ refl

flip-toBool2 : ∀ b c → fromBool b ≡ c → b ≡ toBool c
flip-toBool2 true .(suc zero) refl = refl
flip-toBool2 false .zero refl = refl

_∧f_ : Fin 2 → Fin 2 → Fin 2
x ∧f y = fromBool (toBool x ∧ toBool y)

_∨f_ : Fin 2 → Fin 2 → Fin 2
x ∨f y = fromBool (toBool x ∨ toBool y)

¬f : Fin 2 → Fin 2
¬f = fromBool ∘ not ∘ toBool

elim-fin0 : ∀ {n} {A : Set n} → Fin 0 → A
elim-fin0 ()
```

```
module CTL.ListGen where

open import Data.Nat
open import Data.Fin
open import Data.List hiding (_++_)
open import Data.String

[_⋯_] : ℕ → ℕ → List ℕ
[ n ⋯ zero ] = [ n ]
[ n ⋯ (suc m) ] = n ∷ [ suc n ⋯ m ]



private
  primitive
    primShowNat : ℕ → String

natlist : ℕ → String
natlist 0 = ""
natlist 1 = "0"
natlist (suc n) = natlist n ++ "," ++ primShowNat n

natlist-1 : ℕ → String
natlist-1 0 = "-1"
natlist-1 1 = "-1,0"
natlist-1 (suc n) = natlist-1 n ++ "," ++ primShowNat n

∀fin : ∀{n} → (Fin n → String) → String → String
∀fin {zero} f d = ""
∀fin {suc zero} f d = f zero
∀fin {suc n} f d = f zero ++ d ++ ∀fin (λ x → f (suc x)) d
```

```
module CTL.DecProc where

open import Data.Fin hiding (_<_)
open import Data.Fin.EqReasoning
open import Data.Fin.Arithmetic
open import Data.Nat hiding (_<_)
open import Data.Bool renaming (_∧_ to _∧ᵇ_ ; _∨_ to _∨ᵇ_)
open import Data.String
open import Data.Product as Prod
open import Data.Sum

open import PropIso

open import CTL.TransitionSystem
open import CTL.Definition
open import CTL.Sink
open import CTL.ListGen

{-
Decision procedure for CTL logic
-}
open FSM

-- checks that a property holds for a path of given length
check : (M : FSM) → ℕ → (p : State M → Bool) → State M → Bool
check M zero p s    = p s
check M (suc n) p s = p s ∧ᵇ fin-∨-↔ (action M s) (check M n p ∘ transition M s)

-- checks if some property holds in i'th state on infinite path
checkX : (M : FSM) → (i : ℕ) → (p : State M → Bool) → State M → Bool
checkX M zero p s    = p s ∧ᵇ check M (state M) (const true) s
checkX M (suc i) p s = fin-∨-↔ (action M s) (checkX M i p ∘ transition M s)

-- slightly different version, the number determines the "prefix" length
-- then the above check is called to determine if we are one an infinite
-- path.
checkU : (M : FSM) → ℕ → (p : State M → Bool) → (q : State M → Bool) → State M → Bool
checkU M zero p q s    = q s ∧ᵇ check M (state M) (const true) s
checkU M (suc n) p q s = (p s ∧ᵇ fin-∨-↔ (action M s) (checkU M n p q ∘ transition M s))
                       ∨ᵇ (q s ∧ᵇ check M (state M) (const true) s)

eval : CTLProblem → Bool
eval (M , s ⊧ false)      = false
eval (M , s ⊧ ~ φ)        = not (eval (M , s ⊧ φ))
eval (M , s ⊧ (φ ∨ ψ))    = (eval (M , s ⊧ φ)) ∨ᵇ (eval (M , s ⊧ ψ))
eval (M , s ⊧ (φ ∧ ψ))    = (eval (M , s ⊧ φ)) ∧ᵇ (eval (M , s ⊧ ψ))
eval (M , s ⊧ P[ ap ])    = label M s ap
eval (M , s ⊧ EX φ)       = checkX M 1 (\ s' → eval (M , s' ⊧ φ)) s
eval (M , s ⊧ EG φ)       = check M (state M) (λ s' → eval (M , s' ⊧ φ)) s
eval (M , s ⊧ E[ φ U ψ ]) = checkU M (state M) (λ t → eval (M , t ⊧ φ))
                                               (λ t → eval (M , t ⊧ ψ)) s

private primitive primShowNat : ℕ → String

genSmv : CTLProblem → String
genSmv (M , s ⊧ φ) = header ++ input ++ label' ++ vars ++ init ++ trans ++ spec
  where
    header : String
    header = "MODULE main\n"

    input : String
    input = "IVAR\n-- from 0 to max action -1\n  input : {" ++
              natlist (maxfin (action M)) ++
            "};\n"

    label' : String
    label' = "DEFINE\n  labels := [" ++ ∀fin (λ s' → "{ -1" ++
              ∀fin {atom M} (λ a → if label M s' a then ("," ++ primShowNat (toℕ a)) else "")
                "" ++ "}") "," ++ "];\n"

    vars : String
    vars = "VAR\n  state : {" ++ natlist (state M) ++ "};\n"

    init : String
    init = "INIT\n  state = " ++ primShowNat (toℕ  s) ++ ";\n"

    trans : String
    trans = ("TRANS\n  next(state) =\n    case\n" ++ ∀fin (λ s' → ∀fin (λ a →
             "      state = " ++ primShowNat (toℕ s') ++ " & input = " ++ primShowNat (toℕ a) ++
                    " : " ++ primShowNat (toℕ (transition M s' a))) ";\n") ";\n") ++
             ";\n      TRUE : 0;\n    esac;\n"

    showctl : ∀{M} → CTL M → String
    showctl false      = "FALSE"
    showctl (~ φ)      = "!(" ++ showctl φ ++ ")"
```

```
    showctl (φ ∨ ψ)     = "(" ++ showctl φ ++ ") | (" ++ showctl ψ ++ ")"
    showctl (φ ∧ ψ)     = "(" ++ showctl φ ++ ") & (" ++ showctl ψ ++ ")"
    showctl P[ ap ]     = primShowNat (toℕ ap) ++ " in labels[state]"
    showctl (EX φ)      = "EX (" ++ showctl φ ++ ")"
    showctl (EG φ)      = "EG (" ++ showctl φ ++ ")"
    showctl E[ φ U ψ ] = "E[" ++ showctl φ ++ " U " ++ showctl ψ ++ "]"

    spec : String
    spec = "SPEC\n  " ++ showctl φ ++ ";\n"

genSinkedSmv : CTLProblem → String
genSinkedSmv (M , s ⊧ φ) = genSmv (mksink M , suc s ⊧ liftCTL φ)

lem-checkU : (ts : FSM) → (d1 d2 : ℕ) → T (d1 < suc d2) → (p q : State ts → Bool) → (s : State ts)
          → T (checkU ts d1 p q s) → T (checkU ts d2 p q s)
lem-checkU ts zero zero d1<d2 p q s checkp    = checkp
lem-checkU ts zero (suc n) d1<d2 p q s checkp = ∨-intror (p s ∧ᵇ _) (q s ∧ᵇ _) checkp
lem-checkU ts (suc zero) zero () p q s checkp
lem-checkU ts (suc (suc n)) zero () p q s checkp
lem-checkU ts (suc d1) (suc d2) d1<d2 p q s checkp
  = f∨g (f∧g { a = p s } id
           (λ r → lem-fin-∨ (proj₁ (x r))
                            (λ x' → checkU ts d2 p q
                                      (transition ts s x'))
                            (lem-checkU ts d1 d2 d1<d2 p q _ (proj₂ (x r))))  )
      id
      checkp

  where
    x : T (fin-∨-↔ (action ts s)
                   (λ a → checkU ts d1 p q (transition ts s a)))
      → Σ (Fin (action ts s))
          (λ x' → T (checkU ts d1 p q (transition ts s x')))
    x z = lem-fin-∨' (λ x' → checkU ts d1 p q (transition ts s x')) z

lem-check-Σ : (ts : FSM) → (n : ℕ) → (p : State ts → Bool) → (s : State ts) → T (check ts n p s)
           → Σ[ r : FinRun ts s n ] ((m : Fin (suc n)) → T (p (fnth ts r m)))
lem-check-Σ ts zero p s q = end , f
  where
    f : (m : Fin 1) → T (p (fnth ts {_} {s} end m))
    f zero = q
    f (suc ())
lem-check-Σ ts (suc n) p s q = Prod.map (next (proj₁ π)) x
                                  (lem-check-Σ ts n p (transition ts s (proj₁ π)) (proj₂ π))
  where
    π : Σ (Fin (action ts s))
          (λ i → T (check ts n p (transition ts s i)))
    π = (lem-fin-∨' (λ x → check ts n p (transition ts s x))
                    (∧-elimr (p s) q))

    x : ∀{a} → ((m : Fin (suc n))
      → T (p (fnth ts a m)))
      → (m : Fin (suc (suc n)))
      → T (p (fnth ts (next (proj₁ π) a) m))
    x y zero    = ∧-eliml q
    x y (suc i) = y i

tool : String
tool = "nusmv"

-- {-# BUILTIN ATPTOOL    tool         #-}
-- {-# BUILTIN ATPINPUT   genSinkedSmv #-}
-- {-# BUILTIN ATPDECPROC eval         #-}
```

```
module CTL.Proof where

-- contains the correctness proof of the CTL decision proc.
open import CTL.TransitionSystem
open import CTL.Definition
open import CTL.DecProc

open import Data.Fin hiding (_<_;#_;inject≤) renaming (_+_ to _F+_)
open import Data.Nat hiding (_<_)
open import Data.Bool renaming (_∧_ to _∧ᵇ_; _∨_ to _∨ᵇ_)
open import Data.List
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Fin.EqReasoning
open import Data.Fin.Arithmetic hiding (inject₁)
open import Data.Fin.Pigeon

open import PropIso

open import Relation.Binary.PropositionalEquality

open import Coinduction

open import CompleteInduction

open import Function

open FSM

-- define what it means for a point on a run to witness φ U ψ
WitnessesU : (M : FSM) → {s : State M} → ℕ → CTL (FSM.atom M) → CTL (FSM.atom M) → Set
WitnessesU M {s} k φ ψ = Σ (FinRun M s k)
                           (λ frun → Run M (last M frun)
                                   × ((j : Fin k) → ⟦ M , fnth M frun (inject₁ j) ⊨ φ ⟧)
                                   × ⟦ M , last M frun ⊨ ψ ⟧)

toWitness : ∀ M s φ ψ → (p : ⟦ M , s ⊨ E[ φ U ψ ] ⟧) → WitnessesU M {s} (proj₁ (proj₂ p)) φ ψ
toWitness M s φ ψ (run , zero , jp , kp) = end , run , (λ ()) , kp
toWitness M s φ ψ (run , suc n , jp , kp)
  = Prod.map (λ frun → next (run-head M run) frun)
             (λ {a} p → (proj₁ p) , jphelper a (proj₁ (proj₂ p)) , proj₂ (proj₂ p))
             (toWitness M (transition M s (proj₁ (run-decomp M run))) φ ψ
                          (run-tail M run , n , (λ x → jp (suc x)) , kp))
  where
    jphelper : (frun : FinRun M _ n)
             → ((j : Fin n) → ⟦ M , fnth M frun (inject₁ j) ⊨ φ ⟧)
             → (j : Fin (suc n))
             → ⟦ M , fnth M (next (proj₁ (run-decomp M run)) frun) (inject₁ j) ⊨ φ ⟧
    jphelper frun f zero    = jp zero tt
    jphelper frun f (suc i) = f i

fromWitness : (M : FSM)
            → (s : State M)
            → (φ : CTL (FSM.atom M))
            → (ψ : CTL (FSM.atom M))
            → {k : ℕ}
            → WitnessesU M {s} k φ ψ
            → ⟦ M , s ⊨ E[ φ U ψ ] ⟧
fromWitness M s φ ψ (end , run , jp , kp) = run , 0 , (λ _ ()) , kp
fromWitness M s φ ψ (next s' y , run , jp , kp)
  = Prod.map (λ run' → next s' (♯ run'))
             (λ {a} p → suc (proj₁ p) , jphelper (♯ a) (proj₁ (proj₂ p)) , proj₂ (proj₂ p))
             (fromWitness M (transition M s s') φ ψ (y , run , (λ z → jp (suc z)) , kp))
  where
    -- need to wrap ♯ for source code identity
    # : ∀{t} → Run M t → ∞ (Run M t)
    # {_} r = ♯ r

    jphelper : (run : ∞ (Run M (transition M s s')))
             → {x : ℕ}
             → ((j : ℕ) → T (j < x) → ⟦ M , nth M j (♭ run) ⊨ φ ⟧)
             → (j : ℕ)
             → T (j < suc x)
             → ⟦ M , nth M j (next s' run) ⊨ φ ⟧
    jphelper run' jp' zero p    = jp zero
    jphelper run' jp' (suc n) p = jp' n p

cut-out-loopU : (M : FSM)
          → {s : State M}
          → (φ ψ : CTL (FSM.atom M))
          → (k : ℕ)
          → ¬ T (k < state M)
          → (p : WitnessesU M {s} k φ ψ)
          → (pigeon : Σ (Fin (suc k)) × Fin (suc k))
                        (λ π → T (flt (proj₁ π) (proj₂ π))
                             × fnth M (proj₁ p) (proj₁ π) ≡ fnth M (proj₁ p) (proj₂ π)))
          → WitnessesU M {s} (toℕ (proj₁ (proj₁ pigeon)) + fin-remainder (proj₂ (proj₁ pigeon))) φ ψ
```

```
cut-out-loopU M φ ψ k ¬k<n (frun , run , jp , kp) ((k₁ , k₂) , 1<2 , eq)
  = frconc M (frtake M frun k₁)
             (subst (\ x → FinRun M x (fin-remainder k₂)) (trans (sym eq) (lem-frtake M frun k₁))
                    (frdrop M frun k₂))
  , subst (Run M) (trans (trans (lem-frdrop-last M frun k₂)
                                (cong' (λ x → FinRun M x (fin-remainder k₂))
                                       (λ s r → last M {_} {s} r)
                                       (trans (sym eq) (lem-frtake M frun k₁))
                                       (frdrop M frun k₂)))
                         (lem-frconc-last M (frtake M frun k₁) _)) run
  , jphelper frun k₁ k₂ 1<2 eq jp
  , subst (λ z → ⟦ M , z ⊨ ψ ⟧)
          (trans (trans (lem-frdrop-last M frun k₂)
                        (cong' (λ x → FinRun M x (fin-remainder k₂))
                               (λ x y → last M {_} {x} y)
                               (trans (sym eq) (lem-frtake M frun k₁))
                               (frdrop M frun k₂)))
                 (lem-frconc-last M (frtake M frun k₁) _)) kp
  where
    jphelper : {k : ℕ}
             → {s : State M}
             → (frun : FinRun M s k)
             → (k₁ k₂ : Fin (suc k))
             → T (flt k₁ k₂)
             → (eq : fnth M frun k₁ ≡ fnth M frun k₂)
             → ((j : Fin k) → ⟦ M , fnth M frun (inject₁ j) ⊨ φ ⟧)
             → (j : Fin (toℕ k₁ + fin-remainder k₂))
             → ⟦ M , fnth M (frconc M (frtake M frun k₁)
                                      (subst (\ x → FinRun M x (fin-remainder k₂))
                                             (trans (sym eq) (lem-frtake M frun k₁))
                                             (frdrop M frun k₂))) (inject₁ j) ⊨ φ ⟧
    jphelper {zero} frun' zero zero () eq' jp' j
    jphelper {zero} frun' zero (suc ()) k₁<k₂ eq' jp' j
    jphelper {zero} frun' (suc ()) k₂' k₁<k₂ eq' jp' j
    jphelper {suc k'} {s} (next s' y) zero k₂' k₁<k₂ eq' jp' j
      = subst (λ z → ⟦ M , z ⊨ φ ⟧)
              (trans (lem-frdrop-lift-nolast M (next s' y) k₂' j)
                     (cong' (λ x → FinRun M x (fin-remainder k₂'))
                            (λ x y' → fnth M {_} {x} y' (inject₁ j))
                            (trans (sym eq') refl)
                     (frdrop M (next s' y) k₂')))
              (jp' (inject≤ (<-+-rsuc (toℕ k₂') (fin-remainder k₂') (suc (suc k')))
                            (lem-fin-remainder-plus-lt (suc k₂')))
                                      (k₂' F+ j)))
    jphelper {suc k'} {s} frun' (suc k₁') k₂' k₁<k₂ eq' jp' zero = jp' zero
    jphelper {suc k'} {s} (next s' y) (suc k₁') zero () eq' jp' (suc j')
    jphelper {suc k'} {s} (next s' y) (suc k₁') (suc k₂') k₁<k₂ eq' jp' (suc j')
      = jphelper y k₁' k₂' k₁<k₂ eq' (λ z → jp' (suc z)) j'

reduceWitness : ∀ M {s} φ ψ {k} → WitnessesU M {s} k φ ψ
              → Σ[ k' ∶ ℕ ] (T (k' < state M) × WitnessesU M {s} k' φ ψ)
reduceWitness M {s} φ ψ {k} wit = completeind ρ ih k wit
  where
    ρ : ℕ → Set
    ρ k' = WitnessesU M {s} k' φ ψ → Σ ℕ (λ k0 → T (k0 < state M) × WitnessesU M {s} k0 φ ψ)

    ih : (k' : ℕ) → ((l : ℕ) → T (l < k') → ρ l) → ρ k'
    ih k' step wit with ex-mid (k' < (state M))
    ih k' step wit' | inj₁ x = k' , x , wit'
    ih k' step wit' | inj₂ y = step _ (<-fin-remainder (proj₁ (proj₁ pid)) (proj₂ (proj₁ pid))
                                          (proj₁ (proj₂ pid)))
                                      new-wit
      where
        pid = pigeon (suc k') (state M) (<-¬ k' (state M) y) (fnth M (proj₁ wit'))
        new-wit : WitnessesU M _ φ ψ
        new-wit = cut-out-loopU M φ ψ k' y wit' pid

inf-c : (M : FSM) → ∀{s'} → (r : Run M s') → (n : ℕ) → T (check M n (λ _ → true) s')
inf-c M r zero    = tt
inf-c M r (suc n) = lem-fin-∨ (run-head M r) _ (inf-c M (run-tail M r) n)

mutual
  complete-aux : ∀ M s φ → ⟦ M , s ⊨ φ ⟧ → T (eval (M , s ⊨ φ))
  complete-aux M s false p   = p
  complete-aux M s (~ φ) p   = lem-bool-neg-c (eval (M , s ⊨ φ)) (λ q → p (soundness-aux M s φ q))
  complete-aux M s (φ ∧ ψ) p = lem-bool-∧-c _ _ (Prod.map (complete-aux M s φ)(complete-aux M s ψ)p)
  complete-aux M s (φ ∨ ψ) p = lem-bool-∨-c _ _ (Sum.map (complete-aux M s φ)(complete-aux M s ψ) p)
  complete-aux M s P[ ap ] p = p
  complete-aux M s (EX φ) p  = lem-fin-∨ (run-head M $ proj₁ p) _
                                         (∧-intro _ _ (complete-aux M _ φ (proj₂ p))
                                                      (inf-c M (run-tail M (proj₁ p)) (state M)))
  complete-aux M s (EG φ) p  = eg (proj₁ p) (proj₂ p) (state M)
    where
      eg : ∀{s'}
         → (r : Run M s')
         → (p : (n : ℕ) → ⟦ M , nth M n r ⊨ φ ⟧)
         → (n : ℕ)
         → T (check M n (λ t → eval (M , t ⊨ φ)) s')
```

```
      eg r p' zero = complete-aux M _ φ (p' 0)
      eg r p' (suc n) = ∧-intro _ _ (complete-aux M _ φ (p' 0))
                                    (lem-fin-∨ (run-head M r)
                                               (λ a → check M n (λ s' → eval (M , s' ⊨ φ))
                                                                (transition M _ a))
                                               (eg (run-tail M r) (λ x → p' (suc x)) n))

  complete-aux M s E[ φ U ψ ] p = eu (reduceWitness M {s} φ ψ (toWitness M s φ ψ p))
    where
      eu' : ∀{s'}
         → (wit : Σ ℕ (λ k' → T (k' < state M) × WitnessesU M {s'} k' φ ψ))
         → T (checkU M (proj₁ wit) (λ t → eval (M , t ⊨ φ )) (λ t → eval (M , t ⊨ ψ )) s')
      eu' (zero , k<n , end , run , φp , ψp)
        = ∧-intro _ _ (complete-aux M _ ψ ψp) (inf-c M run (state M))
      eu' {s'} (suc k , k<n , next s0 y , run , φp , ψp)
        = ∨-introl ((eval (M , s' ⊨ φ)) ∧♭ _) _
                   (∧-intro _ _ (complete-aux M s' φ (φp zero))
                                (lem-fin-∨ s0 _
          (eu' (k , <-lsuc k (state M) k<n , y , run , (λ x → φp (suc x)) , ψp)) ))

      eu : ∀{s'}
         → (wit : Σ ℕ (λ k' → T (k' < state M) × WitnessesU M {s'} k' φ ψ))
         → T (checkU M (state M) (λ t → eval (M , t ⊨ φ)) (λ t → eval (M , t ⊨ ψ)) s')
      eu wit = lem-checkU M (proj₁ wit) (state M)
                 (<-rsuc (proj₁ wit) (state M) (proj₁ (proj₂ wit))) _ _ _ (eu' wit)

  soundness-aux : ∀ M s φ → T (eval (M , s ⊨ φ)) → ⟦ M , s ⊨ φ ⟧
  soundness-aux M s false p   = p
  soundness-aux M s (~ φ) p   = λ q → lem-bool-neg-s (eval (M , s ⊨ φ)) p (complete-aux M s φ q)
  soundness-aux M s (φ ∨ ψ) p = Sum.map (soundness-aux M s φ) (soundness-aux M s ψ)
                                        (lem-bool-∨-s (eval (M , s ⊨ φ)) _ p)
  soundness-aux M s (φ ∧ ψ) p = Prod.map (soundness-aux M s φ) (soundness-aux M s ψ)
                                         (lem-bool-∧-s (eval (M , s ⊨ φ)) _ p)
  soundness-aux M s P[ ap ] p = p
  soundness-aux M s (EX φ) p
    = ((λ π1 → Prod.map (λ x → next (proj₁ π1)
                                    (♯ lasso2run M (path2lasso-G M (proj₁
                                      (lem-check-Σ M (state M) (const true) _ x)))))
                        (soundness-aux M _ φ)
                        (swap (lem-bool-∧-s (eval (M , _ ⊨ φ)) _ (proj₂ π1)))) ∘'
      lem-fin-∨' (λ a → eval (M , transition M s a ⊨ φ) ∧♭
                        check M (state M) (λ _ → true) (transition M s a))) p
  soundness-aux M s (EG φ) p
    = Prod.map (λ r → lasso2run M (path2lasso-G M r))
        (λ {a} q n → soundness-aux M _ φ (lem-path2lasso-G M a (λ t → eval (M , t ⊨ φ)) q n))
        (lem-check-Σ M (state M) _ s p)
  soundness-aux M s E[ φ U ψ ] p = fromWitness M s φ ψ
    (frun s (state M) p , run s (state M) p , φp s (state M) p , ψp s (state M) p)
    where
      k : (s : State M)
        → (n : ℕ)
        → T (checkU M n (λ t → eval (M , t ⊨ φ)) (λ t → eval (M , t ⊨ ψ)) s)
        → ℕ
      k s' zero p'     = 0
      k s' (suc n) p' =
        let π : _ → _
            π x = lem-fin-∨' (λ a → checkU M n (λ t → eval (M , t ⊨ φ))
                                               (λ t → eval (M , t ⊨ ψ))
                                               (transition M s' a))
                             (∧-elimr (eval (M , s' ⊨ φ)) x)
        in ∨-elim (λ x → suc (k (transition M s' (proj₁ (π x))) n (proj₂ (π x)))) (const 0) p'

      frun : (s : State M)
           → (n : ℕ)
           → (p : T (checkU M n (λ t → eval (M , t ⊨ φ)) (λ t → eval (M , t ⊨ ψ)) s))
           → FinRun M s (k s n p)
      frun s' zero p' = end
      frun s' (suc n') p' with lem-bool-∨-s ((eval (M , s' ⊨ φ)) ∧♭ _) _ p'
      frun s' (suc n') p' | inj₁ x = next _ (frun _ n' _)
      frun s' (suc n') p' | inj₂ y = end

      run : (s : State M)
          → (n : ℕ)
          → (p : T (checkU M n (λ t → eval (M , t ⊨ φ)) (λ t → eval (M , t ⊨ ψ)) s))
          → Run M (last M (frun s n p))
      run s' zero p' = lasso2run M (path2lasso-G M (proj₁ (lem-check-Σ M (state M) (const true)
                                                            s' (∧-elimr (eval (M , s' ⊨ ψ)) p'))))
      run s' (suc n') p' with lem-bool-∨-s ((eval (M , s' ⊨ φ)) ∧♭ _) _ p'
      run s' (suc n') p' | inj₁ x = run _ n' _
      run s' (suc n)  p' | inj₂ y = lasso2run M (path2lasso-G M (proj₁ (lem-check-Σ M (state M)
                                                  (const true) s' (∧-elimr (eval (M , s' ⊨ ψ)) y))))

      φp : (s : State M)
         → (n : ℕ)
         → (p : T (checkU M n (λ t → eval (M , t ⊨ φ)) (λ t → eval (M , t ⊨ ψ)) s))
         → (j : Fin (k s n p))
         → ⟦ M , fnth M (frun s n p) (inject₁ j) ⊨ φ ⟧
      φp s' zero p' ()
```

```
      φp s' (suc n') p' j with lem-bool-∨-s ((eval (M , s' ⊧ φ)) ∧ᵇ _) _ p'
      φp s' (suc n') p' zero | inj₁ x = soundness-aux M s' φ (∧-eliml x)
      φp s' (suc n') p' (suc j) | inj₁ x = φp _ n' _ j
      φp s' (suc n') p' () | inj₂ y

      ψp : (s : State M)
         → (n : ℕ)
         → (p : T (checkU M n (λ t → eval (M , t ⊧ φ)) (λ t → eval (M , t ⊧ ψ)) s))
         → ⟦ M , last M (frun s n p) ⊧ ψ ⟧
      ψp s' zero p' = soundness-aux M s' ψ (∧-eliml p')
      ψp s' (suc n') p' with lem-bool-∨-s ((eval (M , s' ⊧ φ)) ∧ᵇ _) _ p'
      ψp s' (suc n') p' | inj₁ x = ψp _ n' _
      ψp s' (suc n') p' | inj₂ y = soundness-aux M s' ψ (∧-eliml y)

soundness : ∀ γ → T (eval γ) → ⟦ γ ⟧
soundness (M , s ⊧ φ) = soundness-aux M s φ

complete : ∀ γ → ⟦ γ ⟧ → T (eval γ)
complete (M , s ⊧ φ) = complete-aux M s φ

--{-# BUILTIN ATPSOUND soundness #-}
--{-# BUILTIN ATPCOMPLETE complete #-}
```

```
module Data.Fin.Pigeon where

open import PropIso renaming (_<_ to nlt)

open import Data.Fin.Arithmetic
open import Data.Fin hiding (inject₁)
open import Data.Nat
open import Data.Bool
open import Data.Product
open import Data.Sum as Sum

open import Relation.Binary.PropositionalEquality

private
  ψ₁ : {n : ℕ} → (p : Fin n → Bool) → Set
  ψ₁ {n} p = Σ[ k ∶ Fin n × Fin n ] T (flt (proj₁ k) (proj₂ k)) × T (p (proj₁ k)) × T (p (proj₂ k))

  ψ₂ : {n : ℕ} → (p : Fin n → Bool) → Set
  ψ₂ {n} p = Σ[ k ∶ Fin n ] (T (p k) × ((j : Fin n) → j ≢ k → ¬ (T (p j))))

  ψ₃ : {n : ℕ} → (p : Fin n → Bool) → Set
  ψ₃ {n} p = (k : Fin n) → ¬ T (p k)

  ψ : {n : ℕ} → (p : Fin n → Bool) → Set
  ψ p = ψ₁ p ⊎ ψ₂ p ⊎ ψ₃ p

Pigeon : {n m : ℕ} → (f : Fin n → Fin m) → Set
Pigeon {n} f = Σ[ k ∶ Fin n × Fin n ] (T (flt (proj₁ k) (proj₂ k)) × f (proj₁ k) ≡ f (proj₂ k))

tri-choice-step-c2 : {n : ℕ} → (p : Fin (suc n) → Bool) → ψ₂ (p ∘ suc) → T (p zero) ⊎ ¬ (T (p zero))
                   → ψ₁ p ⊎ ψ₂ p
tri-choice-step-c2 p (x , (y , z)) (inj₁ x') = inj₁ ((zero , suc x) , tt , x' , y)
tri-choice-step-c2 {n} p (x , (y , z)) (inj₂ y') = inj₂ (suc x , y , lem)
  where
    lem : (j : Fin (suc n)) → (j ≢ suc x) → ¬ T (p j)
    lem zero    p' = y'
    lem (suc i) p' = z i (λ k → p' (cong suc k))

tri-choice-step-c3 : ∀ {n} (p : Fin (suc n) → Bool) → ((k : Fin n) → ¬ T (p (suc k)))
                   → T (p zero) ⊎ ¬ (T (p zero)) → ψ₂ p ⊎ ψ₃ p
tri-choice-step-c3 {n} p anz (inj₁ x) = inj₁ (zero , x , lem)
  where
    lem : (j : Fin (suc n)) → (j ≢ zero) → ¬ (T ( (p j)))
    lem zero    p' = ⊥-elim (p' refl)
    lem (suc i) p' = anz i
tri-choice-step-c3 {n} p anz (inj₂ y) = inj₂ lem
  where
    lem : (k : Fin (suc n)) → ¬ (T ( (p k)))
    lem zero    = y
    lem (suc i) = anz i

tri-choice-step : {n : ℕ} → (p : Fin (suc n) → Bool) → ψ (p ∘ suc) → ψ p
tri-choice-step p (inj₁ ((x , y) , q)) = inj₁ ((suc x , suc y) , q)
tri-choice-step p (inj₂ (inj₁ x)) = Sum.map id inj₁ $ tri-choice-step-c2 p x (ex-mid (p zero))
tri-choice-step p (inj₂ (inj₂ y)) = inj₂ $ tri-choice-step-c3 p y (ex-mid (p zero))

tri-choice : {n : ℕ} → (p : Fin n → Bool) → ψ p
tri-choice {zero} p = inj₂ (inj₂ (λ ()))
tri-choice {suc n} p = tri-choice-step p (tri-choice {n} (p ∘ suc))

isZero : {n : ℕ} → Fin n → Bool
isZero zero = true
isZero (suc i) = false

lem-iszero : {n : ℕ} → {x : Fin (suc n)} → T (isZero x) → x ≡ zero
lem-iszero {_} {zero} p = refl
lem-iszero {_} {suc i} ()

fin1-eq : {a b : Fin 1} → a ≡ b
fin1-eq {zero} {zero} = refl
fin1-eq {suc ()}
fin1-eq {_} {suc ()}

pred' : {n : ℕ} → Fin (suc (suc n)) → Fin (suc n)
pred' zero = zero
pred' (suc i) = i

lem-pred' : ∀ {n} {a b : Fin (suc (suc n))} (anz : ¬ T ( (isZero a))) (bnz : ¬ T ( (isZero b)))
          → pred' a ≡ pred' b → a ≡ b
lem-pred' {n} {zero} {b} anz bnz p = ⊥-elim $ anz tt
lem-pred' {n} {suc i} {zero} anz bnz p = ⊥-elim $ bnz tt
lem-pred' {n} {suc i} {suc i'} anz bnz p = cong suc p

-- inject suc
is : ∀ {n} (k : Fin (suc n)) x → T (flt (inject₁ x) k) ⊎ ¬ (T (flt (inject₁ x) k)) → Fin (suc n)
is k x (inj₁ x<k) = inject₁ x
is k x (inj₂ x>=k) = suc x
```

```
lem-is-mono : ∀ {n} (k : Fin (suc n)) (a b : Fin n)
            → (p : T (flt (inject₁ a) k) ⊎ ¬ T (flt (inject₁ a) k))
            → (q : T (flt (inject₁ b) k) ⊎ ¬ T (flt (inject₁ b) k))
            → T (flt a b) → T (flt (is k a p) (is k b q))
lem-is-mono k a b (inj₁ a<k) (inj₁ b<k) a<b rewrite toℕ-inj-eq a 1 _ refl 0 _ refl
                                                  | toℕ-inj-eq b 1 _ refl 0 _ refl = a<b
lem-is-mono k a b (inj₁ a<k) (inj₂ b>=k) a<b rewrite toℕ-inj-eq a 1 _ refl 0 _ refl
  = <-rsuc _ (toℕ b) a<b
lem-is-mono {n} k a b (inj₂ a≮k) (inj₁ b<k) a<b rewrite toℕ-inj-eq a 1 _ refl 0 _ refl
                                                      | toℕ-inj-eq b 1 _ refl 0 _ refl
  = ⊥-elim $ <-¬' (toℕ b) (toℕ a)
                  (<-trans' _ (toℕ k) (suc _) b<k (<-¬ (toℕ a) (toℕ k) a≮k))
                  (<-rsuc (toℕ a) _ a<b)
lem-is-mono k a b (inj₂ a>=k) (inj₂ b>=k) a<b = a<b

lem-is-not-k : ∀ {n} {j} {k : Fin (suc n)} (p : T (flt (inject₁ j) k) ⊎ ¬ (T (flt (inject₁ j) k)))
             → (is k j p) ≠ k
lem-is-not-k {n} {j} (inj₁ x) refl = <-¬refl (toℕ (inject₁ j)) x
lem-is-not-k {n} {j} {._} (inj₂ y) refl rewrite toℕ-inj-eq j 1 _ refl 0 _ refl = y (<-ord (toℕ j))

lem-is-not-zero : {n m : ℕ} (f : Fin (suc n) → Fin (suc m)) (c2 : ψ₂ (isZero ∘ f)) (j : Fin n)
                → ¬ T (isZero (f (is (proj₁ c2) j (ex-mid (flt (inject₁ j) (proj₁ c2))))))
lem-is-not-zero f (k , kz , jnz) j = jnz (is k j (ex-mid ((flt (inject₁ j) k))))
                                         (lem-is-not-k {_} {j} (ex-mid (flt (inject₁ j) k)))

lem-reduce-f : {n m : ℕ} (f : Fin (suc (suc n)) → Fin (suc (suc m))) (point : Fin (suc (suc n)))
             → Fin (suc n) →  Fin (suc m)
lem-reduce-f f p k = pred' (f (is p k (ex-mid (nlt (toℕ (inject₁ k)) (toℕ p)))))

lem-pidgen-c2 : (n m : ℕ)
              → T (nlt (suc (suc m)) (suc (suc n)))
              → (f : Fin (suc (suc n)) → Fin (suc (suc m)))
              → (c2 : ψ₂ (isZero ∘ f))
              → Pigeon (lem-reduce-f f (proj₁ c2))
              → Pigeon f
lem-pidgen-c2 n m nlt f (k' , k'z , jnz) ((k , l) , k<l , eq)
  = (is k' k (ex-mid (flt (inject₁ k) k'))
    , is k' l (ex-mid (flt (inject₁ l) k')))
    , lem-is-mono k' k l (ex-mid (flt (inject₁ k) k'))
        (ex-mid (flt (inject₁ l) k')) k<l
    , lem-pred' (lem-is-not-zero f (k' , k'z , jnz) k)
                (lem-is-not-zero f (k' , k'z , jnz) l) eq

lem-pidgen-c3 : (n m : ℕ)
              → T (nlt (suc (suc m)) (suc (suc n)))
              → (f : Fin (suc (suc n)) → Fin (suc (suc m)))
              → (c3 : ψ₃ (isZero ∘ f))
              → Pigeon (pred' ∘ f ∘ inject₁)
              → Pigeon f
lem-pidgen-c3 n m nlt f c3 ((k , l) , k<l , eq) rewrite toℕ-inj-eq k 0 _ refl 1 _ refl
                                                      | toℕ-inj-eq l 0 _ refl 1 _ refl
  = (inject₁ k , inject₁ l) , k<l , lem-pred' (c3 (inject₁ k)) (c3 (inject₁ l)) eq

mutual
  pidgen' : (n m : ℕ) → T (nlt m n) → (f : Fin (suc n) → Fin (suc m)) → Pigeon f
  pidgen' zero zero () f
  pidgen' (suc n) zero p f = (zero , (suc zero)) , tt , fin1-eq
  pidgen' zero (suc n') () f
  pidgen' (suc n) (suc m) nlt f  = lem-pidgen n m nlt  f (tri-choice (isZero ∘ f))

  lem-pidgen : ∀ n m → T (nlt m n) → (f : Fin (suc (suc n)) → Fin (suc (suc m))) → ψ (isZero ∘ f)
             → Pigeon f
  lem-pidgen n m nlt f (inj₁ ((k , l) , k<l , kz , lz))
    = (k , l)
      , k<l
      , subst (λ j → f k ≡ j) (sym (lem-iszero lz))
              (subst (λ j → j ≡ zero) (sym (lem-iszero kz)) refl)

  lem-pidgen n m nlt f (inj₂ (inj₁ c2)) = lem-pidgen-c2 n m nlt f c2
                                          (pidgen' n m nlt (lem-reduce-f f (proj₁ c2)))
  lem-pidgen n m nlt f (inj₂ (inj₂ c3)) = lem-pidgen-c3 n m nlt f c3
                                          (pidgen' n m nlt (λ z → pred' (f (inject₁ z))))

pigeon : (n m : ℕ) → T (nlt m n) → (f : Fin n → Fin m) → Pigeon f
pigeon zero m () f
pigeon (suc n) (suc m) p f = pidgen' n m p f
pigeon (suc n) zero p f with f zero ; ...| ()
```

```
module CTL.Sink where

open import Data.Fin hiding (_<_;#_)
open import Data.Nat hiding (_<_)
open import Data.List
open import Data.Bool hiding (_∧_;_∨_)
open import Data.Product as Prod
open import Data.Sum as Sum

open import Relation.Binary.PropositionalEquality

open import CTL.TransitionSystem
open import CTL.Definition

open import Coinduction

open import PropIso

{-
  adds a sink to the transition system, so that the relation
  is assured to be total. also the ctl formula are transformed
  accordingly.
-}

mksink : FSM → FSM
mksink ts
  = fsm (suc (FSM.state ts)) (suc (FSM.atom ts)) act (init (FSM.initial ts)) trans' lbl
  where
    act : Fin (suc (FSM.state ts)) → ℕ
    act zero    = 1
    act (suc i) = suc (FSM.action ts i)

    init : ∀{n} → List (Fin n) → List (Fin (suc n))
    init []       = []
    init (a ∷ as) = suc a ∷ init as

    trans' : (s : Fin (suc (FSM.state ts))) (a : Fin (act s)) → Fin (suc (FSM.state ts))
    trans' zero a          = zero
    trans' (suc s) zero    = zero
    trans' (suc s) (suc a) = suc (FSM.transition ts s a)

    lbl : Fin (suc (FSM.state ts)) → Fin (suc (FSM.atom ts)) → Bool
    lbl zero zero       = true
    lbl zero (suc a)    = false
    lbl (suc s) zero    = false
    lbl (suc s) (suc a) = FSM.label ts s a

private
  ¬sink : ∀ {a} → CTL (suc a)
  ¬sink = ~ P[ zero ]

liftCTL : ∀ {n} → CTL n → CTL (suc n)
liftCTL false      = false
liftCTL (~ φ)      = ~ (liftCTL φ)
liftCTL (φ ∨ ψ)    = liftCTL φ ∨ liftCTL ψ
liftCTL (φ ∧ ψ)    = liftCTL φ ∧ liftCTL ψ
liftCTL P[ ap ]    = P[ suc ap ]
liftCTL (EX φ)     = (EX (liftCTL φ ∧ EG ¬sink))
liftCTL (EG φ)     = EG (liftCTL φ ∧ ¬sink)
liftCTL E[ φ U ψ ] = E[ liftCTL φ ∧ ¬sink U liftCTL ψ ∧ EG ¬sink ]

runlift : ∀{M s} → Run M s → Run (mksink M) (suc s)
runlift {M} r = next (suc (run-head M r)) (♯ (runlift (run-tail M r)))

rundown : ∀{M s} → (p : ⟦ mksink M , suc s ⊧ EG ¬sink ⟧) → Run M s
rundown (next zero x , proj₂)     = ⊥-elim $ proj₂ 1 tt
rundown (next (suc s') x , proj₂) = next s' (♯ rundown ((♭ x) , (proj₂ ∘ suc)))

private
  neverinsink : ∀ {M s'} → (a : Run M s') → (n : ℕ)
    → ¬ T (FSM.label (mksink M) (nth (mksink M) n (runlift a)) zero)
  neverinsink a zero p'        = p'
  neverinsink {M} a (suc n) p' = neverinsink (run-tail M a) n p'

  neverinsink' : ∀ {M s'} → (k : ℕ) → (a : Run M s') → (n : ℕ)
      → ¬ T (FSM.label (mksink M) (nth (mksink M) n (run-drop (mksink M) k (runlift a))) zero)
  neverinsink' zero a zero p        = p
  neverinsink' {M} zero a (suc n) p = neverinsink' zero (run-tail M a) n p
  neverinsink' {M} (suc k) a n p    = neverinsink' k (run-tail M a) n p


lem-nth-eq : ∀ M s (r : Run M s) n → suc (nth M n r) ≡ nth (mksink M) n (runlift r)
lem-nth-eq M s r zero    = refl
lem-nth-eq M s r (suc n) = lem-nth-eq M _ (run-tail M r) n

lem-nth-eq' : (M : FSM)
            → (s : _)
```

```
            → (r : Run (mksink M) (suc s))
            → (p : ∀ n → ⟦ mksink M , nth (mksink M) n r ⊨ ¬sink ⟧)
            → (n : ℕ) → nth (mksink M) n r ≡ suc (nth M n (rundown (r , p)))
lem-nth-eq' M s (next zero x) p n           = ⊥-elim $ p 1 tt
lem-nth-eq' M s (next (suc s') x) p zero    = refl
lem-nth-eq' M s (next (suc s') x) p (suc n) = lem-nth-eq' M _ (♭ x) (p ∘ suc) n

sinked-run : ∀ M s → Run M s → ⟦ mksink M , suc s ⊨ EG ¬sink ⟧
sinked-run M s r = runlift r , (neverinsink r)

sinked-run-eg : ∀ M φ ψ
              → (∀ s → ⟦ M , s ⊨ φ ⟧ → ⟦ mksink M , suc s ⊨ ψ ⟧)
              → ∀ s → ⟦ M , s ⊨ EG φ ⟧ → ⟦ mksink M , suc s ⊨ EG (ψ ∧ ¬sink) ⟧
sinked-run-eg M φ ψ p s q
  = Prod.map runlift (λ {a} eg n → subst (λ k → ⟦ mksink M , k ⊨ ψ ⟧)
                                         (lem-nth-eq M _ a n)
                                         (p (nth M n a) (eg n))
                                 , neverinsink a n) q

sinked-run-eg' : ∀ M φ ψ
               → (∀ s → ⟦ mksink M , suc s ⊨ ψ ⟧ → ⟦ M , s ⊨ φ ⟧)
               → ∀ s → ⟦ mksink M , suc s ⊨ EG (ψ ∧ ¬sink) ⟧ → ⟦ M , s ⊨ EG φ ⟧
sinked-run-eg' M φ ψ p s q = (rundown (proj₁ q , (λ n → proj₂ (proj₂ q n))))
                           , λ n → p (nth M n (rundown (proj₁ q , (λ n₁ → proj₂ (proj₂ q n₁)))))
                                     (subst (λ k → ⟦ mksink M , k ⊨ ψ ⟧)
                                            (lem-nth-eq' M _ (proj₁ q) (λ n₁ → proj₂ (proj₂ q n₁)) n)
                                            (proj₁ (proj₂ q n)))

sinked-run-eu : ∀ M φ φ' ψ ψ'
              → (∀ s → ⟦ M , s ⊨ φ ⟧ → ⟦ mksink M , suc s ⊨ φ' ⟧)
              → (∀ s → ⟦ M , s ⊨ ψ ⟧ → ⟦ mksink M , suc s ⊨ ψ' ⟧)
              → ∀ s → ⟦ M , s ⊨ E[ φ U ψ ] ⟧
              → ⟦ mksink M , suc s ⊨ E[ φ' ∧ ¬sink U (ψ' ∧ EG ¬sink) ] ⟧
sinked-run-eu M φ φ' ψ ψ' p q s r
  = Prod.map runlift (λ {a} → Prod.map id (λ {k} → Prod.map
     (λ jp j j<k → subst (λ k → ⟦ mksink M , k ⊨ φ' ⟧) (lem-nth-eq M _ a j) (p _ (jp j j<k))
                   , (neverinsink a j))
     (λ kp → subst (λ k₁ → ⟦ mksink M , k₁ ⊨ ψ' ⟧) (lem-nth-eq M _ a k) (q _ kp)
              , run-drop _ k (runlift a) , neverinsink' k a))) r

Z : {A : ℕ → Set} → (n : ℕ) → A 0 → ((m : ℕ) → T (m < n)  → A (suc m)) → A n
Z zero a0 an    = a0
Z (suc n) a0 an = an n (<-ord n)

mutual
  lemma : (M : FSM)
        → (s : FSM.State M)
        → (φ : CTL (FSM.atom M))
        → ⟦ M , s ⊨ φ ⟧
        → ⟦ mksink M , (suc s) ⊨ liftCTL φ ⟧
  lemma M s false p      = p
  lemma M s (~ φ) p      = λ x → p (lemma' M s φ x)
  lemma M s (φ ∨ ψ) p    = Sum.map (lemma _ _ φ) (lemma _ _ ψ) p
  lemma M s (φ ∧ ψ) p    = Prod.map (lemma _ _ φ) (lemma _ _ ψ) p
  lemma M s P[ ap ] p    = p
  lemma M s (EX φ) p     = Prod.map runlift (λ {a} q → lemma M _ φ q , runlift (run-tail M a)
                                                                              , neverinsink (run-tail M a)) p
  lemma M s (EG φ) p     = sinked-run-eg M φ (liftCTL φ) (λ s₁ → lemma M s₁ φ) s p
  lemma M s E[ φ U ψ ] p = sinked-run-eu M φ (liftCTL φ) ψ (liftCTL ψ)
                                        (λ s₁ → lemma M s₁ φ) (λ s₁ → lemma M s₁ ψ) s p

  lemma' : (M : FSM)
         → (s : FSM.State M)
         → (φ : CTL (FSM.atom M)) → ⟦ mksink M , (suc s) ⊨ liftCTL φ ⟧
         → ⟦ M , s ⊨ φ ⟧
  lemma' M s false p   = p
  lemma' M s (~ φ) p   = λ x → p (lemma M s φ x)
  lemma' M s (φ ∨ ψ) p = Sum.map (lemma' M s φ) (lemma' M s ψ) p
  lemma' M s (φ ∧ ψ) p = Prod.map (lemma' M s φ) (lemma' M s ψ) p
  lemma' M s P[ ap ] p = p
  lemma' M s (EX φ) (next zero y , φp , g¬sink) = ⊥-elim (proj₂ g¬sink 0 tt)
  lemma' M s (EX φ) (next (suc a) y , φp , g¬sink) = next a (♯ rundown g¬sink) , lemma' M _ φ φp
  lemma' M s (EG φ) p = sinked-run-eg' M φ (liftCTL φ) (λ s₁ → lemma' M s₁ φ) s p

  lemma' M s E[ φ U ψ ] (urun , zero , jp , kp)
    = (rundown (proj₂ kp)) , zero , (λ j ()) , lemma' M _ ψ (proj₁ kp)
  lemma' M s E[ φ U ψ ] (next zero y , suc zero , jp , kp)     = ⊥-elim (proj₂ (proj₂ kp) 0 tt)
  lemma' M s E[ φ U ψ ] (next zero y , suc (suc n) , jp , kp) = ⊥-elim (proj₂ (jp 1 tt) tt)
  lemma' M s E[ φ U ψ ] (next (suc a) y , suc k , jp , kp)
    = Prod.map (λ r → next a (# r)) (λ {νrun} → Prod.map suc (λ {k'} → Prod.map
               (λ jp' j j<k → Z {λ j' → ⟦ M , nth M j' (next a (# νrun)) ⊨ φ ⟧} j
                 (lemma' M _ φ (proj₁ (jp 0 tt))) (λ m m<j → jp' m (<-trans' m j (suc k') m<j j<k)))
                 id)) (lemma' M _ (E[ φ U ψ ]) ((♭ y) , k , (λ j j<k → jp (suc j) j<k) , kp))
    where
      -- needed for source code identity
      # : {A : Set} → A → ∞ A
      # = ♯_
```

```
module Data.Fin.Record where

open import Data.Fin hiding (inject+;_<_;_+_;inject₁)
open import Data.Fin.Arithmetic
open import Data.Fin.EqReasoning
open import Data.Nat hiding (_<_)
open import Data.List
import Data.List.Util as L
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Bool

open import PropIso

open import Relation.Binary.PropositionalEquality

Record : List ℕ → Set
Record [] = ⊤
Record (n ∷ xs) = Fin n × Record xs

fin-prod : {n m : ℕ} → Fin (suc n) × Fin (suc m) → Fin (suc n * suc m)
fin-prod                 (zero   , zero)        = zero
fin-prod                 (zero   , suc zero)    = suc zero
fin-prod {zero }         (zero   , suc (suc l)) = suc (fin-prod {zero} (zero , suc l))
fin-prod {suc n} {suc m} (zero   , suc (suc l)) = inject+ (suc (suc m))
                                                    (fin-prod {n} {suc m} (zero , suc (suc l)))
fin-prod {zero }         (suc () , l)
fin-prod {suc n} {m}     (suc k  , l)           = fin-prod (k , l) suc^ suc m

fin-proj : {n m : ℕ} → Fin (suc n * suc m) → Fin (suc n) × Fin (suc m)
fin-proj {zero}  {zero}  x       = zero , zero
fin-proj {zero}  {suc m} zero    = zero , zero
fin-proj {zero}  {suc m} (suc x) = Prod.map id suc (fin-proj {zero} {m} x)
fin-proj {suc n} {m}     x       = [ (λ x' → zero , fromℕ< (toℕ x) x') ,
                                     (λ x' → Prod.map suc id (fin-proj {n} (finminus {suc m} x x')))
                                   ]' (ex-mid (toℕ x < suc m))

lem-fin-prod-<' : (n m o : ℕ) → (l : Fin (suc m))
                → T (toℕ (inject+ o (fin-prod (zero {n} , l))) < suc m)
lem-fin-prod-<' zero m zero zero = tt
lem-fin-prod-<' zero .(suc n) zero (suc (zero {n})) = tt
lem-fin-prod-<' zero .(suc n) zero (suc (suc {n} l)) = lem-fin-prod-<' 0 n 0 (suc l)
lem-fin-prod-<' (suc n) m zero zero = tt
lem-fin-prod-<' (suc n) .(suc n') zero (suc (zero {n'})) = tt
lem-fin-prod-<' (suc n) .(suc m) zero (suc (suc {m} l)) = lem-fin-prod-<' n (suc m) (suc (suc m))
                                                            (suc (suc l))
lem-fin-prod-<' n m (suc o) l
  rewrite sym $ toℕ-inj-eq (fin-prod (zero {n} , l)) o _ refl (suc o) _ refl
  = lem-fin-prod-<' n m o l

lem-fin-prod-0' : (n m o : ℕ) → (l : Fin (suc m))
                → (z : T (toℕ (inject+ o (fin-prod (zero {n} , l))) < suc m))
                → fromℕ< (toℕ (inject+ o (fin-prod (zero {n} , l)))) z ≡ l
lem-fin-prod-0' zero m zero zero z = refl
lem-fin-prod-0' zero .(suc n) zero (suc (zero {n})) z = refl
lem-fin-prod-0' zero .(suc m) zero (suc (suc {m} l)) z = cong suc (lem-fin-prod-0' 0 m 0 (suc l) z)
lem-fin-prod-0' (suc n) m zero zero z = refl
lem-fin-prod-0' (suc n) .(suc n') zero (suc (zero {n'})) z = refl
lem-fin-prod-0' (suc n) .(suc m) zero (suc (suc {m} l)) z
  = lem-fin-prod-0' n (suc m) (suc (suc m)) (suc (suc l)) z
lem-fin-prod-0' n m (suc o) l z
  rewrite sym $ toℕ-inj-eq (fin-prod (zero {n} , l)) o _ refl (suc o) _ refl
  = lem-fin-prod-0' n m o l z

fin-prod-iso1 : (n m : ℕ) → (r : Fin (suc n) × Fin (suc m)) → fin-proj (fin-prod r) ≡ r
fin-prod-iso1 zero zero (zero , zero) = refl
fin-prod-iso1 zero zero (zero , suc ())
fin-prod-iso1 zero zero (suc () , l)
fin-prod-iso1 zero (suc m) (zero , zero) = refl
fin-prod-iso1 zero (suc m) (zero , suc zero)
  = cong (Prod.map id suc) (fin-prod-iso1 zero m (zero , zero))
fin-prod-iso1 zero (suc m) (zero , suc (suc l))
  = cong (Prod.map id suc) (fin-prod-iso1 zero m (zero , suc l))
fin-prod-iso1 zero (suc m) (suc () , l)
fin-prod-iso1 (suc n) m (zero , zero) = refl
fin-prod-iso1 (suc n) .(suc n') (zero , suc (zero {n'})) = refl
fin-prod-iso1 (suc n) .(suc m) (zero , suc (suc {m} l))
  with ex-mid (toℕ (inject+ (suc (suc m)) (fin-prod {n} (zero , suc (suc l))) )) < suc (suc m))
fin-prod-iso1 (suc n) .(suc m) (zero , suc (suc {m} l)) | inj₁ x
  = cong (λ p → zero , p) (lem-fin-prod-0' n (suc m) (suc (suc m)) (suc (suc l)) x)
fin-prod-iso1 (suc n) .(suc m) (zero , suc (suc {m} l)) | inj₂ y
  = ⊥-elim (y $ lem-fin-prod-<' n (suc m) (suc (suc m)) (suc (suc l)))
fin-prod-iso1 (suc n) zero (suc k , l) = cong (Prod.map suc id) (fin-prod-iso1 n 0 (k , l))
fin-prod-iso1 (suc n) (suc m) (suc k , l) with (ex-mid (toℕ (fin-prod (k , l) suc^ m) < m))
fin-prod-iso1 (suc n) (suc m) (suc {.(suc n)} k , l) | inj₁ x
  = ⊥-elim ((lem-fin-suc^ m (fin-prod (k , l))) x)
fin-prod-iso1 (suc n) (suc m) (suc {.(suc n)} k , l) | inj₂ y
```

```
  = cong (Prod.map suc id) (subst (λ x → fin-proj x ≡ (k , l))
                                  (lem-finminus-id m (fin-prod (k , l)) y)
                                  (fin-prod-iso1 n (suc m) (k , l)))

-- minor suc
lem-fin-prod-n0 : ∀ n (π : Fin 1 × Fin (suc n)) → suc (fin-prod π) ≡ fin-prod ((Prod.map id suc) π)
lem-fin-prod-n0 n (zero , zero) = refl
lem-fin-prod-n0 n (zero , suc i) = refl
lem-fin-prod-n0 n (suc () , y)

lem-fin-prod-n0' : (n m : ℕ) → (i : Fin (suc n * suc m)) → (x : T (toℕ i < suc m))
                 → fin-prod {n} {m} (zero , (fromℕ< (toℕ i) x)) ≡ i
lem-fin-prod-n0' n m zero x = refl
lem-fin-prod-n0' n zero (suc i) ()
lem-fin-prod-n0' n (suc n') (suc zero) x = refl
lem-fin-prod-n0' zero (suc m) (suc (suc i)) x = cong suc (lem-fin-prod-n0' 0 m (suc i) x)
lem-fin-prod-n0' (suc n) (suc m) (suc (suc i)) x = trans ih' (lem-fromtoℕ-inj+ _ (suc (suc m)) _ _)
  where
    ih = lem-fin-prod-n0' n (suc m) (fromℕ< (toℕ (suc (suc i)))
                                            (<-weaken (toℕ i) m (n * suc (suc m)) x))
                            (subst (λ k → T (k < m)) (lem-tofromtoℕ i (<-weaken (toℕ i) m
                                                                      (n * suc (suc m)) x)) x)

    ih' : fin-prod {suc n} {suc m} (zero , fromℕ< (toℕ (suc (suc i))) x)
          ≡ inject+ (suc (suc m)) (suc (suc (fromℕ< (toℕ i)
                                    (<-weaken (toℕ i) m (n * suc (suc m)) x))))
    ih' = cong (inject+ (suc (suc m)))
           (trans (cong (λ k → fin-prod {n} {suc m} (zero , suc (suc k)))
                        (cong' (λ k → T (k < m)) (fromℕ< {m})
                               (lem-tofromtoℕ i (<-weaken (toℕ i) m (n * suc (suc m)) x)) x)) ih)

fin-prod-iso2 : (n m : ℕ) → (r : Fin (suc n * suc m)) → fin-prod {n} {m} (fin-proj r) ≡ r
fin-prod-iso2 zero zero zero = refl
fin-prod-iso2 zero zero (suc ())
fin-prod-iso2 (suc n) zero zero = refl
fin-prod-iso2 (suc n) zero (suc i) = cong suc (fin-prod-iso2 n 0 i)
fin-prod-iso2 zero (suc m) zero = refl
fin-prod-iso2 zero (suc m) (suc i)
  = trans (sym (lem-fin-prod-n0 m (fin-proj i))) (cong suc (fin-prod-iso2 0 m i))
fin-prod-iso2 (suc n) (suc m) zero = refl
fin-prod-iso2 (suc n) (suc m) (suc i) with ex-mid (toℕ i < suc m)
fin-prod-iso2 (suc n) (suc m) (suc i) | inj₁ x = lem-fin-prod-n0' (suc n) (suc m) (suc i) x
fin-prod-iso2 (suc n) (suc m) (suc i) | inj₂ y
  = cong suc (trans (cong (λ k → k suc^ suc m) (fin-prod-iso2 n (suc m) (finminus {suc m} i y)))
                    (sym (lem-finminus-id' (suc m) i y)))

fin-pair : {n m : ℕ} → Fin n × Fin m → Fin (n * m)
fin-pair {zero} π = proj₁ π
fin-pair {suc n} {zero} π = elim-fin0 (proj₂ π)
fin-pair {suc n} {suc n'} π = fin-prod π

fin-unpair : {n m : ℕ} → Fin (n * m) → Fin n × Fin m
fin-unpair {zero} ()
fin-unpair {suc n} {zero} x = Prod.map suc id (fin-unpair x)
fin-unpair {suc n} {suc n'} x = fin-proj x

fin-pair-iso1 : (n m : ℕ) → (r : Fin n × Fin m) → fin-unpair (fin-pair r) ≡ r
fin-pair-iso1 zero m (() , y)
fin-pair-iso1 (suc n) zero (x , ())
fin-pair-iso1 (suc n) (suc m) r = fin-prod-iso1 n m r

fin-pair-iso2 : (n m : ℕ) → (r : Fin (n * m)) → fin-pair {n} (fin-unpair r) ≡ r
fin-pair-iso2 zero m ()
fin-pair-iso2 (suc n) zero r = f (suc n) r
  where
    f : ∀{A} → ∀ n → Fin (n * 0) → A
    f zero ()
    f (suc n) r = f n r
fin-pair-iso2 (suc n) (suc m) r = fin-prod-iso2 n m r

encode : (l : List ℕ) → Record l → Fin (L.Π l)
encode [] _ = zero
encode (a ∷ as) r = fin-pair (proj₁ r , (encode as (proj₂ r)))

decode : (l : List ℕ) → Fin (L.Π l) → Record l
decode [] _ = tt
decode (a ∷ as) x = Prod.map id (decode as) (fin-unpair x)

record-lookup : (l : List ℕ) → (r : Record l) → Σ ℕ (\ i → Fin ([ id , const 0 ]' (L.lookup l i)))
              → Bool
record-lookup [] r (i , x) = false
record-lookup (a ∷ l) (x , y) (zero , x₁) = toℕ x == toℕ x₁
record-lookup (a ∷ l) (x , y) (suc i , x₁) = record-lookup l y (i , x₁)

embed : (l : List ℕ) → Σ ℕ (\ i → Fin ([ id , const 0 ]' (L.lookup l i))) → Fin (L.Σ l)
embed [] (x , y) = y
embed (.(suc n) ∷ l) (zero , zero {n}) = zero
embed (.(suc n) ∷ l) (zero , suc {n} y) = suc (embed (n ∷ l) (zero , y))
```

```
embed (zero ∷ l) (suc x , y) = embed l (x , y)
embed (suc a ∷ l) (suc x , y) = suc (embed (a ∷ l) (suc x , y))

extract : (l : List ℕ) → Fin (L.Σ l) → Σ ℕ (\ i → Fin ([ id , const 0 ]′ (L.lookup l i)))
extract [] ()
extract (zero ∷ l) x = Prod.map suc id (extract l x)
extract (suc a ∷ l) zero = 0 , zero
extract (suc a ∷ l) (suc x)
  = Prod.map id (λ {n} → assembleℕ (λ k → Fin ([ id , const 0 ]′ (L.lookup (a ∷ l) k)))
                                   (λ k → Fin ([ id , const 0 ]′ (L.lookup (suc a ∷ l) k))) suc id {n})
             (extract (a ∷ l) x)

embed-iso1 : (l : List ℕ) → (π : Σ ℕ (\ i → Fin ([ id , const 0 ]′ (L.lookup l i))))
          → extract l (embed l π) ≡ π
embed-iso1 [] (x , ())
embed-iso1 (zero ∷ l) (zero , ())
embed-iso1 (suc a ∷ l) (zero , zero) = refl
embed-iso1 (suc a ∷ l) (zero , suc i)
  = cong (Prod.map id (λ {n} → assembleℕ (λ k' → Fin ([ id , const 0 ]′ (L.lookup (a ∷ l) k')))
                                         (λ k' → Fin ([ id , const 0 ]′ (L.lookup (suc a ∷ l) k')))
                                         suc id {n}))
         (embed-iso1 (a ∷ l) (zero , i))
embed-iso1 (zero ∷ l) (suc n , i) = cong (Prod.map suc id) (embed-iso1 l (n , i))
embed-iso1 (suc a ∷ l) (suc n , i)
  = cong (Prod.map id (λ {m} → assembleℕ (λ k' → Fin ([ id , const 0 ]′ (L.lookup (a ∷ l) k')))
                                         (λ k' → Fin ([ id , const 0 ]′ (L.lookup (suc a ∷ l) k')))
                                         suc id {m}))
         (embed-iso1 (a ∷ l) (suc n , i))

embed-iso2 : (l : List ℕ) → (x : Fin (L.Σ l)) → embed l (extract l x) ≡ x
embed-iso2 [] ()
embed-iso2 (zero ∷ l) x = subst (λ k → embed l k ≡ x) refl (embed-iso2 l x)
embed-iso2 (suc a ∷ l) zero = refl
embed-iso2 (suc a ∷ l) (suc i) = subst (λ k → k ≡ suc i) (eq (extract (a ∷ l) i)) ih'
  where
    ih = embed-iso2 (a ∷ l) i

    ih' = cong Fin.suc ih

    eq : (π : Σ ℕ (λ i' → Fin ([ id , const 0 ]′ (L.lookup (a ∷ l) i'))))
       → suc (embed (a ∷ l) π) ≡
           embed (suc a ∷ l) (Prod.map id (λ {n} → assembleℕ
            (λ k → Fin ([ id , const 0 ]′ (L.lookup (a ∷ l) k)))
            (λ k → Fin ([ id , const 0 ]′ (L.lookup (suc a ∷ l) k))) suc
            id {n}) π)
    eq (zero , y) = refl
    eq (suc n , y) = refl

fin-record-iso1 : (l : List ℕ) → (r : Record l) → decode l (encode l r) ≡ r
fin-record-iso1 [] tt = refl
fin-record-iso1 (a ∷ l) (x , y) = subst (λ k → Prod.map id (decode l) k ≡ (x , y))
                                        (sym (fin-pair-iso1 _ _ (x , encode l y)))
                                        (cong (λ k → x , k) (fin-record-iso1 l y))

fin-record-iso2 : (l : List ℕ) → (r : Fin (L.Π l)) → encode l (decode l r) ≡ r
fin-record-iso2 [] zero = refl
fin-record-iso2 [] (suc ())
fin-record-iso2 (a ∷ l) r = subst (λ k → (fin-pair {a} (x , encode l (decode l y))) ≡ k)
                                  (fin-pair-iso2 a (L.Π l) r)
                                  (cong (\ k → fin-pair (x , k)) (fin-record-iso2 (l) y))
  where
    x = proj₁ (fin-unpair {a} r)
    y = proj₂ (fin-unpair {a} r)
```

```
module CTL.RecordSystem where

-- symbolic ctl
open import CTL.TransitionSystem
open import CTL.Definition
open import CTL.DecProc

open import Data.List as List
open import Data.Nat hiding (_<_)
import Data.List.Util as L
open import Data.Sum as Sum
open import Data.Product as Prod
open import Data.Bool using (T;Bool)
open import Data.Fin.Record
open import Data.Fin.Arithmetic
open import Data.Fin hiding (_<_)

open import PropIso

open import Relation.Binary.PropositionalEquality

open import Coinduction

-- finite state machine defined over finite records
record FSMʳ : Set where
  constructor
    frm
  field
    state      : List ℕ
    action     : Record state → List ℕ
    initial    : List (Record state)
    transition : (s : Record state) → (a : Record (action s)) → Record state

  State : Set
  State = Record state

  Action : State → Set
  Action s = Record (action s)

open FSMʳ

toState : (M : FSMʳ) → State M → Fin (L.Π (state M))
toState M = encode (state M)

fromState : (M : FSMʳ) → Fin (L.Π (state M)) → State M
fromState M = decode (state M)

toAction : (M : FSMʳ) → (s : State M) → Action M s → Fin (L.Π (action M s))
toAction M s = encode (action M s)

fromAction : (M : FSMʳ) → (s : State M)
                        → Fin (L.Π (action M s))
                        → Action M s
fromAction M s = decode (action M s)


toFSM : FSMʳ → FSM
toFSM M
  = fsm (L.Π (state M))
        (L.Σ (state M))
        (λ x → L.Π (action M (fromState M x)))
        (List.map (toState M) (initial M))
        (λ s a → toState M (transition M (fromState M s)
                           (fromAction M (fromState M s) a)))
        (λ s ap → record-lookup (state M) (fromState M s)
                                (extract (state M) ap))

symlookup : ∀ l ap → ℕ
symlookup l ap = [ id , const 0 ]′ (L.lookup l ap)

{- CTL formula -}
data CTLʳ (l : List ℕ) : Set where
  false : CTLʳ l
  _∨_ _∧_ E[_U_] : (φ : CTLʳ l) → (ψ : CTLʳ l) → CTLʳ l
  P[_==_] : (ap : ℕ) → (v : Fin (symlookup l ap)) → CTLʳ l
  ~ EX EG : (φ : CTLʳ l) → CTLʳ l

data CTLProblemʳ : Set where
  _,_⊧ʳ_ : (M : FSMʳ) → State M → CTLʳ (state M) → CTLProblemʳ

toCTL : {l : List ℕ} → CTLʳ l → CTL (L.Σ l)
toCTL false          = false
```

```
toCTL (~ φ)           = ~ (toCTL φ)
toCTL (φ ∨ ψ)         = toCTL φ ∨ toCTL ψ
toCTL (φ ∧ ψ)         = toCTL φ ∧ toCTL ψ
toCTL {l} (P[ ap == v ]) = P[ embed l (ap , v) ]
toCTL (EX φ)          = EX (toCTL φ)
toCTL (EG φ)          = EG (toCTL φ)
toCTL E[ φ U ψ ]      = E[ toCTL φ U toCTL ψ ]

toCTLProblem : CTLProblemʳ → CTLProblem
toCTLProblem (M , s ⊨ʳ φ) = toFSM M , toState M s ⊨ toCTL φ

data Runʳ (M : FSMʳ) : State M → Set where
  next : (s : State M) → (s' : Action M s) → ∞ (Runʳ M (transition M s s')) → Runʳ M s

run-decompʳ : ∀ M {s} → Runʳ M s → Σ[ x ∶ Action M s ] (Runʳ M (transition M s x))
run-decompʳ ts {s} (next .s s' y) = s' , ♭ y

run-headʳ : (M : FSMʳ) → {s : State M} → Runʳ M s → Action M s
run-headʳ M r = proj₁ (run-decompʳ M r)

run-tailʳ : ∀ M {s} → (r : Runʳ M s) → Runʳ M (transition M s (run-headʳ M r))
run-tailʳ M r = proj₂ (run-decompʳ M r)

fromRun' : ∀ M {s t} → (fromState M (toState M t) ≡ s) → Run (toFSM M) (toState M t) → Runʳ M s
fromRun' M {._} {t} refl r
  = next (fromState M (toState M t))
         (fromAction M (fromState M (toState M t)) (run-head (toFSM M) r))
         (♯ fromRun' M (fin-record-iso1 (state M) _) (run-tail (toFSM M) r))

fromRun : (M : FSMʳ) → {s : State M} → Run (toFSM M) (toState M s) → Runʳ M s
fromRun M {s} r = fromRun' M (fin-record-iso1 (state M) s) r

toRun' : ∀ M {s t} → (fromState M(toState M t) ≡ s) → Runʳ M s → Run (toFSM M) (toState M t)
toRun' M {._} {t} refl r = next (toAction M (fromState M (toState M t)) (run-headʳ M r))
                                (♯ toRun' M {_} {_} q (run-tailʳ M r))
  where
    p = fin-record-iso1 (state M) (transition M (fromState M (toState M t)) (run-headʳ M r))

    q = subst (λ k → fromState M (toState M (transition M (fromState M (toState M t)) k))
                   ≡ transition M (fromState M (toState M t)) (run-headʳ M r))
              (sym (fin-record-iso1 (action M (fromState M (toState M t))) (run-headʳ M r))) p

toRun : (M : FSMʳ) → {s : State M} → Runʳ M s → Run (toFSM M) (toState M s)
toRun M {s} r = toRun' M (fin-record-iso1 (state M) s) r

lem-eq-id : (l : List ℕ) → (s t : Record l) → (decode l (encode l s)) ≡ t → s ≡ t
lem-eq-id l t ._ refl = sym (fin-record-iso1 l t)

nthʳ : (M : FSMʳ) → {s : State M} → (n : ℕ) → Runʳ M s → State M
nthʳ M {s} zero r = s
nthʳ M (suc n) r = nthʳ M n (run-tailʳ M r)

lem-nth''ʳ : (M : FSMʳ)
         → {s t : State M}
         → (r : Runʳ M s)
         → (n : ℕ)
         → (eq : fromState M (toState M t) ≡ s)
         → nthʳ M n r ≡ fromState M (nth (toFSM M) n (toRun' M {s} {t} eq r))
lem-nth''ʳ M r zero refl = refl
lem-nth''ʳ M {._} {t} (next ._ s' y) (suc n) refl = lem-nth''ʳ M (♭ y) n _

lem-nthʳ : (M : FSMʳ)
         → {s : State M}
         → (r : Runʳ M s)
         → (n : ℕ)
         → nthʳ M n r ≡ fromState M (nth (toFSM M) n (toRun M r))
lem-nthʳ M {s} r n = lem-nth''ʳ M {s} {s} r n (fin-record-iso1 (state M) s)

lem-nth' : (M : FSMʳ)
         → {s t : State M}
         → (r : Runʳ M t)
         → (n : ℕ)
         → (eq : fromState M (toState M s) ≡ t)
         → toState M (nthʳ M n r) ≡ nth (toFSM M) n (toRun' M {t} {s} eq r)
lem-nth' M r zero refl = cong (toState M) (fin-record-iso1 (state M) _)
lem-nth' M {s} (next ._ s' y) (suc n) refl = lem-nth' M (♭ y) n _

lem-nth : (M : FSMʳ)
         → {s : State M}
         → (r : Runʳ M s)
         → (n : ℕ)
         → toState M (nthʳ M n r) ≡ nth (toFSM M) n (toRun M r)
lem-nth M r n = lem-nth' M r n (fin-record-iso1 (state M) _)
```

```
lem-nth'''' : (M : FSMʳ)
            → {s t : State M}
            → (r : Run (toFSM M) (toState M t))
            → (n : ℕ)
            → (eq : fromState M (toState M t) ≡ s)
            → nthʳ M n (fromRun' M eq r) ≡ fromState M (nth (toFSM M) n r)
lem-nth'''' M r zero refl = refl
lem-nth'''' M r (suc n) refl = lem-nth'''' M (run-tail (toFSM M) r) n _

lem-nth''' : (M : FSMʳ)
           → {s : State M}
           → (r : Run (toFSM M) (toState M s))
           → (n : ℕ)
           → nthʳ M n (fromRun M r) ≡ fromState M (nth (toFSM M) n r)
lem-nth''' M r n = lem-nth'''' M r n _

lem-nth'' : (M : FSMʳ)
          → {s t : State M}
          → (r : Run (toFSM M) (toState M t))
          → (n : ℕ)
          → (eq : fromState M (toState M t) ≡ s)
          → toState M (nthʳ M n (fromRun' M eq r)) ≡ nth (toFSM M) n r
lem-nth'' M r zero refl = cong (λ k → toState M k) (fin-record-iso1 (state M) _)
lem-nth'' M r (suc n) refl = lem-nth'' M (run-tail (toFSM M) r) n (fin-record-iso1 (state M) _)

lem-nth''' : ∀ M {s} → (r : Run (toFSM M) (toState M s)) → (n : ℕ)
           → toState M (nthʳ M n (fromRun M r)) ≡ nth (toFSM M) n r
lem-nth''' M r n = lem-nth'' M r n (fin-record-iso1 (state M) _)

⟦_⟧ʳ : CTLProblemʳ → Set
⟦ M , s ⊧ʳ false ⟧ʳ          = ⊥
⟦ M , s ⊧ʳ ~ φ ⟧ʳ            = ¬ ⟦ M , s ⊧ʳ φ ⟧ʳ
⟦ M , s ⊧ʳ (φ ∨ ψ) ⟧ʳ        = (⟦ M , s ⊧ʳ φ ⟧ʳ) ⊎ (⟦ M , s ⊧ʳ ψ ⟧ʳ)
⟦ M , s ⊧ʳ (φ ∧ ψ) ⟧ʳ        = (⟦ M , s ⊧ʳ φ ⟧ʳ) × (⟦ M , s ⊧ʳ ψ ⟧ʳ)
⟦ M , s ⊧ʳ (P[ ap == v ]) ⟧ʳ = T (record-lookup (state M) s (ap , v))
⟦ M , s ⊧ʳ EX φ ⟧ʳ           = Σ[ run ∶ Runʳ M s ] ⟦ M , nthʳ M 1 run ⊧ʳ φ ⟧ʳ
⟦ M , s ⊧ʳ EG φ ⟧ʳ           = Σ[ run ∶ Runʳ M s ] (∀ (n : ℕ) → ⟦ M , nthʳ M n run ⊧ʳ φ ⟧ʳ)
⟦ M , s ⊧ʳ E[ φ U ψ ] ⟧ʳ     = Σ[ run ∶ Runʳ M s ] Σ[ k ∶ ℕ ]
                                  ((∀ (j : ℕ) → T(j < k) → ⟦ M , nthʳ M j run ⊧ʳ φ ⟧ʳ)
                                  × ⟦ M , nthʳ M k run ⊧ʳ ψ ⟧ʳ)

mutual
  correct1ʳ : (P : CTLProblemʳ) → ⟦ P ⟧ʳ → ⟦ toCTLProblem P ⟧
  correct1ʳ (M , s ⊧ʳ false) p    = p
  correct1ʳ (M , s ⊧ʳ ~ φ) p      = λ x → p (correct2ʳ (M , s ⊧ʳ φ) x)
  correct1ʳ (M , s ⊧ʳ (φ ∨ ψ)) p = Sum.map (correct1ʳ (M , s ⊧ʳ φ)) (correct1ʳ (M , s ⊧ʳ ψ)) p
  correct1ʳ (M , s ⊧ʳ (φ ∧ ψ)) p = Prod.map (correct1ʳ (M , s ⊧ʳ φ)) (correct1ʳ (M , s ⊧ʳ ψ)) p
  correct1ʳ (M , s ⊧ʳ P[ ap == v ]) p rewrite fin-record-iso1 (state M) s
                                           | embed-iso1 (state M) (ap , v) = p
  correct1ʳ (M , s ⊧ʳ EX φ) p
    = Prod.map (toRun M) (λ {r} x → subst (λ k → ⟦ toFSM M , toState M k ⊧ toCTL φ ⟧)
                                          (trans (lem-nthʳ M r 1)
                                                 (fin-record-iso1 (state M) _))
                                          (correct1ʳ (M , _ ⊧ʳ φ) x)) p
  correct1ʳ (M , s ⊧ʳ EG φ) p
    = Prod.map (toRun M) (λ {r} egp n → subst (λ k → ⟦ toFSM M , k ⊧ toCTL φ ⟧)
                                              (trans (cong (toState M) (lem-nthʳ M r n))
                                                     (fin-record-iso2 (state M) _))
                                              (correct1ʳ (M , nthʳ M n r ⊧ʳ φ) (egp n))) p
  correct1ʳ (M , s ⊧ʳ E[ φ U ψ ]) p
    = Prod.map (toRun M) (λ {r} → Prod.map id (λ {n} → Prod.map (λ eu j jp →
                subst (λ k → ⟦ toFSM M , k ⊧ toCTL φ ⟧) (lem-nth M r j)
                      (correct1ʳ (M , (nthʳ M j r) ⊧ʳ φ) (eu j jp)))
                (λ kp → subst (λ k → ⟦ toFSM M , k ⊧ toCTL ψ ⟧)
                              (lem-nth M r n)
                              (correct1ʳ (M , (nthʳ M n r) ⊧ʳ ψ) kp)))) p

  correct2ʳ : (P : CTLProblemʳ) → ⟦ toCTLProblem P ⟧ → ⟦ P ⟧ʳ
  correct2ʳ (M , s ⊧ʳ false) ()
  correct2ʳ (M , s ⊧ʳ ~ φ) p      = λ x → p (correct1ʳ (M , s ⊧ʳ φ) x)
  correct2ʳ (M , s ⊧ʳ (φ ∨ ψ)) p = Sum.map (correct2ʳ (M , s ⊧ʳ φ)) (correct2ʳ (M , s ⊧ʳ ψ)) p
  correct2ʳ (M , s ⊧ʳ (φ ∧ ψ)) p = Prod.map (correct2ʳ (M , s ⊧ʳ φ)) (correct2ʳ (M , s ⊧ʳ ψ)) p
  correct2ʳ (M , s ⊧ʳ P[ ap == v ]) p rewrite fin-record-iso1 (state M) s
                                           | embed-iso1 (state M) (ap , v) = p
  correct2ʳ (M , s ⊧ʳ EX φ) p
    = Prod.map (fromRun M) (λ {r} exp → subst (λ k → ⟦ M , k ⊧ʳ φ ⟧ʳ)
                                              (trans (sym (fin-record-iso1 (state M) _))
                                                     (sym (lem-nthʳ'' M r 1)))
                                              (correct2ʳ (M , _ ⊧ʳ φ) exp)) p
  correct2ʳ (M , s ⊧ʳ EG φ) p
    = Prod.map (fromRun M) (λ {r} egp n → subst (λ k → ⟦ M , k ⊧ʳ φ ⟧ʳ)
                                                (sym (lem-nthʳ'' M r n))
```

```
                                                (correct2ʳ (M , _ ⊧ʳ φ)
                                                   (subst (λ k → ⟦ toFSM M , k ⊧ toCTL φ ⟧)
                                                          (sym (fin-record-iso2 (state M) _))
                                                          (egp n)))) p
  correct2ʳ (M , s ⊧ʳ E[ φ U ψ ]) p = Prod.map (fromRun M)
                                                (λ {r} → Prod.map id (\ {n} → Prod.map
          (λ eup j jp → correct2ʳ (M , _ ⊧ʳ φ)
                                  (subst (λ k → ⟦ toFSM M , k ⊧ toCTL φ ⟧)
                                         (sym (lem-nth''' M r j)) (eup j jp)))
          (λ kp → correct2ʳ (M , _ ⊧ʳ ψ)
                            (subst (λ k → ⟦ toFSM M , k ⊧ toCTL ψ ⟧)
                                   (sym (lem-nth''' M r n)) kp)))) p

evalʳ : CTLProblemʳ → Bool
evalʳ P = eval (toCTLProblem P)

open import CTL.Proof

soundnessʳ : (P : CTLProblemʳ) → (T (evalʳ P)) → ⟦ P ⟧ʳ
soundnessʳ P p = correct2ʳ P (soundness (toCTLProblem P) p)

completenessʳ : (P : CTLProblemʳ) → ⟦ P ⟧ʳ → (T (evalʳ P))
completenessʳ P p = complete (toCTLProblem P) (correct1ʳ P p)
```

```
module Proof.Util where

open import Data.List hiding ([_])
open import Data.Bool
open import Data.Sum as Sum
open import Data.Product as Prod

open import PropIso

open import Relation.Binary.PropositionalEquality hiding ([_])

open import Boolean.Formula

_∈_ : PL-Formula → List PL-Formula → Bool
φ ∈ [] = false
φ ∈ (ψ :: Γ) = φ ≡pl ψ ∨ φ ∈ Γ

_∪_ : List PL-Formula → List PL-Formula → List PL-Formula
_∪_ = _++_

_⊆_ : List PL-Formula → List PL-Formula → Bool
[] ⊆ Γ' = true
(γ :: Γ) ⊆ Γ' = γ ∈ Γ' ∧ Γ ⊆ Γ'

_|_ : List PL-Formula → PL-Formula → List PL-Formula
[] | _ = []
(φ :: φs) | ψ with φ ≡pl ψ
...| true  = φs | ψ
...| false = φ :: (φs | ψ)

_⊇≈_ : List PL-Formula → List PL-Formula → Set
Γ₁ ⊇≈ Γ₂ = ∀ γ → T (γ ∈ Γ₂) → T (γ ∈ Γ₁)

lift-⊆ : ∀ Γ₁ Γ₂ → T (Γ₁ ⊆ Γ₂) → (Γ₂ ⊇≈ Γ₁)
lift-⊆ [] Γ₂ p         = λ _ ()
lift-⊆ (γ₁ :: Γ₁) Γ₂ p = λ γ → ∨-elim (λ γ=γ₁ → subst (λ x → T (x ∈ Γ₂))
                                                    (sym (lift-≡pl γ γ₁ γ=γ₁)) (∧-eliml p))
                                  (lift-⊆ Γ₁ Γ₂ (∧-elimr (γ₁ ∈ Γ₂) p) γ)

lem-seq-restrict-foldr : ∀ ξ Γ φ → ⟦ ξ ⊨ andpl Γ ⟧pl → ⟦ ξ ⊨ andpl (Γ | φ) ⟧pl
lem-seq-restrict-foldr ξ []       φ p = tt
lem-seq-restrict-foldr ξ (γ :: Γ) φ p with γ ≡pl φ
...| true = lem-seq-restrict-foldr ξ Γ φ (proj₂ p)
...| false = Prod.map id (lem-seq-restrict-foldr ξ Γ φ) p

lem-seq-restrict-foldr' : ∀ ξ Γ φ → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ andpl (Γ | φ) ⟧pl → ⟦ ξ ⊨ andpl Γ ⟧pl
lem-seq-restrict-foldr' ξ []       φ p q = tt
lem-seq-restrict-foldr' ξ (γ :: Γ) φ p q with ex-mid (γ ≡pl φ)
...| inj₁ x rewrite Tb x = subst (⟦_⊨_⟧pl ξ) (sym (lift-≡pl γ φ x)) p
                                                , (lem-seq-restrict-foldr' ξ Γ φ p q)
...| inj₂ x rewrite ¬Tb x = Prod.map id (lem-seq-restrict-foldr' ξ Γ φ p) q

lem-seq : ∀ Γ₁ Γ₂ φ → Γ₁ ⊇≈ (φ :: Γ₂) → Γ₁ ⊇≈ Γ₂
lem-seq Γ₁ Γ₂ φ f = λ γ x → f γ (∨-intror (γ ≡pl φ) (γ ∈ Γ₂) x)

lem-seq-atom : ∀ ξ Γ φ → ⟦ ξ ⊨ andpl Γ ⟧pl → T (φ ∈ Γ) → ⟦ ξ ⊨ φ ⟧pl
lem-seq-atom ξ [] φ conj   = λ ()
lem-seq-atom ξ (γ :: Γ) φ c = ∨-elim (λ φ=γ → subst (⟦_⊨_⟧pl ξ) (sym (lift-≡pl φ γ φ=γ)) (proj₁ c))
                                  (lem-seq-atom ξ Γ φ (proj₂ c))

lem-seq-subst-foldr : ∀ ξ Γ₁ Γ₂ → Γ₁ ⊇≈ Γ₂ → ⟦ ξ ⊨ andpl Γ₁ ⟧pl → ⟦ ξ ⊨ andpl Γ₂ ⟧pl
lem-seq-subst-foldr ξ Γ₁ [] y p        = tt
lem-seq-subst-foldr ξ Γ₁ (γ₂ :: Γ₂) y p = lem-seq-atom ξ Γ₁ γ₂ p
                                                (y γ₂ (∨-introl (γ₂ ≡pl γ₂) _ (id-≡pl γ₂)))
                                          , lem-seq-subst-foldr ξ Γ₁ Γ₂ (lem-seq Γ₁ Γ₂ γ₂ y) p

seq-split : ∀ ξ Γ₁ Γ₂ → ⟦ ξ ⊨ andpl (Γ₁ ++ Γ₂) ⟧pl → ⟦ ξ ⊨ andpl Γ₁ ⟧pl × ⟦ ξ ⊨ andpl Γ₂ ⟧pl
seq-split ξ []        Γ₂ p++ = tt , p++
seq-split ξ (γ₁ :: Γ₁) Γ₂ p++ = Prod.map (λ x → (proj₁ p++) , x) id (seq-split ξ Γ₁ Γ₂ (proj₂ p++))

record [_⇒_] (A B : Set) : Set where
  constructor _⇒_
  field πΓ : List A
        πφ : B

open [_⇒_] public

mkconj : PL-Formula → List PL-Formula → PL-Formula
mkconj γ []       = γ
mkconj γ (x :: xs) = γ && mkconj x xs

⟦_⊨_⟧⇒ : Env → [ PL-Formula ⇒ PL-Formula ] → Set
⟦ ξ ⊨ [] ⇒ φ ⟧⇒       = ⟦ ξ ⊨ φ ⟧pl
⟦ ξ ⊨ (x :: xs) ⇒ φ ⟧⇒ = ⟦ ξ ⊨ mkconj x xs ⟧pl → ⟦ ξ ⊨ φ ⟧pl
```

```
module Proof.List where

open import Data.Nat hiding (_<_)
open import Data.Vec as Vec using (Vec;lookup;_∷_;[];zip;map)
open import Data.Fin hiding (_<_)
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.List as List
open import Data.Bool

open import Relation.Binary.PropositionalEquality hiding ([_])

open import Algebra

open import PropIso

data Vec* {A : Set} (F : A → Set) : {n : ℕ} → Vec A n → Set where
  []  : Vec* F []
  _∷_ : {n : ℕ} {a : A} {v : Vec A n} (x : F a) (xs : Vec* F v) → Vec* F (a ∷ v)

infixr 5 _∷_

data List* {A : Set} (F : A → Set) : List A → Set where
  [] : List* F []
  _∷_ : {a : A} {l : List A} (x : F a) (xs : List* F l) → List* F (a ∷ l)


_++*_ : {A : Set} → {F : A → Set} → {l1 l2 : List A} → List* F l1 → List* F l2 → List* F (l1 ++ l2)
[] ++* m = m
(x ∷ l₁) ++* m = x ∷ l₁ ++* m

_vlt_ : ∀ {n} → Vec ℕ n → ℕ → Bool
[] vlt _ = true
(n ∷ ns) vlt m = n < m ∧ ns vlt m

lookup' : {A : Set} → (l : List A) → (n : ℕ) → T (n < length l) → A
lookup' [] n ()
lookup' (x ∷ l) zero p = x
lookup' (x ∷ l) (suc n) p = lookup' l n p

lookup* : ∀ {A l F} → List* {A} F l → (n : ℕ) → (p : T (n < length l)) → F (lookup' l n p)
lookup* [] n ()
lookup* (x ∷ l) zero p = x
lookup* (x ∷ l) (suc n) p = lookup* l n p

lastnode : {A : Set} → (l : List A) → T (not (null l)) → A
lastnode [] ()
lastnode (n ∷ []) p = n
lastnode (n ∷ x ∷ ns) p = lastnode (x ∷ ns) tt

lastnode* : ∀ {A l F} → List* {A} F l → (q : T (not (null l))) → F (lastnode l q)
lastnode* [] ()
lastnode* (n ∷ []) p = n
lastnode* (n ∷ x ∷ ns) p = lastnode* (x ∷ ns) tt

record RuleSystem (Δ Φ Ξ : Set) (⟦_⊧_⟧ : Ξ → Φ → Set) : Set₁ where
  field
    arity : Δ → ℕ
    correct : (k : Δ) → Vec Φ (arity k) → Φ → Bool
    sound : (k : Δ) → (seq : Vec Φ (arity k))
                    → (conc : Φ)
                    → T (correct k seq conc)
                    → Vec* (λ φ → ∀ ξ → ⟦ ξ ⊧ φ ⟧) seq
                    → ∀ ξ → ⟦ ξ ⊧ conc ⟧

  record ProofNode : Set where
    constructor
      node
    field
      rule : Δ
      conc : Φ
      seq : Vec ℕ (arity rule)

  open ProofNode

  ProofList : Set
  ProofList = List ProofNode

  hypothesis : (l : ProofList) → ∀ {n} → (v : Vec ℕ n) → T (v vlt (length l)) → Vec ProofNode n
  hypothesis l [] p = []
  hypothesis l (x ∷ v) p = lookup' l x (∧-eliml p) ∷ (hypothesis l v (∧-elimr (x < length l) p))

  correct-rule : ProofList → ProofNode → Bool
  correct-rule l n with ex-mid (seq n vlt length l)
  ...| inj₁ x = correct (rule n) (Vec.map conc (hypothesis l (seq n) x)) (conc n)
  ...| inj₂ x = false
```

```
  correct-list' : ProofList → ProofList → Bool
  correct-list' done [] = true
  correct-list' done (n ∷ todo) = correct-rule done n ∧ correct-list' (done ++ [ n ]) todo

  ProvedList : (l : ProofList) → Set
  ProvedList = List* (\ n → ∀ ξ → ⟦ ξ ⊨ conc n ⟧)
  hypothesis' : (l : ProofList) → ∀ {n} → (seq : Vec ℕ n)
              → ProvedList l
              → (x : T (seq vlt (length l)))
              → Vec* (λ φ → ∀ ξ → ⟦ ξ ⊨ φ ⟧) (Vec.map conc (hypothesis l seq x))
  hypothesis' l [] p q = []
  hypothesis' l (x ∷ s) p q = (lookup* p x (∧-eliml q))
                              ∷ (hypothesis' l s p (∧-elimr (x < length l) q))

  sound-rule : ∀ l n → (p : T (correct-rule l n)) → (q : ProvedList l) → ∀ ξ → ⟦ ξ ⊨ conc n ⟧
  sound-rule l n p q with ex-mid (seq n vlt foldr (λ _ → suc) 0 l)
  sound-rule l n p q | inj₁ x = sound (rule n) (Vec.map conc (hypothesis l (seq n) x))
                                      (conc n) p (hypothesis' l (seq n) q x)
  sound-rule l n p q | inj₂ y = ⊥-elim p

  sound-list' : (l1 : ProofList)
              → (l2 : ProofList)
              → (p : T (correct-list' l1 l2))
              → (q : ProvedList l1)
              → ProvedList (l1 ++ l2)
  sound-list' l1 [] p q = q ++* []
  sound-list' l1 (n ∷ l2) p q rewrite sym (Monoid.assoc (monoid ProofNode) l1 [ n ] l2)
    = sound-list' (l1 ++ [ n ]) l2 (∧-elimr (correct-rule l1 n) p)
                  (q ++* (sound-rule l1 n (∧-eliml p) q ∷ []))

  correct-list : (l : ProofList) → Bool
  correct-list l = correct-list' [] l

  sound-list : (l : ProofList)
             → (x : T (not (null l)))
             → (p : T (correct-list l))
             → ∀ ξ → ⟦ ξ ⊨ conc (lastnode l x) ⟧
  sound-list l x p = lastnode* (sound-list' [] l p []) x

open RuleSystem public

open import Data.Maybe

{-# BUILTIN MAYBE Maybe #-}
{-# BUILTIN JUST just #-}
{-# BUILTIN NOTHING nothing #-}
```

```
module Proof.PropLogic where

open import Data.List hiding ([_])
open import Data.Nat
open import Data.Bool
open import Data.Sum as Sum
open import Data.Product as Prod
open import Data.Vec using (Vec;_∷_;[])

open import PropIso

open import Relation.Binary.PropositionalEquality hiding ([_])

open import Function

open import Proof.List
open import Proof.Util

open import Boolean.Formula

data PropositionalRule : Set where
  ∧+ →- efq raa ax : PropositionalRule
  ∧l- ∧r- ∨l+ ∨r+ →+ : {φ : PL-Formula} → PropositionalRule
  ∨- : {φ ψ : PL-Formula} → PropositionalRule

proparity : PropositionalRule → ℕ
proparity ∧+  = 2 -- and intro
proparity ∧l- = 1 -- and elim l
proparity ∧r- = 1 -- and elim r
proparity →+  = 1 -- imp intro
proparity →-  = 2 -- imp elim
proparity ∨l+ = 1 -- or intro l
proparity ∨r+ = 1 -- or intro r
proparity ∨-  = 3 -- or elim
proparity efq = 1 -- efq
proparity raa = 1 -- raa
proparity ax  = 0 -- axiom

propcorrect : (k : PropositionalRule) → Vec [ PL-Formula ⇒ PL-Formula ] (proparity k)
           → [ PL-Formula ⇒ PL-Formula ] → Bool
propcorrect ∧+ (Γ₁ ⇒ φ₁ ∷ Γ₂ ⇒ φ₂ ∷ []) (Γ₃ ⇒ φ₃) = (φ₃ ≡pl (φ₁ && φ₂)) ∧ ((Γ₁ ∪ Γ₂) ⊆ Γ₃)
propcorrect (∧l- {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂)    = (φ₁ ≡pl (φ₂ && φ₃)) ∧ (Γ₁ ⊆ Γ₂)
propcorrect (∧r- {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂)    = (φ₁ ≡pl (φ₃ && φ₂)) ∧ (Γ₁ ⊆ Γ₂)
propcorrect (→+ {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂)     = (φ₂ ≡pl (φ₃ => φ₁)) ∧ ((Γ₁ - φ₃) ⊆ Γ₂)
propcorrect →- (Γ₁ ⇒ φ₁ ∷ Γ₂ ⇒ φ₂ ∷ []) (Γ₃ ⇒ φ₃) = (φ₁ ≡pl (φ₂ => φ₃)) ∧ ((Γ₁ ∪ Γ₂) ⊆ Γ₃)
propcorrect (∨l+ {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂)    = (φ₂ ≡pl (φ₁ || φ₃)) ∧ (Γ₁ ⊆ Γ₂)
propcorrect (∨r+ {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂)    = (φ₂ ≡pl (φ₃ || φ₁)) ∧ (Γ₁ ⊆ Γ₂)
propcorrect (∨- {φ₅} {φ₆}) (Γ₁ ⇒ φ₁ ∷ Γ₂ ⇒ φ₂ ∷ Γ₃ ⇒ φ₃ ∷ []) (Γ₄ ⇒ φ₄)
  = (φ₂ ≡pl φ₃) ∧ (φ₂ ≡pl φ₄) ∧ (φ₁ ≡pl (φ₅ || φ₆)) ∧ ((Γ₁ ∪ ((Γ₂ - φ₅) ∪ (Γ₃ - φ₆))) ⊆ Γ₄)
propcorrect efq (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂)           = (φ₁ ≡pl ¥false) ∧ (Γ₁ ⊆ Γ₂)
propcorrect raa (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂)           = (φ₁ ≡pl ¥false) ∧ ((Γ₁ - ~ φ₂) ⊆ Γ₂)
propcorrect ax [] (Γ ⇒ φ) = φ ∈ Γ

propsound : (k : PropositionalRule) → (seq : Vec [ PL-Formula ⇒ PL-Formula ] (proparity k))
          → (conc : [ PL-Formula ⇒ PL-Formula ]) → T (propcorrect k seq conc)
          → Vec* (λ x → ∀ ξ → ⟦ ξ ⊨ andpl (πΓ x) => πφ x ⟧pl) seq
          → ∀ ξ → ⟦ ξ ⊨ andpl (πΓ conc) => πφ conc ⟧pl
propsound ∧+ (Γ₁ ⇒ φ₁ ∷ Γ₂ ⇒ φ₂ ∷ []) (Γ₃ ⇒ φ₃) p (q1 ∷ q2 ∷ []) ξ hyp
  rewrite lift-≡pl φ₃ _ (∧-eliml p)
  = Prod.map (q1 ξ) (q2 ξ) (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ++ Γ₂)
                    (lift-⊆ (Γ₁ ++ Γ₂) Γ₃ (∧-elimr (φ₃ ≡pl (φ₁ && φ₂)) p)) hyp))
propsound (∧l- {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂) p (q ∷ []) ξ hyp rewrite lift-≡pl φ₁ _ (∧-eliml p)
  = proj₁ (q ξ (lem-seq-subst-foldr ξ Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂ (∧-elimr (φ₁ ≡pl (φ₂ && φ₃)) p)) hyp))
propsound (∧r- {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂) p (q ∷ []) ξ hyp rewrite lift-≡pl φ₁ _ (∧-elimr p)
  = proj₂ (q ξ (lem-seq-subst-foldr ξ Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂ (∧-elimr (φ₁ ≡pl (φ₃ && φ₂)) p)) hyp))
propsound (→+ {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂) p (q ∷ []) ξ hyp rewrite lift-≡pl φ₂ _ (∧-elimr p)
  = λ x → q ξ (lem-seq-restrict-foldr' ξ Γ₁ φ₃ x (lem-seq-subst-foldr ξ Γ₂ (Γ₁ - φ₃)
                                  (lift-⊆ (Γ₁ - φ₃) Γ₂ (∧-elimr (φ₂ ≡pl (φ₃ => φ₁)) p)) hyp))
propsound →- (Γ₁ ⇒ φ₁ ∷ Γ₂ ⇒ φ₂ ∷ []) (Γ₃ ⇒ φ₃) p (q1 ∷ q2 ∷ []) ξ hyp
  rewrite lift-≡pl φ₁ _ (∧-elimr p)
  = let π = seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ++ Γ₂)
                                (lift-⊆ (Γ₁ ++ Γ₂) Γ₃ (∧-elimr (φ₁ ≡pl (φ₂ => φ₃)) p)) hyp)
    in q1 ξ (proj₁ π) (q2 ξ (proj₂ π))
propsound (∨l+ {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂) p (q ∷ []) ξ hyp rewrite lift-≡pl φ₂ _ (∧-elimr p)
  = inj₁ (q ξ (lem-seq-subst-foldr ξ Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂ ((∧-elimr (φ₂ ≡pl (φ₁ || φ₃))) p)) hyp))
propsound (∨r+ {φ₃}) (Γ₁ ⇒ φ₁ ∷ []) (Γ₂ ⇒ φ₂) p (q ∷ []) ξ hyp rewrite lift-≡pl φ₂ _ (∧-elimr p)
  = inj₂ (q ξ (lem-seq-subst-foldr ξ Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂ ((∧-elimr (φ₂ ≡pl (φ₃ || φ₁))) p)) hyp))
propsound (∨- {φ₅} {φ₆}) (Γ₁ ⇒ φ₁ ∷ Γ₂ ⇒ φ₂ ∷ Γ₃ ⇒ φ₃ ∷ []) (Γ₄ ⇒ φ₄) p (q1 ∷ q2 ∷ q3 ∷ []) ξ hyp
  rewrite sym (lift-≡pl φ₂ _ (∧-elimr p))
        | sym (lift-≡pl φ₂ φ₄ ((∧-elimr ∘ ∧-elimr (φ₂ ≡pl φ₃)) p))
        | lift-≡pl φ₁ (φ₅ || φ₆) ((∧-elimr ∘ ∧-elimr (φ₂ ≡pl φ₄) ∘ ∧-elimr (φ₂ ≡pl φ₃)) p)
  = let Γ₄' = lem-seq-subst-foldr ξ Γ₄ (Γ₁ ++ Γ₂ - φ₅ ++ Γ₃ - φ₆) (lift-⊆ (Γ₁ ++ Γ₂ - φ₅ ++ Γ₃ - φ₆)
                Γ₄ ((∧-elimr (φ₁ ≡pl (φ₅ || φ₆)) ∘ ∧-elimr (φ₂ ≡pl φ₄) ∘ ∧-elimr (φ₂ ≡pl φ₃)) p)) hyp
    in [ (λ x → q2 ξ (lem-seq-restrict-foldr' ξ Γ₂ φ₅ x
              (proj₁ (seq-split ξ (Γ₂ - φ₅) _ (proj₂ (seq-split ξ Γ₁ _ Γ₄')))))))
       , (λ x → q3 ξ (lem-seq-restrict-foldr' ξ Γ₃ φ₆ x
              (proj₂ (seq-split ξ (Γ₂ - φ₅) _ (proj₂ (seq-split ξ Γ₁ _ Γ₄')))))))
```

```
        ]' (q1 ξ (proj₁ (seq-split ξ Γ₁ _ Γ₄')))
propsound efq (Γ₁ ⇒ φ₁ :: []) (Γ₂ ⇒ φ₂) p (q :: []) ξ hyp rewrite lift-≡pl φ₁ _ (∧-eliml p)
  = ⊥-elim (q ξ (lem-seq-subst-foldr ξ Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂ (∧-elimr (φ₁ ≡pl ¥false) p)) hyp))
propsound raa (Γ₁ ⇒ φ₁ :: []) (Γ₂ ⇒ φ₂) p (q :: []) ξ hyp rewrite lift-≡pl φ₁ _ (∧-eliml p)
  = stbl-pl ξ φ₂ (λ x → q ξ (lem-seq-restrict-foldr' ξ Γ₁ (~ φ₂) x (lem-seq-subst-foldr ξ Γ₂
           (Γ₁ , ~ φ₂) (lift-⊆ (Γ₁ , (φ₂ => ¥false)) Γ₂ (∧-elimr (φ₁ ≡pl ¥false) p)) hyp)))
propsound ax [] ([] ⇒ φ) () q ξ hyp
propsound ax [] ((γ :: Γ) ⇒ φ) p q ξ hyp = ∨-elim (λ x → subst (⟦ _ ⊢ _ ⟧pl ξ) (sym (lift-≡pl φ _ x))
  (proj₁ hyp)) (λ k → propsound ax [] (Γ ⇒ φ) k [] ξ (proj₂ hyp)) p

proplogic : RuleSystem PropositionalRule [ PL-Formula ⇒ PL-Formula ] Env
                                         (λ ξ x → ⟦ ξ ⊢ andpl (πΓ x) => πφ x ⟧pl)
proplogic = record { arity = proparity; correct = propcorrect; sound = propsound }

private
  module test where
  open RuleSystem

  φ = ¥ 0

  derivation : ProofList proplogic
  derivation = node ax ((φ :: ~ (φ || ~ φ) :: []) ⇒ ~ (φ || ~ φ)) []
             :: node ax ((φ :: ~ (φ || ~ φ) :: []) ⇒ φ) []
             :: node (∨ˡ+ {~ φ}) ((φ :: ~ (φ || ~ φ) :: []) ⇒ (φ || ~ φ)) (1 :: [])
             :: node →- ((φ :: ~ (φ || ~ φ) :: []) ⇒ ¥false) (0 :: 2 :: [])
{-4-}        :: node (→+ {φ}) ((~ (φ || ~ φ) :: []) ⇒ ~ φ) (3 :: [])
             :: node ax ((~ φ :: ~ (φ || ~ φ) :: []) ⇒ ~ (φ || ~ φ)) []
             :: node ax ((~ φ :: ~ (φ || ~ φ) :: []) ⇒ ~ φ) []
             :: node (∨ʳ+ {φ}) ((~ φ :: ~ (φ || ~ φ) :: []) ⇒ (φ || ~ φ)) (6 :: [])
{-8-}        :: node →- ((~ φ :: ~ (φ || ~ φ) :: []) ⇒ ¥false) (5 :: 7 :: [])
             :: node raa ((~ (φ || ~ φ) :: []) ⇒ φ) (8 :: [])
             :: node →- ((~ (φ || ~ φ) :: []) ⇒ ¥false) (4 :: 9 :: [])
             :: node raa ([] ⇒ (φ || ~ φ)) (10 :: [])
             :: []

  p¬p : ∀ ξ → ⟦ ξ ⊢ φ || ~ φ ⟧pl
  p¬p ξ = sound-list proplogic derivation tt tt ξ tt
```

```
module Proof.EProver where

open import Data.Nat
open import Data.Bool
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Unit
open import Data.Empty
open import Data.List hiding ([_])
open import Data.Vec using (Vec;[];_∷_)
open import Data.Maybe

open import Relation.Binary.PropositionalEquality hiding ([_])

open import Boolean.Formula
open import Boolean.PL-Formula.Equivalence
open import Boolean.PL-Formula.Substitute
open import Boolean.PL-Formula.DropEquivalence
open import Boolean.PL-Formula.Distribute
open import Boolean.PL-Formula.RemoveConstants

open import Proof.EProver.PM
open import Proof.EProver.NNF
open import Proof.List
open import Proof.Util

open import PropIso

private
  Formula : Set
  Formula = [ PL-Formula ⇒ PL-Formula ]

{- TO DO: combine refute1 with refute2 -}
mutual
  refute1 : ∀ ξ φ ψ → ⟦ ξ ⊨ ψ ⟧pl → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ φ [ (~ ψ) / ¥false ] ⟧pl
  refute1 ξ φ ψ [ψ] [φ] with ex-mid ((~ ψ) ≡pl φ)
  ...| inj₁ x rewrite Tb x | sym (lift-≡pl (~ ψ) φ x) = [φ] [ψ]
  refute1 ξ ¥true ψ     [ψ] [φ] | inj₂ x = tt
  refute1 ξ ¥false ψ    [ψ] [φ] | inj₂ x = [φ]
  refute1 ξ (y || y') ψ [ψ] [φ] | inj₂ x = Sum.map  (refute1 ξ y ψ [ψ]) (refute1 ξ y' ψ [ψ]) [φ]
  refute1 ξ (y && y') ψ [ψ] [φ] | inj₂ x = Prod.map (refute1 ξ y ψ [ψ]) (refute1 ξ y' ψ [ψ]) [φ]
  refute1 ξ (y => y') ψ [ψ] [φ] | inj₂ x rewrite ¬Tb x
    = (refute1 ξ y' ψ [ψ]) ∘ [φ] ∘ (refute1' ξ y ψ [ψ])
  refute1 ξ (¥ y) ψ     [ψ] [φ] | inj₂ x = [φ]

  refute1' : ∀ ξ φ ψ → ⟦ ξ ⊨ ψ ⟧pl → ⟦ ξ ⊨ φ [ (~ ψ) / ¥false ] ⟧pl → ⟦ ξ ⊨ φ ⟧pl
  refute1' ξ φ ψ [ψ] [φ] with ex-mid ((~ ψ) ≡pl φ)
  ...| inj₁ x rewrite Tb x | sym (lift-≡pl (~ ψ) φ x) = const [φ]
  refute1' ξ ¥true ψ     [ψ] [φ] | inj₂ x = tt
  refute1' ξ ¥false ψ    [ψ] [φ] | inj₂ x = [φ]
  refute1' ξ (y || y') ψ [ψ] [φ] | inj₂ x = Sum.map  (refute1' ξ y ψ [ψ]) (refute1' ξ y' ψ [ψ]) [φ]
  refute1' ξ (y && y') ψ [ψ] [φ] | inj₂ x = Prod.map (refute1' ξ y ψ [ψ]) (refute1' ξ y' ψ [ψ]) [φ]
  refute1' ξ (y => y') ψ [ψ] [φ] | inj₂ x rewrite ¬Tb x
    = (refute1' ξ y' ψ [ψ]) ∘ [φ] ∘ (refute1 ξ y ψ [ψ])
  refute1' ξ (¥ y) ψ     [ψ] [φ] | inj₂ x = [φ]

mutual
  refute2 : ∀ ξ φ ψ → ⟦ ξ ⊨ ~ ψ ⟧pl → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ φ [ ψ / ¥false ] ⟧pl
  refute2 ξ φ ψ [ψ] [φ] with ex-mid (ψ ≡pl φ)
  ...| inj₁ x rewrite Tb x | sym (lift-≡pl ψ φ x) = [ψ] [φ]
  refute2 ξ ¥true ψ     [ψ] [φ] | inj₂ x rewrite ¬Tb x = tt
  refute2 ξ ¥false ψ    [ψ] [φ] | inj₂ x rewrite ¬Tb x = [φ]
  refute2 ξ (y || y') ψ [ψ] [φ] | inj₂ x rewrite ¬Tb x
    = Sum.map (refute2 ξ y ψ [ψ]) (refute2 ξ y' ψ [ψ]) [φ]
  refute2 ξ (y && y') ψ [ψ] [φ] | inj₂ x rewrite ¬Tb x
    = Prod.map (refute2 ξ y ψ [ψ]) (refute2 ξ y' ψ [ψ]) [φ]
  refute2 ξ (y => y') ψ [ψ] [φ] | inj₂ x rewrite ¬Tb x
    = (refute2 ξ y' ψ [ψ]) ∘ [φ] ∘ (refute2' ξ y ψ [ψ])
  refute2 ξ (¥ y) ψ     [ψ] [φ] | inj₂ x rewrite ¬Tb x = [φ]

  refute2' : ∀ ξ φ ψ → ⟦ ξ ⊨ ~ ψ ⟧pl → ⟦ ξ ⊨ φ [ ψ / ¥false ] ⟧pl → ⟦ ξ ⊨ φ ⟧pl
  refute2' ξ φ ψ [ψ] [φ] with ex-mid (ψ ≡pl φ)
  refute2' ξ φ ψ          [ψ] [φ] | inj₁ x rewrite Tb x | sym (lift-≡pl ψ φ x) = ⊥-elim [φ]
  refute2' ξ ¥true ψ     [ψ] [φ] | inj₂ x rewrite ¬Tb x = tt
  refute2' ξ ¥false ψ    [ψ] [φ] | inj₂ x rewrite ¬Tb x = [φ]
  refute2' ξ (y || y') ψ [ψ] [φ] | inj₂ x rewrite ¬Tb x
    = Sum.map (refute2' ξ y ψ [ψ]) (refute2' ξ y' ψ [ψ]) [φ]
  refute2' ξ (y && y') ψ [ψ] [φ] | inj₂ x rewrite ¬Tb x
    = Prod.map (refute2' ξ y ψ [ψ]) (refute2' ξ y' ψ [ψ]) [φ]
  refute2' ξ (y => y') ψ [ψ] [φ] | inj₂ x rewrite ¬Tb x
    = refute2' ξ y' ψ [ψ] ∘ [φ] ∘ refute2 ξ y ψ [ψ]
  refute2' ξ (¥ y) ψ     [ψ] [φ] | inj₂ x rewrite ¬Tb x = [φ]

apply-Γ : ∀ ξ Γ → ⟦ ξ ⊨ Γ ⟧⇒ → ⟦ ξ ⊨ andpl (πΓ Γ) => πφ Γ ⟧pl
apply-Γ ξ ([] ⇒ φ)            pθ pΓ = pθ
apply-Γ ξ ((x ∷ []) ⇒ φ)      pθ pΓ = pθ (proj₁ pΓ)
```

```
apply-Γ ξ ((x :: x' :: xs) ⇒ φ) pθ pΓ = apply-Γ ξ ((x' :: xs) ⇒ φ)
                                          (curry pθ (proj₁ pΓ)) (proj₂ pΓ)

apply-Γ' : ∀ ξ Γ → ⟦ ξ ⊨ andpl (πΓ Γ) => πφ Γ ⟧pl → ⟦ ξ ⊨ Γ ⟧⇒
apply-Γ' ξ ([] ⇒ φ)           p = p tt
apply-Γ' ξ ((γ :: []) ⇒ φ)    p = λ x → (curry p) x tt
apply-Γ' ξ ((γ :: γ' :: Γ) ⇒ φ) p = λ x → apply-Γ' ξ ((γ' :: Γ) ⇒ φ)
                                          (curry p (proj₁ x)) (proj₂ x)

_isSubConjunct_ : PL-Formula → PL-Formula → Bool
_isSubConjunct_ φ ψ with φ ==pl ψ
...| true = true
φ isSubConjunct ¥true     | false = false
φ isSubConjunct ¥false    | false = false
φ isSubConjunct (y || y') | false = false
φ isSubConjunct (y && y') | false = φ isSubConjunct y ∨ φ isSubConjunct y'
φ isSubConjunct (y => y') | false = false
φ isSubConjunct ¥ y       | false = false

lemSubConjunct : ∀ ξ φ ψ → T (φ isSubConjunct ψ) → ⟦ ξ ⊨ ψ ⟧pl → ⟦ ξ ⊨ φ ⟧pl
lemSubConjunct ξ φ ψ p [ψ] with ex-mid (φ ==pl ψ)
...| inj₁ x = lift-==pl← φ ψ x ξ [ψ]
lemSubConjunct ξ φ ¥true     p [ψ] | inj₂ x rewrite ¬Tb x = ⊥-elim p
lemSubConjunct ξ φ ¥false    p [ψ] | inj₂ x rewrite ¬Tb x = ⊥-elim p
lemSubConjunct ξ φ (y || y') p [ψ] | inj₂ x rewrite ¬Tb x = ⊥-elim p
lemSubConjunct ξ φ (y && y') p [ψ] | inj₂ x rewrite ¬Tb x =
                                       [ (λ x → (lemSubConjunct ξ φ y) x (proj₁ [ψ])) ,
                                         (λ x → (lemSubConjunct ξ φ y') x (proj₂ [ψ])) ]'
                                         (lem-bool-∨-s (φ isSubConjunct y) _ p)
lemSubConjunct ξ φ (y => y') p [ψ] | inj₂ x rewrite ¬Tb x = ⊥-elim p
lemSubConjunct ξ φ (¥ y)     p [ψ] | inj₂ x rewrite ¬Tb x = ⊥-elim p

data ERule : Set where
  fof_nnf fof_simplification split_conjunct cn axiom unsat distribute : ERule
  apply_def rw sr pm fresh : ERule

e-arity : ERule → ℕ
e-arity fof_nnf            = 1 -- negated normal form (fol)
e-arity fof_simplification = 1 -- constsant removal (fol)
e-arity axiom              = 0 -- assumption
e-arity apply_def          = 2 -- rewrite a formula with an equivalence, one way
e-arity split_conjunct     = 1 -- split a n-ary conjunction
e-arity rw                 = 2 -- re-write, refute leaving constant
e-arity sr                 = 2 -- simply reflect, refute removing constant where possible
e-arity pm                 = 2 -- paramodulate, refute two clauses
e-arity cn                 = 1 -- clause normalise, remove constants
e-arity unsat              = 1 -- reducto ad adsurdum
e-arity distribute         = 1 -- place a formula in cnf by niave distribution
e-arity fresh              = 1 -- virtual rule, discharges equivalences from introduced definitions

correct-apply : Formula → Formula → Formula → Bool
correct-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((φ₂a => φ₂b) && (φ₂b' => φ₂a'))) (Γ₃ ⇒ φ₃)
  = φ₂a ≡pl φ₂a' ∧ φ₂b ≡pl φ₂b' ∧ φ₃ ≡pl (φ₁ [ φ₂b / φ₂a ]) ∧ (Γ₁ ++ Γ₂) ⊆ Γ₃
correct-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ φ₂) (Γ₃ ⇒ φ₃) = false

correct-fresh : Formula → Formula → Bool
correct-fresh ((((¥ n => ψ) && (ψ' => ¥ n')) :: Γ₁) ⇒ φ₁) (Γ₂ ⇒ φ₂)
  = φ₁ ≡pl φ₂ ∧ n == n' ∧ ψ ≡pl ψ' ∧ not (¥ n isSubFormula φ₂)
    ∧ not (¥ n isSubFormula andpl Γ₂)
    ∧ not (¥ n isSubFormula ψ) ∧ Γ₁ ⊆ Γ₂
correct-fresh (_ ⇒ φ₁) (Γ₂ ⇒ φ₂) = false

correct-rw :  Formula → Formula → Formula → Bool
correct-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (φ₂ => ¥false)) (Γ₃ ⇒ φ₃) = φ₃ ==pl (φ₁ [ φ₂ / ¥false ])
                                                      ∧ (Γ₁ ++ Γ₂) ⊆ Γ₃
correct-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ φ₂) (Γ₃ ⇒ φ₃) = φ₃ ==pl (φ₁ [ ~ φ₂ / ¥false ])
                                          ∧ (Γ₁ ++ Γ₂) ⊆ Γ₃

correct-sr : Formula → Formula → Formula → Bool
correct-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (φ₂ => ¥false)) (Γ₃ ⇒ φ₃)
  = φ₃ ==pl const-removal (φ₁ [ φ₂ / ¥false ]) ∧ (Γ₁ ++ Γ₂) ⊆ Γ₃
correct-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ φ₂) (Γ₃ ⇒ φ₃) = φ₃ ==pl const-removal (φ₁ [ ~ φ₂ / ¥false ])
                                          ∧ (Γ₁ ++ Γ₂) ⊆ Γ₃

correct-pm : Formula → Formula → Formula → Bool
correct-pm (Γ₁ ⇒ φ₁) (Γ₂ ⇒ φ₂) (Γ₃ ⇒ φ₃)
  = conflict-negleft φ₁ φ₂ ∧ φ₃ ==pl const-removal (pm' φ₁ φ₂) ∧ (Γ₁ ++ Γ₂) ⊆ Γ₃

e-correct : (k : ERule) → Vec Formula (e-arity k) → Formula → Bool
e-correct fof_nnf            (Γ₁ ⇒ φ :: xs) (Γ₂ ⇒ ψ)  = ψ ==pl const-removal (mknnf φ) ∧ Γ₁ ⊆ Γ₂
e-correct fof_simplification (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) = φ₂ ==pl const-removal φ₁ ∧ Γ₁ ⊆ Γ₂
e-correct distribute         (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) = φ₂ ==pl mkdist φ₁ ∧ Γ₁ ⊆ Γ₂
e-correct axiom              seq            (Γ₁ ⇒ φ₁) = φ₁ ∈ Γ₁
e-correct apply_def          (a :: b :: _)  c         = correct-apply a b c
e-correct split_conjunct     (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) = φ₂ isSubConjunct φ₁ ∧ Γ₁ ⊆ Γ₂
e-correct rw                 (a :: b :: _)  c         = correct-rw a b c
e-correct sr                 (a :: b :: _)  c         = correct-sr a b c
```

```
e-correct pm                (a :: b :: _)   c          = correct-pm a b c
e-correct cn                (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) = φ₂ ==pl const-removal φ₁ ∧ Γ₁ ⊆ Γ₂
e-correct unsat             (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) = φ₁ ≡pl ¥false ∧ (Γ₁ ∣ (~ φ₂)) ⊆ Γ₂
e-correct fresh             (a :: _)        b          = correct-fresh a b

sound-apply : (a b c : Formula) → (∀ ξ → ⟦ ξ ⊨ a ⟧⇒) → (∀ ξ → ⟦ ξ ⊨ b ⟧⇒) → T (correct-apply a b c)
           → ∀ ξ → ⟦ ξ ⊨ c ⟧⇒
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((φ₂a => φ₂b) && (φ₂b' => φ₂a'))) (Γ₃ ⇒ φ₃) pa pb p ξ
  rewrite (lift-≡pl φ₃ _ (∧-eliml (∧-elimr (φ₂b ≡pl φ₂b') (∧-elimr (φ₂a ≡pl φ₂a') p))))
  = apply-Γ' ξ (Γ₃ ⇒ (φ₁ [ φ₂b / φ₂a ]) )
             (λ [Γ₃] → lem-subst ξ φ₂b φ₂a φ₁ (Prod.map id
                         (λ x' x0 → subst (⟦_⊨_⟧pl ξ) (sym (lift-≡pl φ₂a _ (∧-eliml p)))
                                          (x' (subst (⟦_⊨_⟧pl ξ) (lift-≡pl φ₂b _
                                                                  (∧-eliml (∧-elimr (φ₂a ≡pl φ₂a') p))) x0)))
                         (apply-Γ ξ (Γ₂ ⇒ ((φ₂a => φ₂b) && (φ₂b' => φ₂a'))) (pb ξ)
                                  (proj₂ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
                                         (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ ((∧-elimr
                                         (φ₃ ≡pl (φ₁ [ φ₂b / φ₂a ])) (∧-elimr (φ₂b ≡pl φ₂b')
                                         (∧-elimr (φ₂a ≡pl φ₂a') p))))) [Γ₃])))))
                       (apply-Γ ξ (Γ₁ ⇒ φ₁) (pa ξ) (proj₁ (seq-split ξ Γ₁ Γ₂
                              (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂) (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃
                               (∧-elimr (φ₃ ≡pl (φ₁ [ φ₂b / φ₂a ])) (∧-elimr
                               (φ₂b ≡pl φ₂b') (∧-elimr (φ₂a ≡pl φ₂a') p)))) [Γ₃])))))
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥true)                    (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥false)                   (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y || y'))                (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (¥true && y'))            (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (¥false && y'))           (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((y || y') && y0))        (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((y && y') && y0))        (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((y => y') && ¥true))     (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((y => y') && ¥false))    (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((y => y') && (y0 || y1))) (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((y => y') && (y0 && y1))) (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ((y => y') && ¥ y0))      (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (¥ y && y'))              (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => y'))                (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p
sound-apply (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥ y)                     (Γ₃ ⇒ φ₃) pa pb p ξ = ⊥-elim p

sound-fresh : (a b : Formula) → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ a) => πφ a ⟧pl) → T (correct-fresh a b)
           → ∀ ξ → ⟦ ξ ⊨ andpl (πΓ b) => πφ b ⟧pl
sound-fresh ([] ⇒ φ₁) (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((¥true :: Γ₁) ⇒ φ₁)                                  (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((¥false :: Γ₁) ⇒ φ₁)                                 (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((y || y' :: Γ₁) ⇒ φ₁)                                (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((¥true && y' :: Γ₁) ⇒ φ₁)                            (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((¥false && y' :: Γ₁) ⇒ φ₁)                           (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((y || y') && y0 :: Γ₁) ⇒ φ₁)                        (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((y && y') && y0 :: Γ₁) ⇒ φ₁)                        (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥true => y') && y0 :: Γ₁) ⇒ φ₁)                    (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥false => y') && y0 :: Γ₁) ⇒ φ₁)                   (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((((y || y') => y0) && y1 :: Γ₁) ⇒ φ₁)                (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((((y && y') => y0) && y1 :: Γ₁) ⇒ φ₁)                (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((((y => y') => y0) && y1 :: Γ₁) ⇒ φ₁)                (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && ¥true :: Γ₁) ⇒ φ₁)                   (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && ¥false :: Γ₁) ⇒ φ₁)                  (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && (y0 || y1) :: Γ₁) ⇒ φ₁)              (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && (y0 && y1) :: Γ₁) ⇒ φ₁)              (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && (y0 => ¥true) :: Γ₁) ⇒ φ₁)           (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && (y0 => ¥false) :: Γ₁) ⇒ φ₁)          (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && (y0 => (y1 || y2)) :: Γ₁) ⇒ φ₁)      (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && (y0 => (y1 && y2)) :: Γ₁) ⇒ φ₁)      (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ y => y') && (y0 => (y1 => y2)) :: Γ₁) ⇒ φ₁)      (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh (((¥ n => ψ) && (ψ' => ¥ n') :: Γ₁) ⇒ φ₁) (Γ₂ ⇒ φ₂) pa p ξ [Γ₂]
  rewrite lift-≡pl φ₁ _ (∧-eliml p)
  = r' φ₂ ψ (andpl Γ₂) n
       (∧-eliml (∧-elimr (ψ ≡pl ψ') (∧-elimr (n == n') (∧-elimr (φ₁ ≡pl φ₂) p))))
       (∧-eliml (∧-elimr  (not (¥ n isSubFormula andpl Γ₂))
                          (∧-elimr (not (¥ n isSubFormula φ₂))
                                   (∧-elimr (ψ ≡pl ψ')
                                            (∧-elimr (n == n') (∧-elimr (φ₁ ≡pl φ₂) p))))))
       (∧-eliml (∧-elimr (not (¥ n isSubFormula φ₂))
                         (∧-elimr (ψ ≡pl ψ') (∧-elimr (n == n') (∧-elimr (φ₁ ≡pl φ₂) p)))))
       (λ ξ' x → pa ξ' (Prod.map (Prod.map id (λ x0 x1 → subst (T ∘ ξ')
                  (lift-== n n' ((∧-eliml (∧-elimr (φ₁ ≡pl φ₂) p))))
                  (x0 (subst (⟦_⊨_⟧pl ξ') (sym (lift-≡pl ψ _ ((∧-eliml (∧-elimr (n == n')
                      (∧-elimr (φ₁ ≡pl φ₂) p)))))) x1))))
                              (lem-seq-subst-foldr ξ' Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂
                      (∧-elimr (not (¥ n isSubFormula ψ)) (∧-elimr (not (¥ n
                      isSubFormula andpl Γ₂)) (∧-elimr (not (¥ n isSubFormula φ₂))
                      (∧-elimr (ψ ≡pl ψ') (∧-elimr (n == n') (∧-elimr (φ₁ ≡pl φ₂) p))))))))) x))
       ξ [Γ₂]
sound-fresh (((¥ y => y') && ¥ y0 :: Γ₁) ⇒ φ₁)                    (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((¥ y && y' :: Γ₁) ⇒ φ₁)                              (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((y => y' :: Γ₁) ⇒ φ₁)                                (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
sound-fresh ((¥ y :: Γ₁) ⇒ φ₁)                                    (Γ₂ ⇒ φ₂) pa p ξ [Γ₂] = ⊥-elim p
```

```
sound-rw' : (a b c : Formula)
          → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ a) => πφ a ⟧pl) → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ b) => πφ b ⟧pl)
          → T (πφ c ==pl ((πφ a) [ ~ (πφ b) / ¥false ]) ∧ ((πΓ a) ++ (πΓ b)) ⊆ (πΓ c))
          → ∀ ξ → ⟦ ξ ⊨ andpl (πΓ c) => πφ c ⟧pl
sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ φ₂) (Γ₃ ⇒ φ₃) pa pb p ξ [Γ₃]
  = lift-==pl← φ₃ ( φ₁ [ (~ φ₂) / ¥false ] ) (∧-eliml p) ξ
              (refute1 ξ φ₁ φ₂
                (pb ξ (proj₂ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
                    (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl (φ₁ [ ~ φ₂ / ¥false ])) p))
                    [Γ₃])))))
                (pa ξ (proj₁ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
                    (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl (φ₁ [ ~ φ₂ / ¥false ])) p))
                    [Γ₃]))))))

sound-rw : (a b c : Formula)
         → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ a) => πφ a ⟧pl) → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ b) => πφ b ⟧pl)
         → T (correct-rw a b c )
         → ∀ ξ → ⟦ ξ ⊨ foldr _&&_ ¥true (πΓ c) => πφ c ⟧pl
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥true)             (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥false)            (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y || y'))         (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y && y'))         (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => ¥true))      (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => (y' || y0))) (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => (y' && y0))) (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => (y' => y0))) (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => ¥ y'))       (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥ y)               (Γ₃ ⇒ φ₃) = sound-rw' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)

sound-rw (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (φ₂ => ¥false)) (Γ₃ ⇒ φ₃) = λ pa pb p ξ [Γ₃] →
  lift-==pl← φ₃ ( φ₁ [ φ₂ / ¥false ] ) (∧-eliml p) ξ
              (refute2 ξ φ₁ φ₂
                (pb ξ (proj₂ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
                    (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl (φ₁ [ φ₂ / ¥false ])) p)) [Γ₃])))))
                (pa ξ (proj₁ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
                    (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl (φ₁ [ φ₂ / ¥false ])) p)) [Γ₃]))))))

sound-sr' : (a b c : Formula)
         → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ a) => πφ a ⟧pl) → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ b) => πφ b ⟧pl)
         → T (πφ c ==pl const-removal ((πφ a) [ ~ (πφ b) / ¥false ])
                ∧ (((πΓ a) ++ (πΓ b)) ⊆ (πΓ c)))
         → ∀ ξ → ⟦ ξ ⊨ andpl (πΓ c) => πφ c ⟧pl
sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ φ₂) (Γ₃ ⇒ φ₃) pa pb p ξ [Γ₃]
  = lift-==pl← φ₃ (const-removal (φ₁ [ (~ φ₂) / ¥false ]))
     (∧-eliml p) ξ (lem-no-const ξ  (φ₁ [ (~ φ₂) / ¥false ]) (refute1 ξ φ₁ φ₂
       (pb ξ (proj₂ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
         (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl const-removal (φ₁ [ ~ φ₂ / ¥false ]))p))
         [Γ₃])))))
       (pa ξ (proj₁ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
         (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl const-removal (φ₁ [ ~ φ₂ / ¥false ]))p))
         [Γ₃])))) ))

sound-sr : (a b c : Formula)
         → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ a) => πφ a ⟧pl) → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ b) => πφ b ⟧pl)
         → T (correct-sr a b c) → ∀ ξ → ⟦ ξ ⊨ andpl (πΓ c) => πφ c ⟧pl
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥true)           (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥false)          (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y || y'))       (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y && y'))       (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => ¥true))    (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (φ₂ => ¥false)) (Γ₃ ⇒ φ₃) = λ  pa pb p ξ [Γ₃] →
  lift-==pl← φ₃ (const-removal (φ₁ [ φ₂ / ¥false ]))
    (∧-eliml p) ξ
    (lem-no-const ξ (φ₁ [ φ₂ / ¥false ])
      (refute2 ξ φ₁ φ₂ (pb ξ (proj₂ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
        (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl const-removal (φ₁ [ φ₂ / ¥false ])) p))
        [Γ₃])))))
      (pa ξ (proj₁ (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂)
          (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl const-removal (φ₁ [ φ₂ / ¥false ])) p))
          [Γ₃])))) ))
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => (y' || y0))) (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => (y' && y0))) (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => (y' => y0))) (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ (y => ¥ y'))       (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)
sound-sr (Γ₁ ⇒ φ₁) (Γ₂ ⇒ ¥ y)               (Γ₃ ⇒ φ₃) = sound-sr' (Γ₁ ⇒ φ₁) (Γ₂ ⇒ _) (Γ₃ ⇒ φ₃)

sound-pm : (a b c : Formula)
         → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ a) => πφ a ⟧pl) → (∀ ξ → ⟦ ξ ⊨ andpl (πΓ b) => πφ b ⟧pl)
         → T (correct-pm a b c )
         → ∀ ξ → ⟦ ξ ⊨ andpl (πΓ c) => πφ c ⟧pl
sound-pm (Γ₁ ⇒ φ₁) (Γ₂ ⇒ φ₂) (Γ₃ ⇒ φ₃) pa pb p ξ [Γ₃]
  = lift-==pl← φ₃ (const-removal (pm' φ₁ φ₂)) (∧-eliml (∧-elimr (conflict-negleft φ₁ φ₂) p))
    ξ (lem-no-const ξ (pm' φ₁ φ₂) (lem-pm φ₁ φ₂ (∧-eliml p) ξ (pa ξ (proj₁ (seq-split ξ Γ₁ Γ₂
      (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂) (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr (φ₃ ==pl
      const-removal (pm' φ₁ φ₂)) (∧-elimr (conflict-negleft φ₁ φ₂) p))) [Γ₃]))))) (pb ξ (proj₂
      (seq-split ξ Γ₁ Γ₂ (lem-seq-subst-foldr ξ Γ₃ (Γ₁ ∪ Γ₂) (lift-⊆ (Γ₁ ∪ Γ₂) Γ₃ (∧-elimr
      (φ₃ ==pl const-removal (pm' φ₁ φ₂)) (∧-elimr (conflict-negleft φ₁ φ₂) p)))
```

```
       [Γ₃]))))) ))

esound : (k : ERule)
       → (seq : Vec Formula (e-arity k))
       → (conc : Formula)
       → T (e-correct k seq conc)
       → Vec* (λ x → ∀ ξ → ⟦ ξ ⊨ x ⟧⇒) seq
       → ∀ ξ → ⟦ ξ ⊨ conc ⟧⇒
esound fof_nnf (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) correct (v :: _) ξ
  = apply-Γ' ξ (Γ₂ ⇒ φ₂)
             (λ x → lift-==pl← φ₂ (const-removal (mknnf φ₁)) (∧-eliml correct) ξ
               (lem-no-const ξ (mknnf φ₁) (lem-nnf← φ₁ ξ
                 (apply-Γ ξ (Γ₁ ⇒ φ₁) (v ξ)
                   (lem-seq-subst-foldr ξ Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂ (∧-elimr
                     (φ₂ ==pl const-removal (mknnf φ₁)) correct)) x)))))
esound fof_simplification (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) correct (v :: _) ξ
  = apply-Γ' ξ (Γ₂ ⇒ φ₂)
             (λ x → lift-==pl← φ₂ (const-removal φ₁) (∧-eliml correct) ξ
               (lem-no-const ξ φ₁ (apply-Γ ξ (Γ₁ ⇒ φ₁) (v ξ)
                 (lem-seq-subst-foldr ξ Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂ (∧-elimr
                   (φ₂ ==pl const-removal φ₁) correct)) x))))
esound distribute (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) correct (v :: _) ξ
  = apply-Γ' ξ (Γ₂ ⇒ φ₂)
             (λ x → lift-==pl← φ₂ (mkdist φ₁) (∧-eliml correct) ξ (lem-mkdist ξ φ₁
               (apply-Γ ξ (Γ₁ ⇒ φ₁) (v ξ) (lem-seq-subst-foldr ξ Γ₂ Γ₁
                 (lift-⊆ Γ₁ Γ₂ (∧-elimr (φ₂ ==pl mkdist φ₁) correct)) x))))
esound split_conjunct (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) correct (v :: _) ξ
  = apply-Γ' ξ (Γ₂ ⇒ φ₂) (λ x → lemSubConjunct ξ φ₂ φ₁ (∧-eliml correct)
     (apply-Γ ξ (Γ₁ ⇒ φ₁) (v ξ) (lem-seq-subst-foldr ξ Γ₂ Γ₁ (lift-⊆ Γ₁ Γ₂
       (∧-elimr (φ₂ isSubConjunct φ₁) correct)) x)))
esound cn (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) correct (v :: _) ξ
  = apply-Γ' ξ (Γ₂ ⇒ φ₂) (λ x → lift-==pl← φ₂ (const-removal φ₁) (∧-eliml correct) ξ
      (lem-no-const ξ φ₁ (apply-Γ ξ (Γ₁ ⇒ φ₁) (v ξ) (lem-seq-subst-foldr ξ Γ₂ Γ₁
        (lift-⊆ Γ₁ Γ₂ (∧-elimr (φ₂ ==pl const-removal φ₁) correct)) x))))
esound axiom seq ([] ⇒ φ) () v ξ
esound axiom seq ((γ :: Γ) ⇒ φ) correct v ξ
  = apply-Γ' ξ ((γ :: Γ) ⇒ φ) (∨-elim (λ x → subst (⟦_⊨_⟧pl ξ) (sym (lift-≡pl φ _ x)) ∘ proj₁)
             (λ x x' → apply-Γ ξ (Γ ⇒ φ) (esound axiom seq (Γ ⇒ φ) x v ξ) (proj₂ x')) correct)
esound apply_def (a :: b :: _) c correct (v₁ :: v₂ :: _) ξ = sound-apply a b c v₁ v₂ correct ξ
esound rw (a :: b :: _) c correct (v₁ :: v₂ :: _) ξ
  = apply-Γ' ξ c (sound-rw a b c (λ ξ' → apply-Γ ξ' a (v₁ ξ'))
                                 (λ ξ' → apply-Γ ξ' b (v₂ ξ')) correct ξ)
esound sr (a :: b :: _) c correct (v₁ :: v₂ :: _) ξ
  = apply-Γ' ξ c (sound-sr a b c (λ ξ' → apply-Γ ξ' a (v₁ ξ'))
                                 (λ ξ' → apply-Γ ξ' b (v₂ ξ')) correct ξ)
esound pm (a :: b :: _) c correct (v₁ :: v₂ :: _) ξ
  = apply-Γ' ξ c (sound-pm a b c (λ ξ' → apply-Γ ξ' a (v₁ ξ'))
                                 (λ ξ' → apply-Γ ξ' b (v₂ ξ')) correct ξ)
esound unsat (Γ₁ ⇒ φ₁ :: _) (Γ₂ ⇒ φ₂) correct (v :: _) ξ
  = apply-Γ' ξ (Γ₂ ⇒ φ₂) λ x → stbl-pl ξ φ₂
      (λ x' → subst id (cong (⟦_⊨_⟧pl ξ) (lift-≡pl φ₁ _ (∧-eliml correct)))
                    (apply-Γ ξ (Γ₁ ⇒ φ₁) (v ξ)
                     (lem-seq-restrict-foldr' ξ Γ₁ (~ φ₂) x'
                      (lem-seq-subst-foldr ξ Γ₂ (Γ₁ ∣ ~ φ₂)
                       (lift-⊆ (Γ₁ ∣ ~ φ₂) Γ₂ (∧-elimr (φ₁ ≡pl ¥false) correct)) x))))
esound fresh (a :: _) b correct (v :: _) ξ
  = apply-Γ' ξ b (sound-fresh a b (λ ξ' → apply-Γ ξ' a (v ξ')) correct ξ)

E : RuleSystem ERule Formula Env ⟦_⊨_⟧⇒
E = record { arity = e-arity; correct = e-correct; sound = esound }

correct-list'ᵇ : (l : ProofList E) → T (not (null l)) → Bool
correct-list'ᵇ l x = correct-list E l ∧ (null $ πΓ $ ProofNode.conc E $ lastnode l x)

semantics-elist' : (l : ProofList E) → (x : T (not (null l))) → Set
semantics-elist' l x = ∀ ξ → ⟦ ξ ⊨ ProofNode.conc E (lastnode l x) ⟧⇒

aux-reconst : (l : ProofList E) → (x : T (not (null l)))
            → (p : T (correct-list'ᵇ l x)) → semantics-elist' l x
aux-reconst l x p = sound-list E l x (∧-eliml p)

lemKK : ∀ ξ θ → ⟦ ξ ⊨ θ ⟧⇒ → T (null (πΓ θ)) → ⟦ ξ ⊨ πφ θ ⟧pl
lemKK ξ ([] ⇒ φ) p q = p
lemKK ξ ((x :: xs) ⇒ φ) p q = ⊥-elim q

correct-elist-φᵇ : (l : Maybe (ProofList E)) → (φ : PL-Formula) → Bool
correct-elist-φᵇ (just l) φ with ex-mid (not (null l))
...| inj₁ x = correct-list'ᵇ l x ∧ (φ ≡pl πφ (ProofNode.conc E $ lastnode l x))
...| inj₂ x = false
correct-elist-φᵇ nothing φ = false

aux-reconstruct : ∀ φ l x → T (correct-list'ᵇ l x ∧ (φ ≡pl πφ (ProofNode.conc E $ lastnode l x)))
                → Taut-pl φ
aux-reconstruct φ l x pτ ξ
  rewrite lift-≡pl φ _ (∧-elimr (correct-list'ᵇ l x) pτ)
  = lemKK ξ (ProofNode.conc E $ lastnode l x) (aux-reconst l x (∧-eliml pτ) ξ)
          (∧-elimr (correct-list E l) (∧-eliml pτ))
```

```
reconstruct : ∀ φ l → T (correct-elist-φᵇ l φ) → Taut-pl φ
reconstruct φ nothing ()
reconstruct φ (just l) p with ex-mid (not (null l))
...| inj₁ x = aux-reconstruct φ l x p
...| inj₂ x = ⊥-elim p

open import Boolean.TPTP
open import Data.String

private primitive primExternal : {A : Set} → String → String → Maybe A

createList : PL-Formula → Maybe (ProofList E)
createList φ = primExternal {ProofList E} "eprover-list" (tptp φ)

correctness : ∀ φ → Set
correctness φ = T (correct-elist-φᵇ (createList φ) φ)

soundness : ∀ φ → correctness φ → Taut-pl φ
soundness φ p = reconstruct φ (createList φ) p
```

```
module Boolean.PL-Formula.Substitute where

open import Boolean.Formula

open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Unit
open import Data.Bool

open import PropIso


substitute : PL-Formula → PL-Formula → PL-Formula → PL-Formula
substitute φ ψ ρ with φ ≡pl ρ
...| true = ψ
substitute φ ψ ¥true | false = ¥true
substitute φ ψ ¥false | false = ¥false
substitute φ ψ (y || y') | false = substitute φ ψ y || substitute φ ψ y'
substitute φ ψ (y && y') | false = substitute φ ψ y && substitute φ ψ y'
substitute φ ψ (y => y') | false = substitute φ ψ y => substitute φ ψ y'
substitute φ ψ (¥ y) | false = ¥ y


-- substitute φ for ψ in ρ
syntax substitute φ ψ ρ = ρ [ φ / ψ ]

mutual
  lem-subst : ∀ ξ φ ψ ρ → ⟦ ξ ⊨ ψ <=> φ ⟧pl → ⟦ ξ ⊨ ρ ⟧pl → ⟦ ξ ⊨ ρ [ φ / ψ ] ⟧pl
  lem-subst ξ φ ψ ρ [φ↔ψ] [ρ] with ex-mid (φ ≡pl ρ)
  ...| inj₁ x  rewrite Tb x | lift-≡pl φ _ x = (proj₂ [φ↔ψ]) [ρ]
  lem-subst ξ φ ψ ¥true     [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = tt
  lem-subst ξ φ ψ ¥false    [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = [ρ]
  lem-subst ξ φ ψ (y || y') [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = Sum.map (lem-subst ξ φ ψ y [φ↔ψ])
                                                                      (lem-subst ξ φ ψ y' [φ↔ψ]) [ρ]
  lem-subst ξ φ ψ (y && y') [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = Prod.map (lem-subst ξ φ ψ y [φ↔ψ])
                                                                       (lem-subst ξ φ ψ y' [φ↔ψ]) [ρ]
  lem-subst ξ φ ψ (y => y') [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = lem-subst ξ φ ψ y' [φ↔ψ] ∘ [ρ] ∘
                                                                 lem-subst' ξ φ ψ y [φ↔ψ]
  lem-subst ξ φ ψ (¥ y)     [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = [ρ]

  lem-subst' : ∀ ξ φ ψ ρ → ⟦ ξ ⊨ ψ <=> φ ⟧pl → ⟦ ξ ⊨ ρ [ φ / ψ ] ⟧pl → ⟦ ξ ⊨ ρ ⟧pl
  lem-subst' ξ φ ψ ρ [φ↔ψ] [ρ] with ex-mid (φ ≡pl ρ)
  ...| inj₁ x  rewrite Tb x | lift-≡pl φ _ x = (proj₁ [φ↔ψ]) [ρ]
  lem-subst' ξ φ ψ ¥true     [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = tt
  lem-subst' ξ φ ψ ¥false    [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = [ρ]
  lem-subst' ξ φ ψ (y || y') [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = Sum.map (lem-subst' ξ φ ψ y [φ↔ψ])
                                                                       (lem-subst' ξ φ ψ y' [φ↔ψ]) [ρ]
  lem-subst' ξ φ ψ (y && y') [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = Prod.map (lem-subst' ξ φ ψ y [φ↔ψ])
                                                                        (lem-subst' ξ φ ψ y' [φ↔ψ]) [ρ]
  lem-subst' ξ φ ψ (y => y') [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = lem-subst' ξ φ ψ y' [φ↔ψ] ∘ [ρ] ∘
                                                                  lem-subst ξ φ ψ y [φ↔ψ]
  lem-subst' ξ φ ψ (¥ y)     [φ↔ψ] [ρ] | inj₂ x rewrite ¬Tb x = [ρ]
```

```
module Proof.EProver.NNF where

open import Boolean.Formula

open import Data.Bool
open import Data.Nat
open import Data.Sum as Sum
open import Data.Product as Prod
open import PropIso

mutual
  mknnf : PL-Formula → PL-Formula
  mknnf ¥true     = ¥true
  mknnf ¥false    = ¥false
  mknnf (y || y') = mknnf y || mknnf y'
  mknnf (y && y') = mknnf y && mknnf y'
  mknnf (y => y') = ¬mknnf y || mknnf y'
  mknnf (¥ y)     = ¥ y

  ¬mknnf : PL-Formula → PL-Formula
  ¬mknnf ¥true     = ¥false
  ¬mknnf ¥false    = ¥true
  ¬mknnf (y || y') = ¬mknnf y && ¬mknnf y'
  ¬mknnf (y && y') = ¬mknnf y || ¬mknnf y'
  ¬mknnf (y => y') = mknnf y && ¬mknnf y'
  ¬mknnf (¥ y)     = ~ (¥ y)

mutual
  lem-nnf→ : (φ : PL-Formula) → (ξ : ℕ → Bool) → ⟦ ξ ⊨ mknnf φ ⟧pl → ⟦ ξ ⊨ φ ⟧pl
  lem-nnf→ ¥true    ξ p = tt
  lem-nnf→ ¥false   ξ p = p
  lem-nnf→ (φ || ψ) ξ p = Sum.map (lem-nnf→ φ ξ) (lem-nnf→ ψ ξ) p
  lem-nnf→ (φ && ψ) ξ p = Prod.map (lem-nnf→ φ ξ) (lem-nnf→ ψ ξ) p
  lem-nnf→ (φ => ψ) ξ p = λ x → lem-nnf→ ψ ξ (lem-→ (Sum.map (lem-nnf'→ φ ξ) id p) x)
  lem-nnf→ (¥ v)    ξ p = p

  lem-nnf← : (φ : PL-Formula) → (ξ : ℕ → Bool) → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ mknnf φ ⟧pl
  lem-nnf← ¥true    ξ p = p
  lem-nnf← ¥false   ξ p = p
  lem-nnf← (φ || ψ) ξ p = Sum.map (lem-nnf← φ ξ) (lem-nnf← ψ ξ) p
  lem-nnf← (φ && ψ) ξ p = Prod.map (lem-nnf← φ ξ) (lem-nnf← ψ ξ) p
  lem-nnf← (φ => ψ) ξ p = Sum.map (lem-nnf'← φ ξ) (lem-nnf← ψ ξ) (material-pl ξ φ ψ p)
  lem-nnf← (¥ v)    ξ p = p

  lem-nnf'→ : (φ : PL-Formula) → (ξ : ℕ → Bool) → ⟦ ξ ⊨ ¬mknnf φ ⟧pl → ⟦ ξ ⊨ ~ φ ⟧pl
  lem-nnf'→ ¥true    ξ p q = p
  lem-nnf'→ ¥false   ξ p q = q
  lem-nnf'→ (φ || ψ) ξ p q = [ lem-nnf'→ φ ξ (proj₁ p) , lem-nnf'→ ψ ξ (proj₂ p) ]′ q
  lem-nnf'→ (φ && ψ) ξ p q
    = [ (λ x → lem-nnf'→ φ ξ x (proj₁ q)) , (λ x → lem-nnf'→ ψ ξ x (proj₂ q)) ]′ p
  lem-nnf'→ (φ => ψ) ξ p q = lem-nnf'→ ψ ξ (proj₂ p) (q (lem-nnf→ φ ξ (proj₁ p)))
  lem-nnf'→ (¥ v)    ξ p q = p q

  lem-nnf'← : (φ : PL-Formula) → (ξ : ℕ → Bool) → ⟦ ξ ⊨ ~ φ ⟧pl → ⟦ ξ ⊨ ¬mknnf φ ⟧pl
  lem-nnf'← ¥true    ξ p = p tt
  lem-nnf'← ¥false   ξ p = tt
  lem-nnf'← (φ || ψ) ξ p = (lem-nnf'← φ ξ (λ x → p (inj₁ x))) , lem-nnf'← ψ ξ (λ x → p (inj₂ x))
  lem-nnf'← (φ && ψ) ξ p = Sum.map (lem-nnf'← φ ξ) (lem-nnf'← ψ ξ) (demorg ξ φ ψ p)
  lem-nnf'← (φ => ψ) ξ p = Prod.map (lem-nnf← φ ξ) (lem-nnf'← ψ ξ) (material-¬pl ξ φ ψ p)
  lem-nnf'← (¥ v)    ξ p = p
```

```
module Proof.EProver.PM where

open import Boolean.Formula

open import Data.Bool
open import Data.Product as Prod
open import Data.Sum as Sum

open import Relation.Binary.PropositionalEquality

open import PropIso

_isSubDisjunct_ : PL-Formula → PL-Formula → Bool
_isSubDisjunct_ φ ψ with φ ≡pl ψ
...| true = true
φ isSubDisjunct ¥true     | false = false
φ isSubDisjunct ¥false    | false = false
φ isSubDisjunct (y && y') | false = false
φ isSubDisjunct (y || y') | false = φ isSubDisjunct y ∨ φ isSubDisjunct y'
φ isSubDisjunct (y => y') | false = false
φ isSubDisjunct ¥ y       | false = false

subst-disjunct : (φ₁ φ₂ ψ : PL-Formula) → PL-Formula
subst-disjunct φ₁ φ₂ ψ with φ₁ ≡pl ψ
subst-disjunct φ₁ φ₂ ψ | true = φ₂
subst-disjunct φ₁ φ₂ (ψ₁ || ψ₂) | false = (subst-disjunct φ₁ φ₂ ψ₁) || (subst-disjunct φ₁ φ₂ ψ₂)
subst-disjunct φ₁ φ₂ ψ | false = ψ

conflict-negleft : PL-Formula → PL-Formula → Bool
conflict-negleft φ ¥true = ~ (¥true) isSubDisjunct φ
conflict-negleft φ ¥false = ~ (¥false) isSubDisjunct φ
conflict-negleft φ (ψ || ψ₁) = conflict-negleft φ ψ ∨ conflict-negleft φ ψ₁
conflict-negleft φ (ψ && ψ₁) = ~ (ψ && ψ₁) isSubDisjunct φ
conflict-negleft φ (ψ => ψ₁) = ~ (ψ => ψ₁) isSubDisjunct φ
conflict-negleft φ (¥ x) = ~ (¥ x) isSubDisjunct φ

-- assume conflict-negleft φ₁ φ₂ for pm'
pm' : (φ₁ φ₂ : PL-Formula)
    → PL-Formula
pm' φ₁ (φ₂ || φ₃) with conflict-negleft φ₁ φ₂
...| true  = (pm' φ₁ φ₂) || φ₃
...| false = (pm' φ₁ φ₃) || φ₂ -- this is implied by assumption
pm' φ₁ φ₂ = subst-disjunct (~ φ₂) ¥false φ₁

lem-subst-d : ∀ φ₁ φ₂ ψ ξ
            → ¬ T (φ₁ isSubDisjunct ψ)
            → ⟦ ξ ⊨ ψ ⟧pl
            → ⟦ ξ ⊨ subst-disjunct φ₁ φ₂ ψ ⟧pl
lem-subst-d φ₁ φ₂ ψ ξ p [ψ] with φ₁ ≡pl ψ
lem-subst-d φ₁ φ₂ ψ ξ p [ψ] | true = ⊥-elim (p tt)
lem-subst-d φ₁ φ₂ ¥true ξ p [ψ] | false = [ψ]
lem-subst-d φ₁ φ₂ ¥false ξ p [ψ] | false = [ψ]
lem-subst-d φ₁ φ₂ (ψ || ψ₁) ξ p [ψ] | false
  = Sum.map (lem-subst-d φ₁ φ₂ ψ ξ (λ x → p (∨-introl _ _ x)))
            (lem-subst-d φ₁ φ₂ ψ₁ ξ (λ x → p (∨-intror (φ₁ isSubDisjunct ψ) _ x))) [ψ]
lem-subst-d φ₁ φ₂ (ψ && ψ₁) ξ p [ψ] | false = [ψ]
lem-subst-d φ₁ φ₂ (ψ => ψ₁) ξ p [ψ] | false = [ψ]
lem-subst-d φ₁ φ₂ (¥ x) ξ p [ψ] | false = [ψ]

lem-pm : ∀ φ₁ φ₂
          → T (conflict-negleft φ₁ φ₂)
          → ∀ ξ
          → ⟦ ξ ⊨ φ₁ ⟧pl
          → ⟦ ξ ⊨ φ₂ ⟧pl
          → ⟦ ξ ⊨ pm' φ₁ φ₂ ⟧pl
lem-pm φ₁ ¥true p ξ [φ₁] [φ₂] with ex-mid ( (¥true => ¥false) ≡pl φ₁)
lem-pm φ₁ ¥true p ξ [φ₁] [φ₂] | inj₁ x rewrite Tb x
  = subst (λ k → ⟦ ξ ⊨ k ⟧pl) (sym (lift-≡pl (¥true => ¥false) φ₁ x)) [φ₁] [φ₂]
lem-pm ¥true ¥true () ξ [φ₁] [φ₂] | inj₂ y
lem-pm ¥false ¥true () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ || φ₂) ¥true p ξ [φ₁] [φ₂] | inj₂ y
  = [ (λ x → inj₁ ([ (λ xx → lem-pm φ₁ ¥true xx ξ x [φ₂]) ,
                     (λ xx → lem-subst-d (¥true => ¥false) ¥false φ₁ ξ xx x)
                   ]' (ex-mid ((¥true => ¥false) isSubDisjunct φ₁)))) ,
      (λ x → inj₂ ([ (λ xx → lem-pm φ₂ ¥true xx ξ x [φ₂]) ,
                     (λ xx → lem-subst-d (¥true => ¥false) ¥false φ₂ ξ xx x)
                   ]' (ex-mid ((¥true => ¥false) isSubDisjunct φ₂))))
    ]' [φ₁]
lem-pm (φ₁ && φ₂) ¥true () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ => φ₂) ¥true p ξ [φ₁] [φ₂] | inj₂ y rewrite ¬Tb y = ⊥-elim p
lem-pm (¥ x) ¥true () ξ [φ₁] [φ₂] | inj₂ y

lem-pm φ₁ ¥false p ξ [φ₁] [φ₂] with ex-mid ( (¥false => ¥false) ≡pl φ₁)
```

```
lem-pm φ₁ ¥false p ξ [φ₁] [φ₂] | inj₁ x rewrite Tb x
  = subst (λ k → ⟦ ξ ⊧ k ⟧pl) (sym (lift-≡pl (¥false => ¥false) φ₁ x)) [φ₁] [φ₂]
lem-pm ¥true ¥false () ξ [φ₁] [φ₂] | inj₂ y
lem-pm ¥false ¥false () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ || φ₂) ¥false p ξ (inj₁ x) [φ₂] | inj₂ y
  with ex-mid ((¥false => ¥false) isSubDisjunct φ₁)
lem-pm (φ₁ || φ₂) ¥false p ξ (inj₁ x₁) [φ₂] | inj₂ y | inj₁ x
  = inj₁ (lem-pm φ₁ ¥false x ξ x₁ [φ₂])
lem-pm (φ₁ || φ₂) ¥false p ξ (inj₁ x) [φ₂] | inj₂ y₁ | inj₂ y
  = inj₁ (lem-subst-d (¥false => ¥false) ¥false φ₁ ξ y x)
lem-pm (φ₁ || φ₂) ¥false p ξ (inj₂ y₁) [φ₂] | inj₂ y
  with ex-mid ((¥false => ¥false) isSubDisjunct φ₂)
lem-pm (φ₁ || φ₂) ¥false p ξ (inj₂ y₁) [φ₂] | inj₂ y | inj₁ x
  = inj₂ (lem-pm φ₂ ¥false x ξ y₁ [φ₂])
lem-pm (φ₁ || φ₂) ¥false p ξ (inj₂ y₂) [φ₂] | inj₂ y₁ | inj₂ y
  = inj₂ (lem-subst-d (¥false => ¥false) ¥false φ₂ ξ y y₂)
lem-pm (φ₁ && φ₂) ¥false () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ => φ₂) ¥false p ξ [φ₁] [φ₂] | inj₂ y rewrite ¬Tb y = ⊥-elim p
lem-pm (¥ x) ¥false () ξ [φ₁] [φ₂] | inj₂ y

lem-pm φ₁ (φ₂ || φ₃) p ξ [φ₁] (inj₁ x) with ex-mid (conflict-negleft φ₁ φ₂)
lem-pm φ₁ (φ₂ || φ₃) p ξ [φ₁] (inj₁ x₁) | inj₁ x rewrite Tb x = inj₁ (lem-pm φ₁ φ₂ x ξ [φ₁] x₁)
lem-pm φ₁ (φ₂ || φ₃) p ξ [φ₁] (inj₁ x) | inj₂ y rewrite ¬Tb y = inj₂ x
lem-pm φ₁ (φ₂ || φ₃) p ξ [φ₁] (inj₂ y) with ex-mid (conflict-negleft φ₁ φ₂)
lem-pm φ₁ (φ₂ || φ₃) p ξ [φ₁] (inj₂ y) | inj₁ x rewrite Tb x = inj₂ y
lem-pm φ₁ (φ₂ || φ₃) p ξ [φ₁] (inj₂ y₁) | inj₂ y rewrite ¬Tb y = inj₁ (lem-pm φ₁ φ₃ p ξ [φ₁] y₁)

lem-pm φ₁ (φ₃ && φ₄) p ξ [φ₁] [φ₂] with ex-mid ( ((φ₃ && φ₄) => ¥false) ≡pl φ₁)
lem-pm φ₁ (φ₃ && φ₄) p ξ [φ₁] [φ₂] | inj₁ x rewrite Tb x
  = subst ([_⊧_]pl ξ) (sym (lift-≡pl ((φ₃ && φ₄) => ¥false) φ₁ x)) [φ₁] [φ₂]
lem-pm ¥true (φ₃ && φ₄) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm ¥false (φ₃ && φ₄) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ || φ₂) (φ₃ && φ₄) p ξ [φ₁] [φ₂] | inj₂ y
  = [ (λ x → inj₁ ([ (λ xx → lem-pm φ₁ (φ₃ && φ₄) xx ξ x [φ₂]) ,
                     (λ xx → lem-subst-d ((φ₃ && φ₄) => ¥false) ¥false φ₁ ξ xx x)
                   ]' (ex-mid (((φ₃ && φ₄) => ¥false) isSubDisjunct φ₁)))) ,
      (λ x → inj₂ ([ (λ xx → lem-pm φ₂ (φ₃ && φ₄) xx ξ x [φ₂]) ,
                     (λ xx → lem-subst-d ((φ₃ && φ₄) => ¥false) ¥false φ₂ ξ xx x)
                   ]' (ex-mid (((φ₃ && φ₄) => ¥false) isSubDisjunct φ₂))))
    ]' ([φ₁])
lem-pm (φ₁ && φ₂) (φ₃ && φ₄) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ => φ₂) (φ₃ && φ₄) p ξ [φ₁] [φ₂] | inj₂ y rewrite ¬Tb y = ⊥-elim p
lem-pm (¥ x) (φ₃ && φ₄) () ξ [φ₁] [φ₂] | inj₂ y

lem-pm φ₁ (φ₃ => φ₄) p ξ [φ₁] [φ₂] with ex-mid ( ((φ₃ => φ₄) => ¥false) ≡pl φ₁)
lem-pm φ₁ (φ₃ => φ₄) p ξ [φ₁] [φ₂] | inj₁ x rewrite Tb x
  = subst (λ k → ⟦ ξ ⊧ k ⟧pl) (sym (lift-≡pl ((φ₃ => φ₄) => ¥false) φ₁ x)) [φ₁] [φ₂]
lem-pm ¥true (φ₃ => φ₄) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm ¥false (φ₃ => φ₄) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ || φ₂) (φ₃ => φ₄) p ξ [φ₁] [φ₂] | inj₂ y
  = [ (λ x → inj₁ ([ (λ xx → lem-pm φ₁ (φ₃ => φ₄) xx ξ x [φ₂]) ,
                     (λ xx → lem-subst-d ((φ₃ => φ₄) => ¥false) ¥false φ₁ ξ xx x)
                   ]' (ex-mid (((φ₃ => φ₄) => ¥false) isSubDisjunct φ₁)))) ,
      (λ x → inj₂ ([ (λ xx → lem-pm φ₂ (φ₃ => φ₄) xx ξ x [φ₂]) ,
                     (λ xx → lem-subst-d ((φ₃ => φ₄) => ¥false) ¥false φ₂ ξ xx x)
                   ]' (ex-mid (((φ₃ => φ₄) => ¥false) isSubDisjunct φ₂)))) ]' [φ₁]
lem-pm (φ₁ && φ₂) (φ₃ => φ₄) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ => φ₂) (φ₃ => φ₄) p ξ [φ₁] [φ₂] | inj₂ y rewrite ¬Tb y = ⊥-elim p
lem-pm (¥ x) (φ₃ => φ₄) () ξ [φ₁] [φ₂] | inj₂ y

lem-pm φ₁ (¥ n) p ξ [φ₁] [φ₂] with ex-mid ( ((¥ n) => ¥false) ≡pl φ₁)
lem-pm φ₁ (¥ n) p ξ [φ₁] [φ₂] | inj₁ x rewrite Tb x
  = subst ([_⊧_]pl ξ) (sym (lift-≡pl ((¥ n) => ¥false) φ₁ x)) [φ₁] [φ₂]
lem-pm ¥true (¥ n) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm ¥false (¥ n) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ || φ₂) (¥ n) p ξ [φ₁] [φ₂] | inj₂ y
  = [ (λ x → inj₁ ([ (λ xx → lem-pm φ₁ (¥ n) xx ξ x [φ₂]) ,
                     (λ xx → lem-subst-d ((¥ n) => ¥false) ¥false φ₁ ξ xx x)
                   ]' (ex-mid (((¥ n) => ¥false) isSubDisjunct φ₁)))) ,
      (λ x → inj₂ ([ (λ xx → lem-pm φ₂ (¥ n) xx ξ x [φ₂]) ,
                     (λ xx → lem-subst-d ((¥ n) => ¥false) ¥false φ₂ ξ xx x)
    ]' (ex-mid (((¥ n) => ¥false) isSubDisjunct φ₂)))) ]' ([φ₁])
lem-pm (φ₁ && φ₂) (¥ n) () ξ [φ₁] [φ₂] | inj₂ y
lem-pm (φ₁ => φ₂) (¥ n) p ξ [φ₁] [φ₂] | inj₂ y rewrite ¬Tb y = ⊥-elim p
lem-pm (¥ x) (¥ n) () ξ [φ₁] [φ₂] | inj₂ y
```

```
module Boolean.PL-Formula.RemoveConstants where

open import Data.Sum as Sum
open import Data.Product as Prod

open import Boolean.Formula

open import PropIso

map-or : PL-Formula → PL-Formula → PL-Formula
map-or ¥true  b      = ¥true
map-or ¥false b      = b
map-or a      ¥true  = ¥true
map-or a      ¥false = a
map-or a      b      = a || b

map-and : PL-Formula → PL-Formula → PL-Formula
map-and ¥true  b      = b
map-and ¥false b      = ¥false
map-and a      ¥true  = a
map-and a      ¥false = ¥false
map-and a      b      = a && b

map-iff : PL-Formula → PL-Formula → PL-Formula
map-iff ¥true  b      = b
map-iff ¥false b      = ¥true
map-iff a      ¥true  = ¥true
map-iff a      ¥false = ~ a
map-iff a      b      = a => b

const-removal : PL-Formula → PL-Formula
const-removal = elim-pl ¥true ¥false ¥ map-or map-and map-iff

lem-map-or : ∀ ξ φ ψ → ⟦ ξ ⊨ φ || ψ ⟧pl → ⟦ ξ ⊨ map-or φ ψ ⟧pl
lem-map-or ξ ¥true = λ _ _ → tt
lem-map-or ξ ¥false = λ _ → [ (λ ()) , id ]'
lem-map-or ξ (φ || ψ) = λ {¥true → const tt; ¥false → [ id , (λ ()) ]'; (ψ' || ψ'') → id;
                           (ψ' && ψ'') → id ; (ψ' => ψ'') → id ; (¥ x) → id }
lem-map-or ξ (φ && ψ) = λ {¥true → const tt; ¥false → [ id , (λ ()) ]'; (ψ' || ψ'') → id;
                           (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-or ξ (φ => ψ) = λ {¥true → const tt; ¥false → [ id , (λ ()) ]'; (ψ' || ψ'') → id;
                           (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-or ξ (¥ x) = λ {¥true → const tt; ¥false → [ id , (λ ()) ]'; (ψ' || ψ'') → id;
                        (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}

lem-map-or' : ∀ ξ φ ψ → ⟦ ξ ⊨ map-or φ ψ ⟧pl → ⟦ ξ ⊨ φ || ψ ⟧pl
lem-map-or' ξ ¥true = λ _ → inj₁
lem-map-or' ξ ¥false = λ _ → inj₂
lem-map-or' ξ (φ || φ₁) = λ {¥true → inj₂; ¥false → inj₁; (ψ' || ψ'') → id; (ψ' && ψ'') → id;
                             (ψ' => ψ'') → id; (¥ x) → id}
lem-map-or' ξ (φ && φ₁) = λ {¥true → inj₂; ¥false → inj₁; (ψ' || ψ'') → id; (ψ' && ψ'') → id;
                             (ψ' => ψ'') → id; (¥ x) → id}
lem-map-or' ξ (φ => φ₁) = λ {¥true → inj₂; ¥false → inj₁; (ψ' || ψ'') → id; (ψ' && ψ'') → id;
                             (ψ' => ψ'') → id; (¥ x) → id}
lem-map-or' ξ (¥ x) = λ {¥true → inj₂; ¥false → inj₁; (ψ' || ψ'') → id; (ψ' && ψ'') → id;
                         (ψ' => ψ'') → id; (¥ x) → id}

lem-map-and : ∀ ξ φ ψ → ⟦ ξ ⊨ φ && ψ ⟧pl → ⟦ ξ ⊨ map-and φ ψ ⟧pl
lem-map-and ξ ¥true = λ _ → proj₂
lem-map-and ξ ¥false = λ _ → proj₁
lem-map-and ξ (φ || φ₁) = λ {¥true → proj₁; ¥false → proj₂; (ψ' || ψ'') → id;
                             (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-and ξ (φ && φ₁) = λ {¥true → proj₁; ¥false → proj₂; (ψ' || ψ'') → id;
                             (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-and ξ (φ => φ₁) = λ {¥true → proj₁; ¥false → proj₂; (ψ' || ψ'') → id;
                             (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-and ξ (¥ x) = λ {¥true → proj₁; ¥false → proj₂; (ψ' || ψ'') → id;
                         (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}

lem-map-and' : ∀ ξ φ ψ → ⟦ ξ ⊨ map-and φ ψ ⟧pl → ⟦ ξ ⊨ φ && ψ ⟧pl
lem-map-and' ξ ¥true = λ _ x → _ , x
lem-map-and' ξ ¥false = λ _ → λ ()
lem-map-and' ξ (φ || φ₁) = λ {¥true → λ x → x , _; ¥false → λ (); (ψ' || ψ'') → id;
                              (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-and' ξ (φ && φ₁) = λ {¥true → λ x → x , _; ¥false → λ (); (ψ' || ψ'') → id;
                              (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-and' ξ (φ => φ₁) = λ {¥true → λ x → x , _; ¥false → λ (); (ψ' || ψ'') → id;
                              (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-and' ξ (¥ x) = λ {¥true → λ x₁ → x₁ , _; ¥false → λ (); (ψ' || ψ'') → id;
                          (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}

lem-map-iff : ∀ ξ φ ψ → ⟦ ξ ⊨ φ => ψ ⟧pl → ⟦ ξ ⊨ map-iff φ ψ ⟧pl
```

```
lem-map-iff ξ ¥true = λ _ x → x tt
lem-map-iff ξ ¥false = λ _ _ → tt
lem-map-iff ξ (φ || φ₁) = λ {¥true → λ _ → tt; ¥false → id; (ψ' || ψ'') → id;
                             (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-iff ξ (φ && φ₁) = λ {¥true → λ _ → tt; ¥false → id; (ψ' || ψ'') → id;
                             (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-iff ξ (φ => φ₁) = λ {¥true → λ _ → tt; ¥false → id; (ψ' || ψ'') → id;
                             (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-iff ξ (¥ x) = λ {¥true → λ _ → tt; ¥false → id; (ψ' || ψ'') → id;
                         (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}

lem-map-iff' : ∀ ξ φ ψ → ⟦ ξ ⊨ map-iff φ ψ ⟧pl → ⟦ ξ ⊨ φ => ψ ⟧pl
lem-map-iff' ξ ¥true = λ _ x _ → x
lem-map-iff' ξ ¥false = λ _ _ → λ ()
lem-map-iff' ξ (φ || φ₁) = λ {¥true → λ x _ → x; ¥false → id; (ψ' || ψ'') → id;
                              (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-iff' ξ (φ && φ₁) = λ {¥true → λ x _ → x; ¥false → id; (ψ' || ψ'') → id;
                              (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-iff' ξ (φ => φ₁) = λ {¥true → λ x _ → x; ¥false → id; (ψ' || ψ'') → id;
                              (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}
lem-map-iff' ξ (¥ x) = λ {¥true → λ x₁ _ → x₁; ¥false → id; (ψ' || ψ'') → id;
                          (ψ' && ψ'') → id; (ψ' => ψ'') → id; (¥ x) → id}

mutual
  lem-no-const : ∀ ξ φ → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ const-removal φ ⟧pl
  lem-no-const ξ ¥true p = p
  lem-no-const ξ ¥false p = p
  lem-no-const ξ (φ || ψ) p = lem-map-or ξ (const-removal φ) _ (Sum.map (lem-no-const ξ φ)
                                                                       (lem-no-const ξ ψ) p)
  lem-no-const ξ (φ && ψ) p = lem-map-and ξ (const-removal φ) _ (Prod.map (lem-no-const ξ φ)
                                                                         (lem-no-const ξ ψ) p)
  lem-no-const ξ (φ => ψ) p = lem-map-iff ξ (const-removal φ) _ (lem-no-const ξ ψ ∘ p ∘
                                                                lem-no-const' ξ φ)
  lem-no-const ξ (¥ v) p = p

  lem-no-const' : ∀ ξ φ → ⟦ ξ ⊨ const-removal φ ⟧pl → ⟦ ξ ⊨ φ ⟧pl
  lem-no-const' ξ ¥true p = p
  lem-no-const' ξ ¥false p = p
  lem-no-const' ξ (φ || ψ) p = Sum.map (lem-no-const' ξ φ) (lem-no-const' ξ ψ)
                                       (lem-map-or' ξ (const-removal φ) _ p)
  lem-no-const' ξ (φ && ψ) p = Prod.map (lem-no-const' ξ φ) (lem-no-const' ξ ψ)
                                        (lem-map-and' ξ (const-removal φ) _ p)
  lem-no-const' ξ (φ => ψ) p = lem-no-const' ξ ψ ∘ lem-map-iff' ξ (const-removal φ) _ p ∘
                               lem-no-const ξ φ
  lem-no-const' ξ (¥ v) p = p
```

```
module Boolean.PL-Formula.Distribute where

open import Boolean.Formula

open import Data.Product as Prod
open import Data.Sum as Sum

open import Function

-- dist-clause-V : CLAUSE → CNF → CNF
dist-clause-V : PL-Formula → PL-Formula → PL-Formula
dist-clause-V cl (φ && ψ) = dist-clause-V cl φ && dist-clause-V cl ψ
dist-clause-V cl φ = cl || φ

-- dist-V : CNF → CNF → CNF
dist-V : PL-Formula → PL-Formula → PL-Formula
dist-V (φ && φ') ψ = dist-V φ ψ && dist-V φ' ψ
dist-V cl ψ = dist-clause-V cl ψ

mkdist : PL-Formula → PL-Formula
mkdist (φ || ψ) = dist-V (mkdist φ) (mkdist ψ)
mkdist (φ && ψ) = mkdist φ && mkdist ψ
mkdist φ = φ

lem-dist-clause-V : ∀ ξ cl φ → ⟦ ξ ⊨ cl || φ ⟧pl → ⟦ ξ ⊨ dist-clause-V cl φ ⟧pl
lem-dist-clause-V ξ cl ¥true p = p
lem-dist-clause-V ξ cl ¥false p = p
lem-dist-clause-V ξ cl (y || y') p = p
lem-dist-clause-V ξ cl (y && y') p
  = [ (λ x → (lem-dist-clause-V ξ cl y (inj₁ x)) , (lem-dist-clause-V ξ cl y' (inj₁ x)))
    , (λ x → lem-dist-clause-V ξ cl y (inj₂ (proj₁ x))
             , lem-dist-clause-V ξ cl y' (inj₂ (proj₂ x))) ]' p
lem-dist-clause-V ξ cl (y => y') p = p
lem-dist-clause-V ξ cl (¥ y) p = p

lem-dist-clause-V' : ∀ ξ cl φ → ⟦ ξ ⊨ dist-clause-V cl φ ⟧pl → ⟦ ξ ⊨ cl || φ ⟧pl
lem-dist-clause-V' ξ cl ¥true p = p
lem-dist-clause-V' ξ cl ¥false p = p
lem-dist-clause-V' ξ cl (y || y') p = p
lem-dist-clause-V' ξ cl (y && y') p
  = uncurry (λ [y] [y'] → [ inj₁ , (λ [y]' → [ inj₁ , (λ [y']' → inj₂ ([y]' , [y']'))
                                              ]' (lem-dist-clause-V' ξ cl y' [y']))
                          ]' (lem-dist-clause-V' ξ cl y [y])) p
lem-dist-clause-V' ξ cl (y => y') p = p
lem-dist-clause-V' ξ cl (¥ y) p = p

lem-dist-V : ∀ ξ φ ψ → ⟦ ξ ⊨ φ || ψ ⟧pl → ⟦ ξ ⊨ dist-V φ ψ ⟧pl
lem-dist-V ξ ¥true = lem-dist-clause-V ξ ¥true
lem-dist-V ξ ¥false = lem-dist-clause-V ξ ¥false
lem-dist-V ξ (y || y') = lem-dist-clause-V ξ (y || y')
lem-dist-V ξ (y && y') =
  λ ψ → [ Prod.map (lem-dist-V ξ y ψ ∘ inj₁) (lem-dist-V ξ y' ψ ∘ inj₁) ,
          (λ x → (lem-dist-V ξ y ψ (inj₂ x)) , (lem-dist-V ξ y' ψ (inj₂ x))) ]'
lem-dist-V ξ (y => y') = lem-dist-clause-V ξ (y => y')
lem-dist-V ξ (¥ y) = lem-dist-clause-V ξ (¥ y)

lem-dist-V' : ∀ ξ φ ψ → ⟦ ξ ⊨ dist-V φ ψ ⟧pl → ⟦ ξ ⊨ φ || ψ ⟧pl
lem-dist-V' ξ ¥true = lem-dist-clause-V' ξ ¥true
lem-dist-V' ξ ¥false = lem-dist-clause-V' ξ ¥false
lem-dist-V' ξ (y || y') = lem-dist-clause-V' ξ (y || y')
lem-dist-V' ξ (y && y')
  = λ ψ → uncurry (λ [y]' [y']' → [ (\ [y] → [ (λ [y'] → inj₁ ([y] , [y'])) , inj₂
                                              ]' (lem-dist-V' ξ y' ψ [y']')) , inj₂
                                  ]' (lem-dist-V' ξ y ψ [y]'))
lem-dist-V' ξ (y => y') = lem-dist-clause-V' ξ (y => y')
lem-dist-V' ξ (¥ y) = lem-dist-clause-V' ξ (¥ y)

lem-mkdist : ∀ ξ φ → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ mkdist φ ⟧pl
lem-mkdist ξ ¥true p = p
lem-mkdist ξ ¥false p = p
lem-mkdist ξ (y || y') p = lem-dist-V ξ (mkdist y) (mkdist y')
                                      (Sum.map (lem-mkdist ξ y) (lem-mkdist ξ y') p)
lem-mkdist ξ (y && y') p = Prod.map (lem-mkdist ξ y) (lem-mkdist ξ y') p
lem-mkdist ξ (y => y') p = p
lem-mkdist ξ (¥ y) p = p
```

```
lem-mkdist' : ∀ ξ φ → ⟦ ξ ⊧ mkdist φ ⟧pl → ⟦ ξ ⊧ φ ⟧pl
lem-mkdist' ξ ¥true p = p
lem-mkdist' ξ ¥false p = p
lem-mkdist' ξ (y || y') p = Sum.map (lem-mkdist' ξ y) (lem-mkdist' ξ y')
                                    (lem-dist-∨' ξ (mkdist y) (mkdist y') p)
lem-mkdist' ξ (y && y') p = Prod.map (lem-mkdist' ξ y) (lem-mkdist' ξ y') p
lem-mkdist' ξ (y => y') p = p
lem-mkdist' ξ (¥ y) p = p
```

```
module Boolean.PL-Formula.Equivalence where

open import Data.Nat
open import Data.Bool
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.List hiding ([_];_++_)

open import Relation.Binary.PropositionalEquality hiding ([_])

open import PropIso hiding (_$_) renaming (_==_ to _=='_)

open import Boolean.Formula

private
  mutual
    data PL-List* : Set where
      [_] : PL-Formula* → PL-List*
      _∷_ : PL-Formula* → PL-List* → PL-List*

    data PL-Formula* : Set where
      $true $false : PL-Formula*
      or* and* : PL-List* → PL-Formula*
      imp* : PL-Formula* → PL-Formula* → PL-Formula*
      $ : ℕ → PL-Formula*

  _++_ : PL-List* → PL-List* → PL-List*
  [ x ] ++ Δ = x ∷ Δ
  (x ∷ Γ) ++ Δ = x ∷ (Γ ++ Δ)

  ⟦_⊨_⟧pl* : (ξ : Env) → PL-Formula* → Set
  ⟦ ξ ⊨ $true ⟧pl* = ⊤
  ⟦ ξ ⊨ $false ⟧pl* = ⊥
  ⟦ ξ ⊨ and* [ x ] ⟧pl* = ⟦ ξ ⊨ x ⟧pl*
  ⟦ ξ ⊨ and* (φ ∷ φs) ⟧pl* = ⟦ ξ ⊨ φ ⟧pl* × ⟦ ξ ⊨ and* φs ⟧pl*
  ⟦ ξ ⊨ or* [ x ] ⟧pl* = ⟦ ξ ⊨ x ⟧pl*
  ⟦ ξ ⊨ or* (φ ∷ φs) ⟧pl* = ⟦ ξ ⊨ φ ⟧pl* ⊎ ⟦ ξ ⊨ or* φs ⟧pl*
  ⟦ ξ ⊨ imp* φ ψ ⟧pl* = ⟦ ξ ⊨ φ ⟧pl* → ⟦ ξ ⊨ ψ ⟧pl*
  ⟦ ξ ⊨ $ v ⟧pl* = T (ξ v)


  flatten-or' : PL-Formula* → PL-Formula* → PL-List*
  flatten-or' (or* φs) (or* ψs) = φs ++ ψs
  flatten-or' (or* φs) ψ = φs ++ [ ψ ]
  flatten-or' φ (or* ψs) = φ ∷ ψs
  flatten-or' φ ψ = φ ∷ [ ψ ]

  flatten-or : PL-Formula* → PL-Formula* → PL-Formula*
  flatten-or a b = or* (flatten-or' a b)

  flatten-and' : PL-Formula* → PL-Formula* → PL-List*
  flatten-and' (and* φs) (and* ψs) = φs ++ ψs
  flatten-and' (and* φs) ψ = φs ++ [ ψ ]
  flatten-and' φ (and* ψs) = φ ∷ ψs
  flatten-and' φ ψ = φ ∷ [ ψ ]

  flatten-and : PL-Formula* → PL-Formula* → PL-Formula*
  flatten-and a b = and* (flatten-and' a b)

flatten : PL-Formula → PL-Formula*
flatten = elim-pl $true $false $ flatten-or flatten-and imp*

private
  mutual
    _==_ : PL-Formula* → PL-Formula* → Bool
    $true     == $true     = true
    $false    == $false    = true
    or* x     == or* y     = x ==[] y
    and* x    == and* y    = x ==[] y
    imp* φ φ₁ == imp* ψ ψ₁ = φ == ψ ∧ φ₁ == ψ₁
    $ x       == $ y       = x ==' y
    x         == y         = false

    _==[]_ : PL-List* → PL-List* → Bool
    [ x ] ==[] [ y ]       = x == y
    (x ∷ xs) ==[] (y ∷ ys) = x == y ∧ xs ==[] ys
    _ ==[] _               = false

  _∈**_ : PL-Formula* → PL-List* → Bool
  φ ∈** [ x ] = φ == x
  φ ∈** (ψ ∷ ψs) = φ == ψ ∨ φ ∈** ψs

  mutual
```

```
  sym-==' : (φ ψ : PL-Formula*) → T (φ == ψ) → T (ψ == φ)
  sym-==' $true       $true       eq = eq
  sym-==' $true       $false      eq = eq
  sym-==' $true       (or* x)     eq = eq
  sym-==' $true       (and* x)    eq = eq
  sym-==' $true       (imp* ψ ψ₁) eq = eq
  sym-==' $true       ($ x)       eq = eq
  sym-==' $false      $true       eq = eq
  sym-==' $false      $false      eq = eq
  sym-==' $false      (or* x)     eq = eq
  sym-==' $false      (and* x)    eq = eq
  sym-==' $false      (imp* ψ ψ₁) eq = eq
  sym-==' $false      ($ x)       eq = eq
  sym-==' (or* x)     $true       eq = eq
  sym-==' (or* x)     $false      eq = eq
  sym-==' (or* x)     (or* x₁)    eq = sym-==[] x x₁ eq
  sym-==' (or* x)     (and* x₁)   eq = eq
  sym-==' (or* x)     (imp* ψ ψ₁) eq = eq
  sym-==' (or* x)     ($ x₁)      eq = eq
  sym-==' (and* x)    $true       eq = eq
  sym-==' (and* x)    $false      eq = eq
  sym-==' (and* x)    (or* x₁)    eq = eq
  sym-==' (and* x)    (and* x₁)   eq = sym-==[] x x₁ eq
  sym-==' (and* x)    (imp* ψ ψ₁) eq = eq
  sym-==' (and* x)    ($ x₁)      eq = eq
  sym-==' (imp* φ φ₁) $true       eq = eq
  sym-==' (imp* φ φ₁) $false      eq = eq
  sym-==' (imp* φ φ₁) (or* x)     eq = eq
  sym-==' (imp* φ φ₁) (and* x)    eq = eq
  sym-==' (imp* φ φ₁) (imp* ψ ψ₁) eq = f∧g (sym-==' φ ψ) (sym-==' φ₁ ψ₁) eq
  sym-==' (imp* φ φ₁) ($ x)       eq = eq
  sym-==' ($ x)       $true       eq = eq
  sym-==' ($ x)       $false      eq = eq
  sym-==' ($ x)       (or* x₁)    eq = eq
  sym-==' ($ x)       (and* x₁)   eq = eq
  sym-==' ($ x)       (imp* ψ ψ₁) eq = eq
  sym-==' ($ x)       ($ x₁)      eq = sym-== x x₁ eq

  sym-==[] : (Γ Δ : PL-List*) → T (Γ ==[] Δ) → T (Δ ==[] Γ)
  sym-==[] [ x ]    [ y ]    p = sym-==' x y p
  sym-==[] [ x ]    (x' :: Δ) p = p
  sym-==[] (x :: Γ) [ y ]    p = p
  sym-==[] (γ :: Γ) (δ :: Δ) p = f∧g (sym-==' γ δ) (sym-==[] Γ Δ) p

mutual
  eq : PL-Formula* → PL-Formula* → Bool
  eq (and* y)     (or* x)      = alleq* x (and* y) ∨ alleq* y (or* x)
  eq (or* x)      (and* y)     = alleq* y (or* x) ∨ alleq* x (and* y)
  eq (or* x)      (or* y)      = (y ⊆pl* x ∧ x ⊆pl* y) ∨ alleq* x (or* y) ∨ alleq* y (or* x)
  eq (or* x)      ψ            = alleq* x ψ
  eq (and* x)     (and* y)     = (y ⊆pl* x ∧ x ⊆pl* y) ∨ alleq* x (and* y) ∨ alleq* y (and* x)
  eq (and* x)     ψ            = alleq* x ψ
  eq φ            (or* x)      = alleq* x φ
  eq φ            (and* x)     = alleq* x φ
  eq (imp* φ φ₁) (imp* ψ ψ₁) = eq φ ψ ∧ eq φ₁ ψ₁
  eq φ            ψ            = φ == ψ

  alleq* : PL-List* → PL-Formula* → Bool
  alleq* [ x ]    φ = eq x φ
  alleq* (x :: l) φ = eq x φ ∧ alleq* l φ

  _⊆pl*_ : PL-List* → PL-List* → Bool
  [ x ]    ⊆pl* y  = x ∈* y
  (x :: xs) ⊆pl* ys = x ∈* ys ∧ xs ⊆pl* ys

  _∈*_ : PL-Formula* → PL-List* → Bool
  φ ∈* [ x ]    = eq φ x
  φ ∈* (ψ :: ψs) = eq φ ψ ∨ φ ∈* ψs

sym-eq : (φ ψ : PL-Formula*) → T (eq φ ψ) → T (eq ψ φ)
sym-eq $true $true eq = eq
sym-eq $true $false eq = eq
sym-eq $true (or* x) eq = eq
sym-eq $true (and* x) eq = eq
sym-eq $true (imp* ψ ψ₁) eq = eq
sym-eq $true ($ x) eq = eq
sym-eq $false $true eq = eq
sym-eq $false $false eq = eq
sym-eq $false (or* x) eq = eq
sym-eq $false (and* x) eq = eq
sym-eq $false (imp* ψ ψ₁) eq = eq
sym-eq $false ($ x) eq = eq
sym-eq (or* x) $true eq = eq
sym-eq (or* x) $false eq = eq
```

```
  sym-eq (or* x) (or* x₁) eq = fV∧g (∧-swap _ (x ⊆pl* x₁))
                                    (V-swap (alleq* x (or* x₁)) _) eq
  sym-eq (or* x) (and* x₁) eq = V-swap (alleq* x₁ (or* x)) _ eq
  sym-eq (or* x) (imp* ψ ψ₁) eq = eq
  sym-eq (or* x) ($ x₁) eq = eq
  sym-eq (and* x) $true eq = eq
  sym-eq (and* x) $false eq = eq
  sym-eq (and* x) (or* x₁) eq = V-swap (alleq* x₁ (and* x)) _ eq
  sym-eq (and* x) (and* x₁) eq = fV∧g (∧-swap _ (x ⊆pl* x₁))
                                      (V-swap (alleq* x (and* x₁)) _) eq
  sym-eq (and* x) (imp* ψ ψ₁) eq = eq
  sym-eq (and* x) ($ x₁) eq = eq
  sym-eq (imp* φ φ₁) $true eq = eq
  sym-eq (imp* φ φ₁) $false eq = eq
  sym-eq (imp* φ φ₁) (or* x) eq = eq
  sym-eq (imp* φ φ₁) (and* x) eq = eq
  sym-eq (imp* φ φ₁) (imp* ψ ψ₁) eq' = f∧g (sym-eq φ ψ) (sym-eq φ₁ ψ₁) eq'
  sym-eq (imp* φ φ₁) ($ x) eq = eq
  sym-eq ($ x) $true eq = eq
  sym-eq ($ x) $false eq = eq
  sym-eq ($ x) (or* x₁) eq = eq
  sym-eq ($ x) (and* x₁) eq = eq
  sym-eq ($ x) (imp* ψ ψ₁) eq = eq
  sym-eq ($ x) ($ x₁) eq = sym-== x x₁ eq

  ord-⊆pl* : ∀ φ Γ Δ → T (Γ ⊆pl* Δ) → T (Γ ⊆pl* (φ :: Δ))
  ord-⊆pl* φ [ γ ] Δ p = V-intror (eq γ φ) (γ ∈* Δ) p
  ord-⊆pl* φ (γ :: Γ) Δ p = f∧g (V-intror (eq γ φ) (γ ∈* Δ)) (ord-⊆pl* φ Γ Δ) p

  mutual
    id-⊆pl* : ∀ Γ → T (Γ ⊆pl* Γ)
    id-⊆pl* [ γ ] = id-eq γ
    id-⊆pl* (γ :: Γ) = ∧-intro _ _ (V-introl _ _ (id-eq γ)) (ord-⊆pl* γ Γ Γ (id-⊆pl* Γ))

    id-eq : ∀ φ → T (eq φ φ)
    id-eq $true = tt
    id-eq $false = tt
    id-eq (or* y) = V-introl _ _ (∧-intro _ _ (id-⊆pl* y) (id-⊆pl* y))
    id-eq (and* y) = V-introl _ _ (∧-intro _ _ (id-⊆pl* y) (id-⊆pl* y))
    id-eq (imp* y y') = ∧-intro (eq y y) _ (id-eq y) (id-eq y')
    id-eq ($ y) = id-== y

    subst-eq : ∀ ξ φ ψ → T (eq φ ψ) → ⟦ ξ ⊨ φ ⟧pl* → ⟦ ξ ⊨ ψ ⟧pl*
    subst-eq ξ $true $true eqp p = p
    subst-eq ξ $true $false eqp p = ⊥-elim eqp
    subst-eq ξ $true (or* x) eqp p = subst-alleq-or ξ x $true tt eqp
    subst-eq ξ $true (and* x) eqp p = subst-alleq-and ξ x $true tt eqp
    subst-eq ξ $true (imp* ψ ψ₁) eqp p = ⊥-elim eqp
    subst-eq ξ $true ($ x) eqp p = ⊥-elim eqp
    subst-eq ξ $false ψ eqp p = ⊥-elim p
    subst-eq ξ (or* x) $true eqp p = subst-alleq-or' ξ x $true p eqp
    subst-eq ξ (or* x) $false eqp p = subst-alleq-or' ξ x $false p eqp
    subst-eq ξ (or* x) (or* x₁) eqp p =
      V-elim (λ k → subst-eq-or ξ x x₁ p (∧-elimr (x₁ ⊆pl* x) k))
                                (V-elim (subst-alleq-or' ξ x (or* x₁) p)
                                        (subst-alleq-or ξ x₁ (or* x) p)) eqp
    subst-eq ξ (or* x) (and* x₁) eqp p = V-elim (subst-alleq-and ξ x₁ (or* x) p)
                                                (subst-alleq-or' ξ x (and* x₁) p) eqp
    subst-eq ξ (or* x) (imp* ψ ψ₁) eqp p = subst-alleq-or' ξ x (imp* ψ ψ₁) p eqp
    subst-eq ξ (or* x) ($ x₁) eqp p = subst-alleq-or' ξ x ($ x₁) p eqp
    subst-eq ξ (and* x) $true eqp p = subst-alleq-and' ξ x $true p eqp
    subst-eq ξ (and* x) $false eqp p = subst-alleq-and' ξ x $false p eqp
    subst-eq ξ (and* x) (or* x₁) eqp p = V-elim (subst-alleq-or ξ x₁ (and* x) p)
                                                (subst-alleq-and' ξ x (or* x₁) p) eqp
    subst-eq ξ (and* x) (and* x₁) eqp p =
      V-elim (λ k → subst-eq-and ξ x x₁ p (∧-eliml k))
                                 (V-elim (subst-alleq-and' ξ x (and* x₁) p)
                                         (subst-alleq-and ξ x₁ (and* x) p)) eqp
    subst-eq ξ (and* x) (imp* ψ ψ₁) eqp p = subst-alleq-and' ξ x (imp* ψ ψ₁) p eqp
    subst-eq ξ (and* x) ($ x₁) eqp p = subst-alleq-and' ξ x ($ x₁) p eqp
    subst-eq ξ (imp* φ φ₁) $true eqp p = ⊥-elim eqp
    subst-eq ξ (imp* φ φ₁) $false eqp p = ⊥-elim eqp
    subst-eq ξ (imp* φ φ₁) (or* x) eqp p = subst-alleq-or ξ x (imp* φ φ₁) p eqp
    subst-eq ξ (imp* φ φ₁) (and* x) eqp p = subst-alleq-and ξ x (imp* φ φ₁) p eqp
    subst-eq ξ (imp* φ φ₁) (imp* ψ ψ₁) eqp p =
      subst-eq ξ φ₁ ψ₁ (∧-elimr (eq φ ψ) eqp) ∘ p ∘ subst-eq ξ ψ φ (sym-eq φ ψ (∧-eliml eqp))
    subst-eq ξ (imp* φ φ₁) ($ x) eqp p = ⊥-elim eqp
    subst-eq ξ ($ x) $true eqp p = ⊥-elim eqp
    subst-eq ξ ($ x) $false eqp p = ⊥-elim eqp
    subst-eq ξ ($ x) (or* x₁) eqp p = subst-alleq-or ξ x₁ ($ x) p eqp
    subst-eq ξ ($ x) (and* x₁) eqp p = subst-alleq-and ξ x₁ ($ x) p eqp
    subst-eq ξ ($ x) (imp* ψ ψ₁) eqp p = ⊥-elim eqp
    subst-eq ξ ($ x) ($ x₁) eqp p rewrite lift-== x x₁ eqp = p
```

```
  subst-alleq-and : ∀ ξ Γ φ → ⟦ ξ ⊨ φ ⟧pl* → T (alleq* Γ φ) → ⟦ ξ ⊨ and* Γ ⟧pl*
  subst-alleq-and ξ [ x ] φ [φ] aeq = subst-eq ξ φ x (sym-eq x φ aeq) [φ]
  subst-alleq-and ξ (x :: Γ) φ [φ] aeq = ∧-elim (λ a b → (subst-eq ξ φ x (sym-eq x φ a) [φ])
                                                , (subst-alleq-and ξ Γ φ [φ] b)) aeq

  subst-alleq-or : ∀ ξ Γ φ → ⟦ ξ ⊨ φ ⟧pl* → T (alleq* Γ φ) → ⟦ ξ ⊨ or* Γ ⟧pl*
  subst-alleq-or ξ [ x ] φ [φ] aeq = subst-eq ξ φ x (sym-eq x φ aeq) [φ]
  subst-alleq-or ξ (x :: Γ) φ [φ] aeq = inj₁ (subst-eq ξ φ x (sym-eq x φ (∧-eliml aeq)) [φ])

  subst-alleq-or' : ∀ ξ Γ φ → ⟦ ξ ⊨ or* Γ ⟧pl* → T (alleq* Γ φ) → ⟦ ξ ⊨ φ ⟧pl*
  subst-alleq-or' ξ [ x ] φ [φ] aeq = subst-eq ξ x φ aeq [φ]
  subst-alleq-or' ξ (x :: Γ) φ (inj₁ x₁) aeq = subst-eq ξ x φ (∧-eliml aeq) x₁
  subst-alleq-or' ξ (x :: Γ) φ (inj₂ y) aeq = subst-alleq-or' ξ Γ φ y (∧-elimr (eq x φ) aeq)

  subst-alleq-and' : ∀ ξ Γ φ → ⟦ ξ ⊨ and* Γ ⟧pl* → T (alleq* Γ φ) → ⟦ ξ ⊨ φ ⟧pl*
  subst-alleq-and' ξ [ x ] φ [φ] aeq = subst-eq ξ x φ aeq [φ]
  subst-alleq-and' ξ (x :: Γ) φ [φ] aeq = subst-eq ξ x φ (∧-eliml aeq) (proj₁ [φ])

  subst-∈-and : ∀ ξ Γ φ → T (φ ∈* Γ) → ⟦ ξ ⊨ and* Γ ⟧pl* → ⟦ ξ ⊨ φ ⟧pl*
  subst-∈-and ξ [ x ] φ p q = subst-eq ξ x φ (sym-eq φ x p) q
  subst-∈-and ξ (x :: Γ) φ p q = ∨-elim (λ k → subst-eq ξ x φ (sym-eq φ x k) (proj₁ q))
                                        (λ k → subst-∈-and ξ Γ φ k (proj₂ q)) p

  subst-eq-and : ∀ ξ Γ Δ → ⟦ ξ ⊨ and* Γ ⟧pl* → T (Δ ⊆pl* Γ) → ⟦ ξ ⊨ and* Δ ⟧pl*
  subst-eq-and ξ Γ [ x ] p f = subst-∈-and ξ Γ x f p
  subst-eq-and ξ Γ (x :: Δ) p f = (subst-∈-and ξ Γ x (∧-eliml f) p)
                                , (subst-eq-and ξ Γ Δ p (∧-elimr (x ∈* Γ) f))

  subst-∈-or : ∀ ξ Γ φ → T (φ ∈* Γ) → ⟦ ξ ⊨ φ ⟧pl* → ⟦ ξ ⊨ or* Γ ⟧pl*
  subst-∈-or ξ [ x ] φ p q = subst-eq ξ φ x p q
  subst-∈-or ξ (x :: Γ) φ p q = ∨-elim (λ k → inj₁ (subst-eq ξ φ x k q))
                                       (λ k → inj₂ (subst-∈-or ξ Γ φ k q)) p

  subst-eq-or : ∀ ξ Γ Δ → ⟦ ξ ⊨ or* Γ ⟧pl* → T (Γ ⊆pl* Δ) → ⟦ ξ ⊨ or* Δ ⟧pl*
  subst-eq-or ξ [ x ] Δ p q = subst-∈-or ξ Δ x q p
  subst-eq-or ξ (x :: Γ) Δ (inj₁ x₁) q = subst-∈-or ξ Δ x (∧-eliml q) x₁
  subst-eq-or ξ (x :: Γ) Δ (inj₂ y) q = subst-eq-or ξ Γ Δ y (∧-elimr (x ∈* Δ) q)

lem-or-elim : ∀ ξ φ ψ
            → ⟦ ξ ⊨ or* (flatten-or' φ ψ) ⟧pl* → ⟦ ξ ⊨ φ ⟧pl* ⊎ ⟦ ξ ⊨ ψ ⟧pl*
lem-or-elim ξ $true $true p = p
lem-or-elim ξ $true $false p = p
lem-or-elim ξ $true (or* x) p = p
lem-or-elim ξ $true (and* x) p = p
lem-or-elim ξ $true (imp* ψ ψ₁) p = p
lem-or-elim ξ $true ($ x) p = p
lem-or-elim ξ $false $true p = p
lem-or-elim ξ $false $false p = p
lem-or-elim ξ $false (or* x) p = p
lem-or-elim ξ $false (and* x) p = p
lem-or-elim ξ $false (imp* ψ ψ₁) p = p
lem-or-elim ξ $false ($ x) p = p
lem-or-elim ξ (or* [ x ]) $true p = p
lem-or-elim ξ (or* [ x ]) $false p = p
lem-or-elim ξ (or* [ x ]) (or* x₁) p = p
lem-or-elim ξ (or* [ x ]) (and* x₁) p = p
lem-or-elim ξ (or* [ x ]) (imp* ψ ψ₁) p = p
lem-or-elim ξ (or* [ x ]) ($ x₁) p = p
lem-or-elim ξ (or* (x :: xs)) $true p =
  [ inj₁ ∘ inj₁ , (λ x' → Sum.map inj₂ id (lem-or-elim ξ (or* xs) $true x')) ]' p
lem-or-elim ξ (or* (x :: xs)) $false p =
  [ inj₁ ∘ inj₁ , (λ x' → Sum.map inj₂ id (lem-or-elim ξ (or* xs) $false x')) ]' p
lem-or-elim ξ (or* (x :: xs)) (or* x₁) p =
  [ inj₁ ∘ inj₁ , (λ x' → Sum.map inj₂ id (lem-or-elim ξ (or* xs) (or* x₁) x')) ]' p
lem-or-elim ξ (or* (x :: xs)) (and* x₁) p =
  [ inj₁ ∘ inj₁ , (λ x' → Sum.map inj₂ id (lem-or-elim ξ (or* xs) (and* x₁) x')) ]' p
lem-or-elim ξ (or* (x :: xs)) (imp* ψ ψ₁) p =
  [ inj₁ ∘ inj₁ , (λ x' → Sum.map inj₂ id (lem-or-elim ξ (or* xs) (imp* ψ ψ₁) x')) ]' p
lem-or-elim ξ (or* (x :: xs)) ($ x₁) p =
  [ inj₁ ∘ inj₁ , (λ x' → Sum.map inj₂ id (lem-or-elim ξ (or* xs) ($ x₁) x')) ]' p
lem-or-elim ξ (and* x) $true p = p
lem-or-elim ξ (and* x) $false p = p
lem-or-elim ξ (and* x) (or* x₁) p = p
lem-or-elim ξ (and* x) (and* x₁) p = p
lem-or-elim ξ (and* x) (imp* ψ ψ₁) p = p
lem-or-elim ξ (and* x) ($ x₁) p = p
lem-or-elim ξ (imp* φ φ₁) $true p = p
lem-or-elim ξ (imp* φ φ₁) $false p = p
lem-or-elim ξ (imp* φ φ₁) (or* x) p = p
lem-or-elim ξ (imp* φ φ₁) (and* x) p = p
lem-or-elim ξ (imp* φ φ₁) (imp* ψ ψ₁) p = p
lem-or-elim ξ (imp* φ φ₁) ($ x) p = p
lem-or-elim ξ ($ x) $true p = p
lem-or-elim ξ ($ x) $false p = p
```

```
  lem-or-elim ξ ($ x) (or* x₁) p = p
  lem-or-elim ξ ($ x) (and* x₁) p = p
  lem-or-elim ξ ($ x) (imp* ψ ψ₁) p = p
  lem-or-elim ξ ($ x) ($ x₁) p = p

  lem-or-elim' : ∀ ξ φ ψ
               → ⟦ ξ ⊨ φ ⟧pl* ⊎ ⟦ ξ ⊨ ψ ⟧pl* → ⟦ ξ ⊨ or* (flatten-or' φ ψ) ⟧pl*
  lem-or-elim' ξ $true $true p = p
  lem-or-elim' ξ $true $false p = p
  lem-or-elim' ξ $true (or* x) p = p
  lem-or-elim' ξ $true (and* x) p = p
  lem-or-elim' ξ $true (imp* ψ ψ₁) p = p
  lem-or-elim' ξ $true ($ x) p = p
  lem-or-elim' ξ $false $true p = p
  lem-or-elim' ξ $false $false p = p
  lem-or-elim' ξ $false (or* x) p = p
  lem-or-elim' ξ $false (and* x) p = p
  lem-or-elim' ξ $false (imp* ψ ψ₁) p = p
  lem-or-elim' ξ $false ($ x) p = p
  lem-or-elim' ξ (or* [ x ]) $true p = p
  lem-or-elim' ξ (or* [ x ]) $false p = p
  lem-or-elim' ξ (or* [ x ]) (or* x₁) p = p
  lem-or-elim' ξ (or* [ x ]) (and* x₁) p = p
  lem-or-elim' ξ (or* [ x ]) (imp* ψ ψ₁) p = p
  lem-or-elim' ξ (or* [ x ]) ($ x₁) p = p
  lem-or-elim' ξ (or* (x ∷ x₁)) $true p = Sum.map id (lem-or-elim' ξ (or* x₁) $true) (lem-⊎ p)
  lem-or-elim' ξ (or* (x ∷ x₁)) $false p = Sum.map id (lem-or-elim' ξ (or* x₁) $false) (lem-⊎ p)
  lem-or-elim' ξ (or* (x ∷ x₁)) (or* x₂) p = Sum.map id (lem-or-elim' ξ (or* x₁) (or* x₂))
                                                        (lem-⊎ p)
  lem-or-elim' ξ (or* (x ∷ x₁)) (and* x₂) p = Sum.map id (lem-or-elim' ξ (or* x₁) (and* x₂))
                                                         (lem-⊎ p)
  lem-or-elim' ξ (or* (x ∷ x₁)) (imp* ψ ψ₁) p = Sum.map id (lem-or-elim' ξ (or* x₁) (imp* ψ ψ₁))
                                                          (lem-⊎ p)
  lem-or-elim' ξ (or* (x ∷ x₁)) ($ x₂) p = Sum.map id (lem-or-elim' ξ (or* x₁) ($ x₂)) (lem-⊎ p)
  lem-or-elim' ξ (and* x) $true p = p
  lem-or-elim' ξ (and* x) $false p = p
  lem-or-elim' ξ (and* x) (or* x₁) p = p
  lem-or-elim' ξ (and* x) (and* x₁) p = p
  lem-or-elim' ξ (and* x) (imp* ψ ψ₁) p = p
  lem-or-elim' ξ (and* x) ($ x₁) p = p
  lem-or-elim' ξ (imp* φ φ₁) $true p = p
  lem-or-elim' ξ (imp* φ φ₁) $false p = p
  lem-or-elim' ξ (imp* φ φ₁) (or* x) p = p
  lem-or-elim' ξ (imp* φ φ₁) (and* x) p = p
  lem-or-elim' ξ (imp* φ φ₁) (imp* ψ ψ₁) p = p
  lem-or-elim' ξ (imp* φ φ₁) ($ x) p = p
  lem-or-elim' ξ ($ x) $true p = p
  lem-or-elim' ξ ($ x) $false p = p
  lem-or-elim' ξ ($ x) (or* x₁) p = p
  lem-or-elim' ξ ($ x) (and* x₁) p = p
  lem-or-elim' ξ ($ x) (imp* ψ ψ₁) p = p
  lem-or-elim' ξ ($ x) ($ x₁) p = p

  lem-and-elim : ∀ ξ φ ψ
               → ⟦ ξ ⊨ and* (flatten-and' φ ψ) ⟧pl* → ⟦ ξ ⊨ φ ⟧pl* × ⟦ ξ ⊨ ψ ⟧pl*
  lem-and-elim ξ $true $true p = p
  lem-and-elim ξ $true $false p = p
  lem-and-elim ξ $true (or* x) p = p
  lem-and-elim ξ $true (and* x) p = p
  lem-and-elim ξ $true (imp* ψ ψ₁) p = p
  lem-and-elim ξ $true ($ x) p = p
  lem-and-elim ξ $false $true p = p
  lem-and-elim ξ $false $false p = p
  lem-and-elim ξ $false (or* x) p = p
  lem-and-elim ξ $false (and* x) p = p
  lem-and-elim ξ $false (imp* ψ ψ₁) p = p
  lem-and-elim ξ $false ($ x) p = p
  lem-and-elim ξ (or* x) $true p = p
  lem-and-elim ξ (or* x) $false p = p
  lem-and-elim ξ (or* x) (or* x₁) p = p
  lem-and-elim ξ (or* x) (and* x₁) p = p
  lem-and-elim ξ (or* x) (imp* ψ ψ₁) p = p
  lem-and-elim ξ (or* x) ($ x₁) p = p
  lem-and-elim ξ (and* [ x ]) $true p = p
  lem-and-elim ξ (and* [ x ]) $false p = p
  lem-and-elim ξ (and* [ x ]) (or* x₁) p = p
  lem-and-elim ξ (and* [ x ]) (and* x₁) p = p
  lem-and-elim ξ (and* [ x ]) (imp* ψ ψ₁) p = p
  lem-and-elim ξ (and* [ x ]) ($ x₁) p = p
  lem-and-elim ξ (and* (x ∷ xs)) $true p =
    Prod.map (_,_ (proj₁ p)) id (lem-and-elim ξ (and* xs) $true (proj₂ p))
  lem-and-elim ξ (and* (x ∷ xs)) $false p =
    Prod.map (_,_ (proj₁ p)) id (lem-and-elim ξ (and* xs) $false (proj₂ p))
  lem-and-elim ξ (and* (x ∷ xs)) (or* x₁) p =
```

```
    Prod.map (_,_ (proj₁ p)) id (lem-and-elim ξ (and* xs) (or* x₁) (proj₂ p))
  lem-and-elim ξ (and* (x :: xs)) (and* x₁) p =
    Prod.map (_,_ (proj₁ p)) id (lem-and-elim ξ (and* xs) (and* x₁) (proj₂ p))
  lem-and-elim ξ (and* (x :: xs)) (imp* ψ ψ₁) p =
    Prod.map (_,_ (proj₁ p)) id (lem-and-elim ξ (and* xs) (imp* ψ ψ₁) (proj₂ p))
  lem-and-elim ξ (and* (x :: xs)) ($ x₁) p =
    Prod.map (_,_ (proj₁ p)) id (lem-and-elim ξ (and* xs) ($ x₁) (proj₂ p))
  lem-and-elim ξ (imp* φ φ₁) $true p = p
  lem-and-elim ξ (imp* φ φ₁) $false p = p
  lem-and-elim ξ (imp* φ φ₁) (or* x) p = p
  lem-and-elim ξ (imp* φ φ₁) (and* x) p = p
  lem-and-elim ξ (imp* φ φ₁) (imp* ψ ψ₁) p = p
  lem-and-elim ξ (imp* φ φ₁) ($ x) p = p
  lem-and-elim ξ ($ x) $true p = p
  lem-and-elim ξ ($ x) $false p = p
  lem-and-elim ξ ($ x) (or* x₁) p = p
  lem-and-elim ξ ($ x) (and* x₁) p = p
  lem-and-elim ξ ($ x) (imp* ψ ψ₁) p = p
  lem-and-elim ξ ($ x) ($ x₁) p = p

  lem-and-elim' : ∀ ξ φ ψ
                → ⟦ ξ ⊨ φ ⟧pl* × ⟦ ξ ⊨ ψ ⟧pl* → ⟦ ξ ⊨ and* (flatten-and' φ ψ) ⟧pl*
  lem-and-elim' ξ $true $true p = p
  lem-and-elim' ξ $true $false p = p
  lem-and-elim' ξ $true (or* x) p = p
  lem-and-elim' ξ $true (and* x) p = p
  lem-and-elim' ξ $true (imp* ψ ψ₁) p = p
  lem-and-elim' ξ $true ($ x) p = p
  lem-and-elim' ξ $false $true p = p
  lem-and-elim' ξ $false $false p = p
  lem-and-elim' ξ $false (or* x) p = p
  lem-and-elim' ξ $false (and* x) p = p
  lem-and-elim' ξ $false (imp* ψ ψ₁) p = p
  lem-and-elim' ξ $false ($ x) p = p
  lem-and-elim' ξ (or* x) $true p = p
  lem-and-elim' ξ (or* x) $false p = p
  lem-and-elim' ξ (or* x) (or* x₁) p = p
  lem-and-elim' ξ (or* x) (and* x₁) p = p
  lem-and-elim' ξ (or* x) (imp* ψ ψ₁) p = p
  lem-and-elim' ξ (or* x) ($ x₁) p = p
  lem-and-elim' ξ (and* [ x ]) $true p = p
  lem-and-elim' ξ (and* [ x ]) $false p = p
  lem-and-elim' ξ (and* [ x ]) (or* x₁) p = p
  lem-and-elim' ξ (and* [ x ]) (and* x₁) p = p
  lem-and-elim' ξ (and* [ x ]) (imp* ψ ψ₁) p = p
  lem-and-elim' ξ (and* [ x ]) ($ x₁) p = p
  lem-and-elim' ξ (and* (x :: x₁)) $true p =
    Prod.map id (λ k → lem-and-elim' ξ (and* x₁) $true (k , proj₂ p)) (proj₁ p)
  lem-and-elim' ξ (and* (x :: x₁)) $false p =
    Prod.map id (λ k → lem-and-elim' ξ (and* x₁) $false (k , proj₂ p)) (proj₁ p)
  lem-and-elim' ξ (and* (x :: x₁)) (or* x₂) p =
    Prod.map id (λ k → lem-and-elim' ξ (and* x₁) (or* x₂) (k , proj₂ p)) (proj₁ p)
  lem-and-elim' ξ (and* (x :: x₁)) (and* x₂) p =
    Prod.map id (λ k → lem-and-elim' ξ (and* x₁) (and* x₂) (k , proj₂ p)) (proj₁ p)
  lem-and-elim' ξ (and* (x :: x₁)) (imp* ψ ψ₁) p =
    Prod.map id (λ k → lem-and-elim' ξ (and* x₁) (imp* ψ ψ₁) (k , proj₂ p)) (proj₁ p)
  lem-and-elim' ξ (and* (x :: x₁)) ($ x₂) p =
    Prod.map id (λ k → lem-and-elim' ξ (and* x₁) ($ x₂) (k , proj₂ p)) (proj₁ p)
  lem-and-elim' ξ (imp* φ φ₁) $true p = p
  lem-and-elim' ξ (imp* φ φ₁) $false p = p
  lem-and-elim' ξ (imp* φ φ₁) (or* x) p = p
  lem-and-elim' ξ (imp* φ φ₁) (and* x) p = p
  lem-and-elim' ξ (imp* φ φ₁) (imp* ψ ψ₁) p = p
  lem-and-elim' ξ (imp* φ φ₁) ($ x) p = p
  lem-and-elim' ξ ($ x) $true p = p
  lem-and-elim' ξ ($ x) $false p = p
  lem-and-elim' ξ ($ x) (or* x₁) p = p
  lem-and-elim' ξ ($ x) (and* x₁) p = p
  lem-and-elim' ξ ($ x) (imp* ψ ψ₁) p = p
  lem-and-elim' ξ ($ x) ($ x₁) p = p

  mutual
    lem-flatten : ∀ ξ φ → ⟦ ξ ⊨ flatten φ ⟧pl* → ⟦ ξ ⊨ φ ⟧pl
    lem-flatten ξ ¥true p = tt
    lem-flatten ξ ¥false p = p
    lem-flatten ξ (y || y') p = Sum.map (lem-flatten ξ y) (lem-flatten ξ y')
                                        (lem-or-elim ξ (flatten y) (flatten y') p)
    lem-flatten ξ (y && y') p = Prod.map (lem-flatten ξ y) (lem-flatten ξ y')
                                         (lem-and-elim ξ (flatten y) (flatten y') p)
    lem-flatten ξ (y => y') p = \ x → lem-flatten ξ y' (p (lem-flatten' ξ y x))
    lem-flatten ξ (¥ y) p = p

    lem-flatten' : ∀ ξ φ → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ flatten φ ⟧pl*
    lem-flatten' ξ ¥true p = tt
```

```
    lem-flatten' ξ ¥false p = p
    lem-flatten' ξ (y || y') p = lem-or-elim' ξ (flatten y) (flatten y')
                                              (Sum.map (lem-flatten' ξ y)
                                                       (lem-flatten' ξ y') p)
    lem-flatten' ξ (y && y') p = lem-and-elim' ξ (flatten y) (flatten y')
                                               (Prod.map (lem-flatten' ξ y)
                                                         (lem-flatten' ξ y') p)
    lem-flatten' ξ (y => y') p = λ x → lem-flatten' ξ y' (p (lem-flatten ξ y x))
    lem-flatten' ξ (¥ y) p = p

_==pl_ : (φ ψ : PL-Formula) → Bool
φ ==pl ψ = eq (flatten φ) (flatten ψ)

private
  lift-flatten : ∀ φ ψ → T (φ ==pl ψ) → ∀ ξ → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ ψ ⟧pl
  lift-flatten φ ψ φ=ψ ξ [φ] = lem-flatten ξ ψ (subst-eq ξ (flatten φ) (flatten ψ)
                                                         φ=ψ (lem-flatten' ξ φ [φ]))

lift-==pl : ∀ φ ψ → T (φ ==pl ψ)
          → ∀ ξ → ((⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ ψ ⟧pl) × (⟦ ξ ⊨ ψ ⟧pl → ⟦ ξ ⊨ φ ⟧pl))
lift-==pl φ ψ φ=ψ ξ = (lift-flatten φ ψ φ=ψ ξ)
                    , (lift-flatten ψ φ (sym-eq (flatten φ) (flatten ψ) φ=ψ) ξ)

lift-==pl→ : ∀ φ ψ → T (φ ==pl ψ) → ∀ ξ → ⟦ ξ ⊨ φ ⟧pl → ⟦ ξ ⊨ ψ ⟧pl
lift-==pl→ φ ψ p ξ = proj₁ (lift-==pl φ ψ p ξ)

lift-==pl← : ∀ φ ψ → T (φ ==pl ψ) → ∀ ξ → ⟦ ξ ⊨ ψ ⟧pl → ⟦ ξ ⊨ φ ⟧pl
lift-==pl← φ ψ p ξ = proj₂ (lift-==pl φ ψ p ξ)
```

```
module Boolean.PL-Formula.DropEquivalence where

open import Boolean.Formula
open import Boolean.PL-Formula.Equivalence

open import Data.Bool
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Nat

open import PropIso

open import Relation.Binary.PropositionalEquality

mutual
  lem-extend-env : ∀ ξ n φ b → T (not ((¥ n) isSubFormula φ))
                 → ⟦ ξ ⊧ φ ⟧pl → ⟦ envupdate ξ n b ⊧ φ ⟧pl
  lem-extend-env ξ n ¥true b n∉φ [φ] = [φ]
  lem-extend-env ξ n ¥false b n∉φ [φ] = [φ]
  lem-extend-env ξ n (y || y') b n∉φ [φ]
    = let π : T _ × T _
          π = lem-bool-∧-s (not (¥ n isSubFormula y)) _ (demorg1 (¥ n isSubFormula y) _ n∉φ)
      in Sum.map (lem-extend-env ξ n y b (proj₁ π)) (lem-extend-env ξ n y' b (proj₂ π)) [φ]
  lem-extend-env ξ n (y && y') b n∉φ [φ]
    = let π : T _ × T _
          π = lem-bool-∧-s (not (¥ n isSubFormula y)) _ (demorg1 (¥ n isSubFormula y) _ n∉φ)
      in Prod.map (lem-extend-env ξ n y b (proj₁ π)) (lem-extend-env ξ n y' b (proj₂ π)) [φ]
  lem-extend-env ξ n (y => y') b n∉φ [φ]
    = let π : T _ × T _
          π = lem-bool-∧-s (not (¥ n isSubFormula y)) _ (demorg1 (¥ n isSubFormula y) _ n∉φ)
      in lem-extend-env ξ n y' b (proj₂ π) ∘ [φ] ∘ lem-extend-env' ξ n y b (proj₁ π)
  lem-extend-env ξ n (¥ y) b n∉φ [φ] with n == y
  ...| true = ⊥-elim n∉φ
  ...| false = [φ]

  lem-extend-env' : ∀ ξ n φ b → T (not ((¥ n) isSubFormula φ))
                  → ⟦ envupdate ξ n b ⊧ φ ⟧pl → ⟦ ξ ⊧ φ ⟧pl
  lem-extend-env' ξ n ¥true b n∉φ [φ] = [φ]
  lem-extend-env' ξ n ¥false b n∉φ [φ] = [φ]
  lem-extend-env' ξ n (y || y') b n∉φ [φ]
    = let π : T _ × T _
          π = lem-bool-∧-s (not (¥ n isSubFormula y)) _ (demorg1 (¥ n isSubFormula y) _ n∉φ)
      in Sum.map (lem-extend-env' ξ n y b (proj₁ π)) (lem-extend-env' ξ n y' b (proj₂ π)) [φ]
  lem-extend-env' ξ n (y && y') b n∉φ [φ]
    = let π : T _ × T _
          π = lem-bool-∧-s (not (¥ n isSubFormula y)) _ (demorg1 (¥ n isSubFormula y) _ n∉φ)
      in Prod.map (lem-extend-env' ξ n y b (proj₁ π)) (lem-extend-env' ξ n y' b (proj₂ π)) [φ]
  lem-extend-env' ξ n (y => y') b n∉φ [φ]
    = let π : T _ × T _
          π = lem-bool-∧-s (not (¥ n isSubFormula y)) _ (demorg1 (¥ n isSubFormula y) _ n∉φ)
      in lem-extend-env' ξ n y' b (proj₂ π) ∘ [φ] ∘ lem-extend-env ξ n y b (proj₁ π)
  lem-extend-env' ξ n (¥ y) b n∉φ [φ] with n == y
  ...| true = ⊥-elim n∉φ
  ...| false = [φ]

private
  lem : ∀ n φ ψ
      → T (not ((¥ n) isSubFormula φ))
      → T (not ((¥ n) isSubFormula ψ))
      → Σ[ ξ ∶ Env ] ⟦ ξ ⊧ ψ ⟧pl
      → Σ[ ξ ∶ Env ] ⟦ ξ ⊧ ((¥ n) <=> φ) && ψ ⟧pl
  lem n φ ψ n∉φ n∉ψ (ξ , [ψ]) = envupdate ξ n (eval-pl ξ φ)
                               , ((lem-extend-env ξ n φ (eval-pl ξ φ) n∉φ ∘ lem-eval' ξ φ ∘
                                     subst T (lem-envupdate ξ n (eval-pl ξ φ)))
                                 , subst T (sym (lem-envupdate ξ n (eval-pl ξ φ))) ∘ lem-eval ξ φ ∘
                                     lem-extend-env' ξ n φ (eval-pl ξ φ) n∉φ)
                               , lem-extend-env ξ n ψ (eval-pl ξ φ) n∉ψ [ψ]

  lem' : ∀ n φ ψ
       → T (not ((¥ n) isSubFormula φ))
       → T (not ((¥ n) isSubFormula ψ))
       → Σ[ ξ ∶ Env ] ⟦ ξ ⊧ ((¥ n) <=> φ) && ψ ⟧pl
       → Σ[ ξ ∶ Env ] ⟦ ξ ⊧ ψ ⟧pl
  lem' n φ ψ n∉φ n∉ψ (ξ , [n<=>φ&&ψ]) = ξ , (proj₂ [n<=>φ&&ψ])

∨-false : ∀ b → b ∨ false ≡ b
∨-false true = refl
∨-false false = refl

¬⊂φ : ∀ φ n → (¥ n isSubFormula φ) ≡ (¥ n isSubFormula (~ φ))
¬⊂φ φ n rewrite ∨-false (¥ n isSubFormula φ) = refl

r : ∀ φ ψ n
  → T (not ((¥ n) isSubFormula φ))
  → T (not ((¥ n) isSubFormula ψ))
  → (∀ ξ → ⟦ ξ ⊧ (¥ n <=> ψ) => φ ⟧pl)
```

```
    → ∀ ξ → ⟦ ξ ⊨ φ ⟧pl
  r φ ψ n n∉φ n∉ψ f ξ =
    [ id , (λ [~φ] → ⊥-elim $ proj₂ (proj₂ (lem n ψ (~ φ) n∉ψ (subst (λ b → T (not b)) (¬⊂φ φ n) n∉φ)
                                                                (_ , [~φ])))
                                    (f _ (proj₁ (proj₂ (lem n ψ (~ φ) n∉ψ
                                                               (subst (λ b → T (not b)) (¬⊂φ φ n) n∉φ)
                                                               (_ , [~φ])))))))
    ]' (ex-mid-pl ξ φ)

  r' : ∀ φ ψ ρ n
    → T (not ((¥ n) isSubFormula φ))
    → T (not ((¥ n) isSubFormula ψ))
    → T (not ((¥ n) isSubFormula ρ))
    → (∀ ξ → ⟦ ξ ⊨ ((¥ n <=> ψ) && ρ) => φ ⟧pl)
    → ∀ ξ → ⟦ ξ ⊨ ρ => φ ⟧pl
  r' φ ψ ρ n n∉φ n∉ψ n∉ρ f = r (ρ => φ) ψ n (demorg2 (¥ n isSubFormula ρ) _
                               (∧-intro (not _) _ n∉ρ n∉φ)) n∉ψ (λ ξ' → curry (f ξ'))
```

```
module RDM.RailYard where

open import Data.List
open import Data.List.Inhabitence
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Empty

open import RDM.fixedtrains

open import Relation.Binary.PropositionalEquality
open import Relation.Decidable

open import Function

record SignalLocation (Segment : Set) (Connected : Segment → Segment → Set) : Set where
  constructor
    sigloc
  field
    facing    : Segment
    trailing  : Segment
    connected : Connected facing trailing

record PhysicalLayout : Set₁ where
  field
    Segment     : Set
    Signal      : Set
    connections : Segment → List Segment

  Connected : Segment → Segment → Set
  Connected s₁ s₂ = s₂ isin connections s₁

  field
    signalLocation : Signal → SignalLocation Segment Connected

record ControlTableEntry (PL : PhysicalLayout) : Set where
  Segment = PhysicalLayout.Segment PL
  Signal  = PhysicalLayout.Signal PL
  field
    start         : Signal
    segments      : List Segment
    normalpoints  : List Segment
    reversepoints : List Segment
    facing        : List Segment

head : ∀ {A} → (l : List A) → Σ[ a ∶ A ] (a isin l) → A
head []      p = ⊥-elim $ proj₂ p
head (l ∷ _) p = l

last : ∀ {A} → (l : List A) → Σ[ a ∶ A ] (a isin l) → A
last []            p = ⊥-elim $ proj₂ p
last (l ∷ [])      p = l
last (l ∷ l' ∷ ls) p = last (l' ∷ ls) (l' , inj₁ refl)

record ControlTable (PL : PhysicalLayout) : Set₁ where
  field
    Route           : Set
    DecidableRoute : Decidable (_≡_ {A = Route})
    entries         : Route → ControlTableEntry PL
    connections     : Route → List Route

  Connected : Route → Route → Set
  Connected rt₁ rt₂ = rt₂ isin connections rt₁

  SegInRoute : PhysicalLayout.Segment PL → Route → Set
  SegInRoute ts rt = ts isin (ControlTableEntry.segments $ entries rt)

  FacingInRoute : PhysicalLayout.Segment PL → Route → Set
  FacingInRoute ts rt = ts isin (ControlTableEntry.facing $ entries rt)

  field
    NonEmptyRoutes : ∀ rt → Σ[ ts ∶ PhysicalLayout.Segment PL ] (SegInRoute ts rt)
    WellFormed : ∀ rt₁ rt₂ rt₃ → Connected rt₁ rt₂ → Connected rt₃ rt₂
               → Σ[ ts ∶ PhysicalLayout.Segment PL ] (SegInRoute ts rt₁ × SegInRoute ts rt₃)

  routeHead : ∀ rt → PhysicalLayout.Segment PL
  routeHead rt = head (ControlTableEntry.segments $ entries rt) (NonEmptyRoutes rt)

  routeLast : ∀ rt → PhysicalLayout.Segment PL
  routeLast rt = last (ControlTableEntry.segments $ entries rt) (NonEmptyRoutes rt)
```

```
  field
    RoutesConnected : ∀ rt₁ rt₂
                    → Connected rt₁ rt₂
                    → (SignalLocation.facing $ PhysicalLayout.signalLocation PL
                              (ControlTableEntry.start $ entries rt₂)) ≡ routeLast rt₁
                    × (SignalLocation.trailing $ PhysicalLayout.signalLocation PL
                              (ControlTableEntry.start $ entries rt₂)) ≡ routeHead rt₂

toLayout : (PL : PhysicalLayout)
         → ControlTable PL
         → Σ[ TrainID ∶ Set ] (Decidable (_≡_ {A = TrainID})) → Layout
toLayout pl ct tid = record {
                       Segment          = PhysicalLayout.Segment pl;
                       Train            = proj₁ tid;
                       DecidableTrain   = proj₂ tid;
                       Route            = ControlTable.Route ct;
                       DecidableRoute   = ControlTable.DecidableRoute ct;
                       RouteConnected   = ControlTable.Connected ct;
                       SegInRoute       = ControlTable.SegInRoute ct;
                       FacingInRoute    = ControlTable.FacingInRoute ct;
                       WellFormedRoutes = ControlTable.WellFormed ct}

record ControlTableSemantics {PL : PhysicalLayout} (C : ControlTable PL) (A : Set) : Set where
  field
    rtSet       : ControlTable.Route     C  → A
    segNormal   : PhysicalLayout.Segment PL → A
    segReverse  : PhysicalLayout.Segment PL → A
    segLock     : PhysicalLayout.Segment PL → A
    segOccupied : PhysicalLayout.Segment PL → A
    sigProceed  : PhysicalLayout.Signal  PL → A
```

```
module RDM.fixedtrains where

open import Data.Nat hiding (_<_)
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Fin hiding (_<_;_+_)
open import Data.Bool
open import Data.Unit
open import Data.Empty

open import Relation.Decidable
open import Relation.Binary.PropositionalEquality as PEQ

open import PropIso

Θ = ℕ

¬sym : {A : Set} {a b : A} → (a ≠ b) → b ≠ a
¬sym p = p ∘ sym

data Aspect : Set where
  Proceed Stop : Aspect

lemaspect : Proceed ≠ Stop
lemaspect ()

lemaspect' : Stop ≠ Proceed
lemaspect' ()

aspeq : (a a' : Aspect) → a ≡ a' ⊎ a ≠ a'
aspeq Proceed Proceed = inj₁ refl
aspeq Proceed Stop    = inj₂ (λ ())
aspeq Stop    Proceed = inj₂ (λ ())
aspeq Stop    Stop    = inj₁ refl

data Locking : Set where
  Locked Unlocked : Locking

lemlocking : Locked ≠ Unlocked
lemlocking ()

lemlocking' : Unlocked ≠ Locked
lemlocking' ()

lockeq : (a a' : Locking) → a ≡ a' ⊎ a ≠ a'
lockeq Locked   Locked   = inj₁ refl
lockeq Locked   Unlocked = inj₂ (λ ())
lockeq Unlocked Locked   = inj₂ (λ ())
lockeq Unlocked Unlocked = inj₁ refl

record Layout : Set₁ where
  field
    Segment : Set
    Train : Set
    DecidableTrain : Decidable (_≡_ {A = Train})
    Route : Set
    DecidableRoute : Decidable (_≡_ {A = Route})
    RouteConnected : (s₁ s₂ : Route) → Set

    SegInRoute : Segment → Route → Set
    FacingInRoute : (s : Segment) → (rt : Route) → Set

    WellFormedRoutes : ∀ rt₁ rt₂ rt₃
                     → RouteConnected rt₁ rt₂
                     → RouteConnected rt₃ rt₂
                     → Σ[ ts ∶ Segment ] (SegInRoute ts rt₁ × SegInRoute ts rt₃)

-- all things that depend on time single time here
open Layout
record LayoutState (l : Layout) : Set where
  field
    trainRoute   : Train l → Route l
    signalAspect : Route l → Aspect
    locked       : Segment l → Locking

-- signalling princples dependent on time here
open LayoutState
record Railway (l : Layout) : Set where
  field
    layoutState : Θ → LayoutState l

    -- trains only move between connected routes with proceed aspects
    CorrectTrains-Route : (t : Θ)
                        → (tr : Train l)
                        → trainRoute (layoutState t) tr ≡ trainRoute (layoutState (ℕ.suc t)) tr
                          ⊎ (RouteConnected l (trainRoute (layoutState t) tr)
                                              (trainRoute (layoutState (ℕ.suc t)) tr))
```

```
                              × (signalAspect (layoutState t)
                                              (trainRoute (layoutState (ℕ.suc t)) tr) ≡ Proceed)

    Principle-SignalsGuard : (t : Θ)
                           → (tr : Train l)
                           → (ts : Segment l)
                           → (p : SegInRoute l ts (trainRoute (layoutState t) tr))
                           → (rt : Route l)
                           → (p : SegInRoute l ts rt)
                           → signalAspect (layoutState t) rt ≡ Stop

    Principle-OpposingSignals : (t : Θ)
                              → (rt₁ rt₂ : Route l)
                              → (ts : Segment l)
                              → (rt₁≠rt₂ : rt₁ ≠ rt₂)
                              → (p : SegInRoute l ts rt₁)
                              → (q : SegInRoute l ts rt₂)
                              → signalAspect (layoutState t) rt₁ ≡ Stop
                                ⊎ signalAspect (layoutState t) rt₂ ≡ Stop

    Principle-ProceedLocked : (t : Θ)
                            → (rt : Route l)
                            → signalAspect (layoutState t) rt ≡ Proceed
                            → (ts : Segment l)
                            → (q : SegInRoute l ts rt)
                            → FacingInRoute l ts rt
                            → locked (layoutState (t)) ts ≡ Locked

    Principle-TrainLocked : (t : Θ)
                          → (tr : Train l)
                          → (ts : Segment l)
                          → SegInRoute l ts (trainRoute (layoutState (suc t)) tr)
                          → locked (layoutState t) ts ≡ Locked
                          → locked (layoutState (suc t)) ts ≡ Locked

  CorrectTrains-RouteMoved : (t : Θ)
                           → (tr : Train l)
                           → trainRoute (layoutState t) tr ≠ trainRoute (layoutState (ℕ.suc t)) tr
                           → (RouteConnected l (trainRoute (layoutState t) tr)
                                               (trainRoute (layoutState (ℕ.suc t)) tr))
                              × (signalAspect (layoutState t)
                                              (trainRoute (layoutState (ℕ.suc t)) tr) ≡ Proceed)
  CorrectTrains-RouteMoved t tr neq = [ ⊥-elim ∘ neq , id ]′ (CorrectTrains-Route t tr)

open Railway
record FacingPointLock {l : Layout} (r : Railway l) : Set where
  attime : ℕ → LayoutState l
  attime = layoutState r

  Safety' : (t : Θ) → (tr : Train l) → Set
  Safety' t tr = ∀ ts → (q : SegInRoute l ts (trainRoute (attime t) tr))
                      → FacingInRoute l ts (trainRoute (attime t) tr)
                      → locked (attime t) ts ≡ Locked

  Safety : (t : Θ) → Set
  Safety t = (tr : Train l) → Safety' t tr

  field InitialSafe : Safety 0

  private
    Stationary : Θ → Train l → Set
    Stationary t tr = trainRoute (attime t) tr ≡ trainRoute (attime (suc t)) tr
    NotStationary : Θ → Train l → Set
    NotStationary t tr = trainRoute (attime t) tr ≠ trainRoute (attime (suc t)) tr

    CaseTrainSameRoute : ∀ t tr → Safety' t tr → Stationary t tr → Safety' (suc t) tr
    CaseTrainSameRoute t tr ih eq ts p q rewrite (sym eq)
      = Principle-TrainLocked r t tr ts (subst (SegInRoute l ts) eq p) (ih ts p q)

    CaseTrainNewRoute : ∀ t tr → NotStationary t tr → Safety' (suc t) tr
    CaseTrainNewRoute t tr eq ts segin facing
      = Principle-TrainLocked r t tr ts segin
            (Principle-ProceedLocked r t (trainRoute (attime (suc t)) tr)
                                     (proj₂ (CorrectTrains-RouteMoved r t tr eq)) ts segin facing)

  AlwaysSafe : (t : Θ) → Safety t
  AlwaysSafe 0 tr = InitialSafe tr
  AlwaysSafe (suc t) tr = elim-Dec _≡_ (DecidableRoute l (trainRoute (attime t) tr)
                                                         (trainRoute (attime (suc t)) tr))
                                       (CaseTrainSameRoute t tr (AlwaysSafe t tr))
                                       (CaseTrainNewRoute t tr)

record TrainsDontCrash {l : Layout} (r : Railway l) : Set where
  attime : ℕ → LayoutState l
  attime = layoutState r

  Safety' : (t : Θ) → (tr₁ tr₂ : Train l) → Set
  Safety' t tr₁ tr₂ = ∀ ts → SegInRoute l ts (trainRoute (attime t) tr₁)
```

```
                        → ¬ SegInRoute l ts (trainRoute (attime t) tr₂)

  Safety : (t : ℕ) → Set
  Safety t = (tr₁ tr₂ : Train l) → (tr₁ ≡ tr₂ ⊎ Safety' t tr₁ tr₂)

  field InitialSafe : Safety 0

  private
    Stationary : ℕ → Train l → Set
    Stationary t tr = trainRoute (attime t) tr ≡ trainRoute (attime (suc t)) tr
    NotStationary : ℕ → Train l → Set
    NotStationary t tr = trainRoute (attime t) tr ≢ trainRoute (attime (suc t)) tr

    sym-Safety' : ∀ t tr₁ tr₂ → Safety' t tr₁ tr₂ → Safety' t tr₂ tr₁
    sym-Safety' t tr₁ tr₂ p ts tsin1 tsin2 = p ts tsin2 tsin1

    CaseTrainsStationary : ∀ t tr₁ tr₂ → Stationary t tr₁ → Stationary t tr₂ → Safety' t tr₁ tr₂
                         → Safety' (suc t) tr₁ tr₂
    CaseTrainsStationary t tr₁ tr₂ eq₁ eq₂ rewrite eq₁ | eq₂ = id

    CaseOneMoving : ∀ t tr₁ tr₂ → Stationary t tr₁ → NotStationary t tr₂ → Safety' (suc t) tr₁ tr₂
    CaseOneMoving t tr₁ tr₂ eq₁ eq₂ ts tsin₁ tsin₂ rewrite (sym eq₁)
      = lemaspect (trans (sym (proj₂ (CorrectTrains-RouteMoved r t tr₂ eq₂)))
                         (Principle-SignalsGuard r t tr₁ ts tsin₁ _ tsin₂))

    CaseBothMovingDiffRoute : ∀ t tr₁ tr₂
                            → NotStationary t tr₁
                            → NotStationary t tr₂
                            → tr₁ ≢ tr₂
                            → trainRoute (layoutState r (suc t)) tr₁ ≢
                                trainRoute (layoutState r (suc t)) tr₂
                            → Safety' (suc t) tr₁ tr₂
    CaseBothMovingDiffRoute t tr₁ tr₂ eq₁ eq₂ tr₁≢tr₂ eq₃ ts tsin₁ tsin₂
      = [ (λ eq₄ → lemaspect (trans (sym (proj₂ (CorrectTrains-RouteMoved r t tr₁ eq₁))) eq₄)) ,
          (λ eq₄ → lemaspect (trans (sym (proj₂ (CorrectTrains-RouteMoved r t tr₂ eq₂))) eq₄))
        ]' (Principle-OpposingSignals r t (trainRoute (attime (suc t)) tr₁)
                                          (trainRoute (attime (suc t)) tr₂)
                                          ts eq₃ tsin₁ tsin₂)

    CaseBothMoving : ∀ t tr₁ tr₂
                   → NotStationary t tr₁
                   → NotStationary t tr₂
                   → Safety' t tr₁ tr₂
                   → tr₁ ≢ tr₂
                   → Safety' (suc t) tr₁ tr₂
    CaseBothMoving t tr₁ tr₂ eq₁ eq₂ ih tr₁≢tr₂
      = elim-Dec _≡_
          (DecidableRoute l (trainRoute (attime (suc t)) tr₁)
                            (trainRoute (attime (suc t)) tr₂))
          (λ eq₃ → ⊥-elim $ ih (proj₁ (x eq₃)) (proj₁ (proj₂ (x eq₃))) (proj₂ (proj₂ (x eq₃))))
          (CaseBothMovingDiffRoute t tr₁ tr₂ eq₁ eq₂ tr₁≢tr₂)
      where
        x : _ → Σ[ ts ∶ Segment l ] ((SegInRoute l ts (trainRoute (attime t) tr₁))
                                       × (SegInRoute l ts (trainRoute (attime t) tr₂)))
        x eq₃ = WellFormedRoutes l _ (trainRoute (attime (suc t)) tr₁) _
                                     (proj₁ (CorrectTrains-RouteMoved r t tr₁ eq₁))
                                     (subst (RouteConnected l (trainRoute (layoutState r t) tr₂))
                                            (sym eq₃)
                                            (proj₁ (CorrectTrains-RouteMoved r t tr₂ eq₂)))

    CaseNotEqualTrains : ∀ t tr₁ tr₂ → Safety' t tr₁ tr₂ → tr₁ ≢ tr₂ → Safety' (suc t) tr₁ tr₂
    CaseNotEqualTrains t tr₁ tr₂ ih
      = elim-Dec _≡_ (DecidableRoute l (trainRoute (attime t) tr₁)
                                       (trainRoute (attime (suc t)) tr₁))
                 (elim-Dec _≡_
                    (DecidableRoute l (trainRoute (attime t) tr₂)
                                      (trainRoute (attime (suc t)) tr₂))
                    (λ eq1 eq2 _ → CaseTrainsStationary t tr₁ tr₂ eq2 eq1 ih)
                    (λ eq1 eq2 _ → CaseOneMoving t tr₁ tr₂ eq2 eq1))
                 (elim-Dec _≡_
                    (DecidableRoute l (trainRoute (attime t) tr₂)
                                      (trainRoute (attime (suc t)) tr₂))
                    (λ eq1 eq2 _ → sym-Safety' (suc t) tr₂ tr₁ (CaseOneMoving t tr₂ tr₁ eq1 eq2))
                    (λ eq1 eq2 → CaseBothMoving t tr₁ tr₂ eq2 eq1 ih))

    CaseNotEqualTrains' : ∀ t tr₁ tr₂
                        → (tr₁ ≡ tr₂) ⊎ Safety' t tr₁ tr₂ → tr₁ ≢ tr₂ → Safety' (suc t) tr₁ tr₂
    CaseNotEqualTrains' t tr₁ tr₂ ih tr₁≢tr₂
      = CaseNotEqualTrains t tr₁ tr₂ ([ ⊥-elim ∘ tr₁≢tr₂ , id ]' ih) tr₁≢tr₂

  AlwaysSafe : (t : ℕ) → Safety t
  AlwaysSafe 0       tr₁ tr₂ = InitialSafe tr₁ tr₂
  AlwaysSafe (suc n) tr₁ tr₂ = elim-Dec _≡_ (DecidableTrain l tr₁ tr₂) inj₁
                                    (inj₂ ∘ CaseNotEqualTrains' n tr₁ tr₂ (AlwaysSafe n tr₁ tr₂))
```

```
module Ladder.Core where

open import Data.Nat hiding (_<_)
open import Data.Bool
open import Data.List
open import Data.Product as Prod

open import TransitionSystem

open import Boolean.SatSolver
open import Boolean.Formula

open import Relation.Binary.PropositionalEquality

open import PropIso

open import Function

record Ladder : Set₁ where
  constructor
    ladder
  field
    statevars : ℕ
    inputvars : ℕ
    rungs : List (ℕ × PL-Formula)
    initialstate : List (ℕ × Bool)
    inp-correct : PL-Formula

  State : Set
  State = Σ[ l ∶ List Bool ] (length l ≡ statevars)

  Input : Set
  Input = Σ[ m ∶ Σ[ l ∶ List Bool ] (length l ≡ inputvars) ] (⟦ mkenv (proj₁ m) ⊧ inp-correct ⟧pl)

open Ladder

map-proj : ∀ {A C : Set} {B : A → Set} → List (Σ A (λ x → B x × C)) → List (A × C)
map-proj []       = []
map-proj (x ∷ xs) = Prod.map id proj₂ x ∷ map-proj xs

mkinitial-aux : List (ℕ × Bool) → PL-Formula
mkinitial-aux []            = ¥true
mkinitial-aux (a ∷ [])      = ((¥ ∘ proj₁) a) <=> injbool (proj₂ a)
mkinitial-aux (a ∷ a' ∷ as) = ((¥ ∘ proj₁) a) <=> injbool (proj₂ a) && mkinitial-aux  (a' ∷ as)

mkinitial : Ladder → PL-Formula
mkinitial l = mkinitial-aux (initialstate l)

mkInitialRel : (l : Ladder) → State l → Set
mkInitialRel l init = ⟦ mkenv (proj₁ init) ⊧ mkinitial l ⟧pl

apply-¥-pl : PL-Formula → (ℕ → ℕ) → PL-Formula
apply-¥-pl φ f = elim-pl ¥true ¥false (¥ ∘ f) _||_ _&&_ _=>_ φ

mymap : ℕ → List ℕ → ℕ → ℕ
mymap s []       n = n
mymap s (a ∷ as) n with a == n
mymap s (a ∷ as) n | true  = s + a
mymap s (a ∷ as) n | false = mymap s as n

mktrans-aux : ℕ → List ℕ → List (ℕ × PL-Formula) → PL-Formula
mktrans-aux n v []             = ¥true
mktrans-aux n v ((a , b) ∷ as) = (¥ (n + a) <=> apply-¥-pl b (mymap n v)) &&
                                 mktrans-aux n (a ∷ v) as

mktrans : Ladder → PL-Formula
mktrans l = mktrans-aux (statevars l + inputvars l) [] (rungs l)

mkTransRel : (l : Ladder) → State l → Input l → State l → Set
mkTransRel l b i a = ⟦ mkenv (proj₁ b ++ proj₁ (proj₁ i) ++ proj₁ a) ⊧ mktrans l ⟧pl

mkTransitionSystem : (l : Ladder) → TransitionSystem (Input l)
mkTransitionSystem l = ts (State l) (mkInitialRel l) (mkTransRel l)

♭initial : (init trans safe : PL-Formula) → PL-Formula
♭initial init trans safe = (init && trans) => safe

ψinitial : (init trans safe : PL-Formula) → (before i after : List Bool) → Set
ψinitial init trans safe b i a = ⟦ mkenv (b ++ i ++ a) ⊧ ♭initial init trans safe ⟧pl

♭transition : ∀ (state inp : ℕ) (safe trans inv : PL-Formula) → PL-Formula
♭transition s i safe trans inv = (safe && trans && shiftpl inv s && shiftpl inv (s + i + s)
                                    && shiftpl trans (s + i)) => shiftpl safe (s + i)

ψtransition : (safe trans inv : PL-Formula) → (pre i before i' after : List Bool) → Set
ψtransition safe trans inv pre i before i' after =
  ⟦ mkenv (pre ++ i ++ before) ⊧ safe ⟧pl    × ⟦ mkenv (pre ++ i ++ before) ⊧ trans ⟧pl ×
```

```
  ⟦ mkenv i ⊧ inv ⟧pl × ⟦ mkenv i' ⊧ inv ⟧pl × ⟦ mkenv (before ++ i' ++ after) ⊧ trans ⟧pl
    → ⟦ mkenv (before ++ i' ++ after) ⊧ safe ⟧pl

lemma-transition : {n m : ℕ}
                 → (pre before after : Σ (List Bool) (λ l → length l ≡ n))
                 → (i1 i2 : Σ (List Bool) (λ l → length l ≡ m))
                 → (safety transition inv : PL-Formula)
                 → T (bound (n + m + n) transition)
                 → T (bound (n + m + n) safety)
                 → T (bound m inv)
                 → ⟦ mkenv ((proj₁ pre) ++ (proj₁ i1) ++ (proj₁ before) ++
                           (proj₁ i2) ++ (proj₁ after)) ⊧ ♭transition n m safety transition inv ⟧pl
                 → ψtransition safety transition inv (proj₁ pre)
                              (proj₁ i1) (proj₁ before) (proj₁ i2) (proj₁ after)
lemma-transition (pre , refl) (before , p1) (after , p2) (i1 , refl) (i2 , p4)
                 φs φt φi bt bs bi p (q1 , q2 , q3 , q4 , q5)
  = lem-shift-pl²' φs pre i1 (before ++ i2 ++ after) (p
      ( lem-mkenv-++-pl³ φs pre i1 before (i2 ++ after)
                         (subst (λ k → T (bound (length pre + length i1 + k) φs)) (sym p1) bs) q1
      , lem-mkenv-++-pl³ φt pre i1 before (i2 ++ after)
                         (subst (λ k → T (bound (length pre + length i1 + k) φt)) (sym p1) bt) q2
      , lem-shift-pl φi pre (i1 ++ before ++ i2 ++ after)
                    (lem-mkenv-++-pl φi i1 (before ++ i2 ++ after) bi q3)
      , subst (λ k → ⟦ mkenv (pre ++ i1 ++ before ++ i2 ++ after)
                         ⊧ shiftpl φi (length pre + length i1 + k) ⟧pl) p1
          (lem-shift-pl³ φi pre i1 before (i2 ++ after)
                         (lem-mkenv-++-pl φi i2 after (subst (λ k → T (bound k φi))(sym p4) bi) q4))
      , lem-shift-pl² φt pre i1 (before ++ i2 ++ after) q5))

base-obligation : ∀ l s → PL-Formula
base-obligation l = ♭initial (mkinitial l) (mktrans l)

inductive-obligation : ∀ l s → PL-Formula
inductive-obligation l s = ♭transition (statevars l) (inputvars l) s (mktrans l) (inp-correct l)

⟦base-obligation⟧ : ∀ l s → List Bool → Set
⟦base-obligation⟧ l s x = ⟦ mkenv x ⊧ base-obligation l s ⟧pl

⟦inductive-obligation⟧ : ∀ l s → List Bool → List Bool → List Bool → Set
⟦inductive-obligation⟧ l p s i s' = ⟦ mkenv (s ++ i ++ s') ⊧ inductive-obligation l p ⟧pl

⟦safety⟧ : ∀ s → List Bool → List Bool → List Bool → Set
⟦safety⟧ p b i s = ⟦ mkenv (b ++ i ++ s) ⊧ p ⟧pl

⟦safety⟧π : ∀ s → {A B D : List Bool → Set} → {C : _ → Set} → Σ (List Bool) A
          → Σ (Σ (List Bool) B) C → Σ (List Bool) D → Set
⟦safety⟧π p x y z = ⟦safety⟧ p (proj₁ x) (proj₁ (proj₁ y)) (proj₁ z)

LadderCorrectness : Ladder → PL-Formula → Set
LadderCorrectness l s = Correctness (mkTransitionSystem l) (⟦safety⟧π s)

inductiveproof : ∀ (rl : Ladder) (safety : PL-Formula)
               → (p : T (decproc (base-obligation rl safety)))
               → (q : T (decproc (inductive-obligation rl safety)))
               → (bi : T (bound (statevars rl) (mkinitial rl)))
               → (bt : T (bound (statevars rl + inputvars rl + statevars rl) (mktrans rl)))
               → (bs : T (bound (statevars rl + inputvars rl + statevars rl) safety))
               → (binv : T (bound (inputvars rl) (inp-correct rl)))
               → LadderCorrectness rl safety
inductiveproof rl safety p q bi bt bs binv before (initial .before y) i after trans
 = sound' (base-obligation rl safety) p (mkenv (proj₁ before ++ proj₁ (proj₁ i) ++ proj₁ after))
      ((lem-mkenv-++-pl (mkinitial rl) (proj₁ before) (proj₁ (proj₁ i) ++ proj₁ after)
                        (subst (λ k → T (bound k (mkinitial rl))) (sym (proj₂ before)) bi) y)
      , trans)
inductiveproof rl safety p q bi bt bs binv before (next s r i .before y) i' after trans
 = lemma-transition s before after (proj₁ i) (proj₁ i') safety
                    (mktrans rl) (inp-correct rl) bt bs binv
                    (sound' (inductive-obligation rl safety) q
                            (mkenv (proj₁ s ++ proj₁ (proj₁ i) ++
                                    proj₁ before ++ proj₁ (proj₁ i') ++ proj₁ after)))
                    (inductiveproof rl safety p q bi bt bs binv s r i before y , y
                                    , proj₂ i , proj₂ i' , trans)

abstract
  inductiveProof : (rl : Ladder) → (safety : PL-Formula)
                 → {p : T (decproc (base-obligation rl safety))}
                 → {q : T (decproc (inductive-obligation rl safety))}
                 → {bi : T (bound (statevars rl) (mkinitial rl))}
                 → {bt : T (bound (statevars rl + inputvars rl + statevars rl) (mktrans rl))}
                 → {bs : T (bound (statevars rl + inputvars rl + statevars rl) safety)}
                 → {binv : T (bound (inputvars rl) (inp-correct rl))}
                 → LadderCorrectness rl safety
  inductiveProof rl safety {p} {q} {bi} {bt} {bs}{binv} = inductiveproof rl safety p q bi bt bs binv

ladderapply : ∀ {φ ψ} l → LadderCorrectness l (φ => ψ) → LadderCorrectness l φ
            → LadderCorrectness l ψ
ladderapply l cor→ corφ b r i a tr = cor→ b r i a tr (corφ b r i a tr)
```

```
module TransitionSystem where

record TransitionSystem (Input : Set) : Set₁ where
  constructor
    ts
  field
    State : Set
    Initial : State → Set
    Transition : State → Input → State → Set

data Reachable {I : Set} (l : TransitionSystem I) : TransitionSystem.State l → Set where
  initial : (s : TransitionSystem.State l)
          → TransitionSystem.Initial l s
          → Reachable l s
  next : (s : TransitionSystem.State l)
       → (r : Reachable l s)
       → (i : I)
       → (s' : TransitionSystem.State l)
       → TransitionSystem.Transition l s i s'
       → Reachable l s'

Correctness : {I : Set}
            → (l : TransitionSystem I)
            → (φ : TransitionSystem.State l → I → TransitionSystem.State l → Set)
            → Set
Correctness {I} l φ = (before : TransitionSystem.State l)
                    → Reachable l before
                    → (i : I)
                    → (after : TransitionSystem.State l)
                    → TransitionSystem.Transition l before i after
                    → φ before i after
```

```
module TransitionSystem.Decidable where

open import Data.Nat
open import TransitionSystem
open TransitionSystem.TransitionSystem

record DecidableTransitionSystem {Input : Set}
                                 (ts : TransitionSystem Input) : Set where
  field
    initialState : State ts
    initialCorrectness : Initial ts initialState
    transitionFunction : State ts → Input → State ts
    transitionCorrectness : ∀ s i → Transition ts s i (transitionFunction s i)


  nthState : (inputs : ℕ → Input) → ℕ → State ts
  nthState inputs zero = initialState
  nthState inputs (suc n) = transitionFunction (nthState inputs n) (inputs n)

  nthReachable : ∀ inputs n → Reachable ts (nthState inputs n)
  nthReachable inputs zero = initial initialState initialCorrectness
  nthReachable inputs (suc n) = next (nthState inputs n) (nthReachable inputs n)
                                     (inputs n)
                                     (nthState inputs (suc n))
                                     (transitionCorrectness (nthState inputs n) (inputs n))
open DecidableTransitionSystem

nthStateCorrect : {Input : Set}
                → {ts : TransitionSystem Input}
                → (dts : DecidableTransitionSystem ts)
                → ∀ φ
                → Correctness ts φ
                → ∀ n inputs
                → φ (nthState dts inputs n) (inputs n) (nthState dts inputs (suc n))
nthStateCorrect dts φ correct n inputs = correct (nthState dts inputs n)
                                                 (nthReachable dts inputs n) (inputs n)
                                                 (nthState dts inputs (suc n))
                                                 (transitionCorrectness dts (nthState dts inputs n)
                                                                            (inputs n))
```

```
module Ladder.Decidable where

open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.List
open import Data.Bool
open import Data.Nat hiding (_<_)
open import Data.List.Inhabitence
open import Data.List.Util using (indicies)

open import Relation.Binary.PropositionalEquality hiding ([_])

open import Ladder.Core

open import TransitionSystem.Decidable
open import TransitionSystem

open import PropIso renaming (_==_ to neq)

open import Boolean.Formula

data FinMap {B : Set} (c : ℕ) : List (ℕ × B) → Set where
  [] : FinMap c []
  [_↦_]∷_ : (a : ℕ)
          → (b : B)
          → {l : List (ℕ × B)}
          → {p : T (a notinℕ indicies l)}
          → {q : T (a < c)}
          → FinMap c l
          → FinMap c ((a , b) ∷ l)

data ListProperty {A : Set} (P : List A → A → Set) : List A → Set where
  [] : ListProperty P []
  [_]<_>∷_ : (a : A) → {l : List A} → (pa : P l a)
                    → (pas : ListProperty P l) → ListProperty P (a ∷ l)

PL-FormulaBound : PL-Formula → ℕ → Set
PL-FormulaBound φ n = T (bound n φ)

πPL-FormulaBound :  ℕ → (ℕ × PL-Formula) → Set
πPL-FormulaBound n p = PL-FormulaBound (proj₂ p) n

record LadderWellFormed (l : Ladder) : Set where
  field
    initialmap : FinMap (Ladder.statevars l) (Ladder.initialstate l)
    rungsmap : FinMap (Ladder.statevars l) (Ladder.rungs l)
    rungvarbound : ListProperty (const (πPL-FormulaBound (Ladder.statevars l + Ladder.inputvars l)))
                                (Ladder.rungs l)

lookup : {A : Set} → ∀ {l} {b} → FinMap {A} b l → (default : A) → ℕ → A
lookup [] d n = d
lookup ([ i ↦ j ]∷ y) d n with neq i n
lookup ([ i ↦ j ]∷ y) d n | true = j
lookup ([ i ↦ j ]∷ y) d n | false = lookup y d n

lookup' : {A : Set} → (partialmap : List (ℕ × A)) → (default : A) → (bound : ℕ) → ℕ → A
lookup' [] default b n = default
lookup' (x ∷ partialmap) default b n with b < suc n
lookup' (x ∷ partialmap) default b n | true = default
lookup' (x ∷ partialmap) default b n | false with neq (proj₁ x) n
lookup' (x ∷ partialmap) default b n | false | true = proj₂ x
lookup' (x ∷ partialmap) default b n | false | false = lookup' partialmap default b n

listify : (n : ℕ) → (f : ℕ → Bool) → List Bool
listify zero f = []
listify (suc n) f = (f 0) ∷ (listify n (f ∘ suc))

listify-length : ∀ n f → length (listify n f) ≡ n
listify-length zero f = refl
listify-length (suc n) f = cong suc (listify-length n _)

lem-listify : ∀ n f → Σ[ l ∶ List Bool ] (length l ≡ n)
lem-listify n f = listify n f , listify-length n f

private
  lem : ∀ m j → T (j < m) → (f : ℕ → Bool) → mkenv (listify m f) j ≡ f j
  lem zero    j       p f = ⊥-elim p
  lem (suc m) zero    p f = refl
  lem (suc m) (suc j) p f = lem m j p (f ∘ suc)

  #lem : ∀ n → (l : List (Σ ℕ (λ x → Bool))) → ∀ j b → T (n < suc j) → b ≡ lookup' l b n j
  #lem n []       j b p = refl
  #lem n (x ∷ xs) j b p rewrite Tb p = refl

  #lem' : ∀ {n} → {A : Set} → {l : List (ℕ × A)} → (m : FinMap {A} n l) → ∀ j b → T (n < suc j)
        → b ≡ lookup m b j
  #lem' {n} []                          j b n<j = refl
```

```
  #lem' {n} ([_↦_]∷_ a b {l} {p} {a<n} y) j d n<j
    rewrite ¬Tb (<→≠ a j (<-trans' a n (suc j) a<n n<j)) = #lem' y j d n<j

  ##lem : ∀ n → (f : ℕ → Bool) → ∀ j → T (n < suc j) → mkenv (listify n f) j ≡ false
  ##lem zero    f j       p = refl
  ##lem (suc n) f zero    p = ⊥-elim p
  ##lem (suc n) f (suc j) p = ##lem n (f ∘ suc) j p

  lem' : ∀ m → (l : List (Σ ℕ (λ x → Bool))) → ∀ j
       → mkenv (listify m (lookup' l false m)) j ≡ lookup' l false m j
  lem' m l j with ex-mid (j < m)
  lem' m l j | inj₁ x = lem m j x (lookup' l false m)
  lem' m l j | inj₂ y = trans (##lem m (lookup' l false m) j (<-¬ j m y))
                              (#lem m l j false (<-¬ j m y))

  lem'' : ∀ n → {l : List (ℕ × Bool)} → (↦ : FinMap n l) → ∀ j
        → mkenv (listify n (lookup ↦ false)) j ≡ lookup ↦ false j
  lem'' m x j with ex-mid (j < m)
  lem'' m x j | inj₁ j<m = lem m j j<m (lookup x false)
  lem'' m x j | inj₂ ¬j<m = trans (##lem m (lookup x false) j (<-¬ j m ¬j<m))
                                  (#lem' x j false (<-¬ j m ¬j<m))

  lem-var : ∀ b → ∀ ξ → T b → ⟦ ξ ⊨ injbool b ⟧pl
  lem-var true  ξ = id
  lem-var false ξ = id

  lem-var' : ∀ b → ∀ ξ → ⟦ ξ ⊨ injbool b ⟧pl → T b
  lem-var' true  ξ = id
  lem-var' false ξ = id

{- INITIAL STATE -}
constructInitialState : (l : ∃ LadderWellFormed)
                      → Σ[ m ∶ List Bool ] length m ≡ Ladder.statevars (proj₁ l)
constructInitialState (l , p) = lem-listify (Ladder.statevars l)
                                            (lookup (LadderWellFormed.initialmap p) false)

module InitialProof where

  lem-⊓notin-aux : {i : ℕ}
                 → (l : List (ℕ × Bool))
                 → {z : length l ≡ i}
                 → ∀ n
                 → T (n notinℕ indicies l)
                 → ¬ T ((¥ n) isSubFormula (mkinitial-aux l))
  lem-⊓notin-aux {zero} [] n ⊓notinl q = q
  lem-⊓notin-aux {suc i} [] {()} n ⊓notinl q
  lem-⊓notin-aux {zero} (x ∷ xs) {()} n ⊓notinl q
  lem-⊓notin-aux {suc i} ((a , true) ∷ []) n ⊓notinl q
    rewrite ¬Tb (lem-bool-neg-s (neq n a) (∧-eliml ⊓notinl)) = q
  lem-⊓notin-aux {suc i} ((a , false) ∷ []) n ⊓notinl q
    rewrite ¬Tb (lem-bool-neg-s (neq n a) (∧-eliml ⊓notinl)) = q
  lem-⊓notin-aux {suc i} ((a , true) ∷ x' ∷ xs) {z} n ⊓notinl q
    rewrite ¬Tb (lem-bool-neg-s (neq n a) (∧-eliml ⊓notinl))
    = lem-⊓notin-aux {i} (x' ∷ xs) {cong pred z} n (∧-elimr (not (neq n a)) ⊓notinl) q
  lem-⊓notin-aux {suc i} ((a , false) ∷ x' ∷ xs) {z} n ⊓notinl q
    rewrite ¬Tb (lem-bool-neg-s (neq n a) (∧-eliml ⊓notinl))
    = lem-⊓notin-aux {i} (x' ∷ xs) {cong pred z} n (∧-elimr (not (neq n a)) ⊓notinl) q

  lem-⊓notin : (l : List (ℕ × Bool)) → ∀ n → T (n notinℕ indicies l)
             → ¬ T ((¥ n) isSubFormula (mkinitial-aux l))
  lem-⊓notin l = lem-⊓notin-aux l {refl}

  #lem-destructlookup : {n : ℕ}
                      → {l : List (ℕ × Bool)}
                      → (m : FinMap n l)
                      → (x₁ : ℕ)
                      → (x₂ : Bool)
                      → (x₃ : T (x₁ notinℕ indicies l))
                      → (x₄ : T (x₁ < n))
                      → ∀ i
                      → T (¥ i isSubFormula (mkinitial-aux l))
                      → lookup ([_↦_]∷_ x₁ x₂ {p = x₃} {q = x₄} m) false i ≡ lookup m false i
  #lem-destructlookup m x₁ x₂ x₁⊓notinl x₄ i subform with ex-mid (neq x₁ i)
  #lem-destructlookup {_} {l} m x₁ x₂ x₁⊓notinl x₄ i subform | inj₁ x rewrite lift-== x₁ i x
    = ⊥-elim (lem-⊓notin l i x₁⊓notinl subform)
  #lem-destructlookup m x₁ x₂ x₁⊓notinl x₄ i subform | inj₂ y rewrite ¬Tb y = refl

  lem-destructlookup' : {n : ℕ}
                      → {l : List (ℕ × Bool)}
                      → (m : FinMap n l)
                      → (x₁ : ℕ)
                      → (x₂ : Bool)
                      → (x₃ : T (x₁ notinℕ indicies l))
                      → (x₄ : T (x₁ < n))
                      → ⟦ lookup ([_↦_]∷_ x₁ x₂ {p = x₃} {q = x₄} m) false ⊨ mkinitial-aux l ⟧pl
                      → ⟦ lookup m false ⊨ mkinitial-aux l ⟧pl
  lem-destructlookup' {n} {l} m x₁ x₂ x₃ x₄
    = env-subst-guard (lookup ([_↦_]∷_ x₁ x₂ {p = x₃} {q = x₄} m) false)
```

```
                    (lookup m false) (mkinitial-aux l)
                    (#lem-destructlookup m x₁ x₂ x₃ x₄)

  lem-destructlookup : {n : ℕ}
                     → {l : List (ℕ × Bool)}
                     → (m : FinMap n l)
                     → (x₁ : ℕ)
                     → (x₂ : Bool)
                     → (x₃ : T (x₁ notinℕ indicies l))
                     → (x₄ : T (x₁ < n))
                     → ⟦ lookup m false ⊧ mkinitial-aux l ⟧pl
                     → ⟦ lookup ([_↦_]∷_ x₁ x₂ {p = x₃} {q = x₄} m) false ⊧ mkinitial-aux l ⟧pl
  lem-destructlookup {n} {l} m x₁ x₂ x₃ x₄
    = env-subst-guard (lookup m false)
                      (lookup ([_↦_]∷_ x₁ x₂ {p = x₃} {q = x₄} m) false)
                      (mkinitial-aux l)
                      (λ q w → sym (#lem-destructlookup m x₁ x₂ x₃ x₄ q w))

  #InitialCorrect : {i : ℕ}
                  → {l : List (ℕ × Bool)}
                  → {z : length l ≡ i}
                  → {statevars : ℕ}
                  → (initialstate : FinMap statevars l)
                  → ⟦ lookup initialstate false ⊧ mkinitial-aux l ⟧pl
  #InitialCorrect [] = tt
  #InitialCorrect {_} {._} {m} ([ a ↦ b ]∷ []) rewrite Tb (id-== a) = lem-var b _ , lem-var' b _
  #InitialCorrect {zero} {._} {()} ([_↦_]∷_ a b {._} {i} {j} ([ c ↦ d ]∷ xs))
  #InitialCorrect {suc i'} {._} {z} ([_↦_]∷_ a b {._} {i} {j} ([ c ↦ d ]∷ xs)) rewrite Tb (id-== a)
    = (lem-var b _ , lem-var' b _)
    , lem-destructlookup ([ c ↦ d ]∷ xs) a b i j
                         (#InitialCorrect {i'} {_} {cong pred z} ([ c ↦ d ]∷ xs))

  InitialCorrect : {statevars : ℕ}
                 → {l : List (ℕ × Bool)}
                 → (initialstate : FinMap statevars l)
                 → ⟦ mkenv (listify statevars (lookup initialstate false)) ⊧ mkinitial-aux l ⟧pl
  InitialCorrect {statevars} {l} initialstate = env-subst _ _ (mkinitial-aux l)
                                                           (sym ∘ lem'' statevars initialstate)
                                                           (#InitialCorrect {z = refl} initialstate)

InitialCorrect : (l : ∃ LadderWellFormed)
               → TransitionSystem.Initial (mkTransitionSystem (proj₁ l)) (constructInitialState l)
InitialCorrect l = InitialProof.InitialCorrect  (LadderWellFormed.initialmap (proj₂ l))

{-
TRANSITION FUNCTION
-}
_[_==_] : {A : Set} → List A → ℕ → A → List A
[]       [ n     == k ] = []
(x ∷ xs) [ zero  == k ] = k ∷ xs
(x ∷ xs) [ suc n == k ] = x ∷ xs [ n == k ]

lem-putat-length : {A : Set} → (l : List A) → (a : A) → (n : ℕ) → length (l [ n == a ]) ≡ length l
lem-putat-length []       a n       = refl
lem-putat-length (x ∷ xs) a zero    = refl
lem-putat-length (x ∷ xs) a (suc n) = cong suc (lem-putat-length xs a n)

updateState : ∀ {i} → Σ[ m ∶ List Bool ] length m ≡ i → ℕ → Bool → Σ[ m ∶ List Bool ] length m ≡ i
updateState (st , len) r b = st [ r == b ] , trans (lem-putat-length st b r) len

executeRung : ∀ i j
            → (rung : ℕ × PL-Formula)
            → Σ[ m ∶ List Bool ] length m ≡ i
            → Σ[ m ∶ List Bool ] length m ≡ j
            → Σ[ m ∶ List Bool ] length m ≡ i
executeRung i j (r , φ) state inp
  = updateState state r (eval-pl (mkenv (proj₁ state ++ proj₁ inp)) φ)

executeLadder : ∀ i j
              → (rungs : List (ℕ × PL-Formula))
              → Σ[ m ∶ List Bool ] length m ≡ i
              → Σ[ m ∶ List Bool ] length m ≡ j
              → Σ[ m ∶ List Bool ] length m ≡ i
executeLadder i j []       state inp = state
executeLadder i j (x ∷ xs) state inp = executeLadder i j xs (executeRung i j x state inp) inp

assoc-executeLadder : ∀ i j rungs1 rungs2 rungs3 s t
                    → executeLadder i j (rungs1 ++ rungs2 ++ rungs3) s t
                      ≡ executeLadder i j ((rungs1 ++ rungs2) ++ rungs3) s t
assoc-executeLadder i j []       rungs2 rungs3 s t = refl
assoc-executeLadder i j (x ∷ xs) rungs2 rungs3 s t
  = assoc-executeLadder i j xs rungs2 rungs3 (executeRung i j x s t) t

elim++-executeLadder : ∀ i j rungs1 rungs2 s t
                     → executeLadder i j (rungs1 ++ rungs2) s t
                       ≡ executeLadder i j rungs2 (executeLadder i j rungs1 s t) t
elim++-executeLadder i j []       r2 s t = refl
elim++-executeLadder i j (x ∷ r1) r2 s t = elim++-executeLadder i j r1 r2 (executeRung i j x s t) t
```

```
_prefix_ : {A : Set} → List A → List A → Set
[]       prefix _         = ⊤
(x ∷ xs) prefix []        = ⊥
(x ∷ xs) prefix (y ∷ ys) = x ≡ y × xs prefix ys

id-take-len : {A : Set} → (l : List A) → l ≡ take (length l) l
id-take-len [] = refl
id-take-len (x ∷ xs) = cong (_∷_ x) (id-take-len xs)

lem-putat-eq : (s : List Bool) → ∀ k x φ → ¬ T (neq k x) → mkenv (s [ x == φ ]) k ≡ mkenv (s) k
lem-putat-eq []       k       x       φ k≠x = refl
lem-putat-eq (s ∷ ss) zero    zero    φ k≠x = ⊥-elim (k≠x tt)
lem-putat-eq (s ∷ ss) zero    (suc n) φ k≠x = refl
lem-putat-eq (s ∷ ss) (suc k) zero    φ k≠x = refl
lem-putat-eq (s ∷ ss) (suc k) (suc x) φ k≠x = lem-putat-eq ss k x φ k≠x

id-notin-executeLadder-aux : ∀ i j rn
                           → (r : List _)
                           → ∀ s t k
                           → T (k notin ℕ indicies r)
                           → mkenv (proj₁ (executeLadder i j (take (suc rn) r) s t)) k
                             ≡ mkenv (proj₁ (executeLadder i j (take rn r) s t)) k
id-notin-executeLadder-aux i j zero    []       s t k knotinr = refl
id-notin-executeLadder-aux i j (suc n) []       s t k knotinr = refl
id-notin-executeLadder-aux i j zero    (x ∷ xs) s t k knotinr
  = lem-putat-eq (proj₁ s) k (proj₁ x) (eval-pl (mkenv (proj₁ s ++ proj₁ t)) (proj₂ x))
                 (lem-bool-neg-s _ (∧-eliml knotinr))
id-notin-executeLadder-aux i j (suc rn) (x ∷ xs) s t k knotinr
  = id-notin-executeLadder-aux i j rn xs (executeRung i j x s t) t k
                               (∧-elimr (not (neq k (proj₁ x))) knotinr)

id-notin-executeLadder-aux' : ∀ i j n m
                            → (r : List _)
                            → ∀ s t k
                            → T (k notin ℕ indicies r)
                            → mkenv (proj₁ (executeLadder i j (take m r) s t)) k
                              ≡ mkenv (proj₁ (executeLadder i j (take n r) s t)) k
id-notin-executeLadder-aux' i j zero zero r s t k knotinr = refl
id-notin-executeLadder-aux' i j zero (suc n) r s t k knotinr
  = trans (id-notin-executeLadder-aux i j n r s t k knotinr)
          (id-notin-executeLadder-aux' i j 0 n r s t k knotinr)
id-notin-executeLadder-aux' i j (suc n) zero r s t k knotinr
  = trans (id-notin-executeLadder-aux' i j n 0 r s t k knotinr)
          (sym (id-notin-executeLadder-aux i j n r s t k knotinr))
id-notin-executeLadder-aux' i j (suc n) (suc n') r s t k knotinr
  = trans (id-notin-executeLadder-aux i j n' r s t k knotinr)
          (trans (id-notin-executeLadder-aux' i j n n' r s t k knotinr)
                 (sym (id-notin-executeLadder-aux i j n r s t k knotinr)))

id-notin-executeLadder : ∀ i j r s t k
                       → T (k notin ℕ indicies r)
                       → mkenv (proj₁ (executeLadder i j r s t)) k ≡ mkenv (proj₁ s) k
id-notin-executeLadder i j r s t k knotinr
  = trans (cong (λ z → mkenv (proj₁ (executeLadder i j z s t)) k) (id-take-len r))
          (id-notin-executeLadder-aux' i j 0 (length r) r s t k knotinr)

id-notin-executeLadder' : ∀ i j r s t k → T (k notin ℕ indicies r)
                        → mkenv (proj₁ (executeLadder i j r s t)) ++ proj₁ t) k
                          ≡ mkenv (proj₁ s ++ proj₁ t) k
id-notin-executeLadder' i j r s t k knotinr
  = extendenv (proj₁ (executeLadder i j r s t)) (proj₁ s) (proj₁ t)
              (\ z → T (z notin ℕ indicies r)) (trans (proj₂ (executeLadder i j r s t))
                                                      (sym (proj₂ s)))
              (id-notin-executeLadder i j r s t) k knotinr

dist-⊓notin : ∀ {B} → (a b : List (ℕ × B)) → ∀ c → T (c notin ℕ indicies (a ++ b))
            → T (c notin ℕ indicies a) × T (c notin ℕ indicies b)
dist-⊓notin [] b c p = tt , p
dist-⊓notin (x ∷ xs) b c p = Prod.map (∧-intro _ _ (∧-eliml {a = not (neq c (proj₁ x))} p)) id
                                      (dist-⊓notin xs b c (∧-elimr (not (neq c (proj₁ x))) p))

lem-elim++-eq-aux : ∀ {i} k
                  → (rungs1 rungs2 : List (ℕ × PL-Formula))
                  → T (k isin ℕ indicies rungs1)
                  → FinMap i (rungs1 ++ rungs2)
                  → T (k notin ℕ indicies rungs2)
lem-elim++-eq-aux k [] r2 kinr1 m = ⊥-elim kinr1
lem-elim++-eq-aux k ((a , b) ∷ xs) r2 kinr1 ([_↦_]∷_ ._ ._ {._} {notin ℕ} y) with ex-mid (neq k a)
...| inj₁ x rewrite Tb x | lift-== k a x = proj₂ (dist-⊓notin xs r2 a notin ℕ)
...| inj₂ y' rewrite ¬Tb y' = lem-elim++-eq-aux k xs r2 kinr1 y

lem-elim++-eq : ∀ i j s t rungs1 rungs2 k
              → T (k isin ℕ indicies rungs1)
              → FinMap i (rungs1 ++ rungs2)
              → mkenv (proj₁ (executeLadder i j rungs2 (executeLadder i j rungs1 s t) t)) k
                ≡ mkenv (proj₁ (executeLadder i j rungs1 s t)) k
```

```
lem-elim++-eq i j s t rungs1 rungs2 k kinr1 m
  = id-notin-executeLadder i j rungs2 (executeLadder i j rungs1 s t) t k
                           (lem-elim++-eq-aux k rungs1 rungs2 kinr1 m)

unfold-mymap : ∀ n v vs k → T (neq v k) → mymap n (v :: vs) k ≡ n + k
unfold-mymap n v vs k eq rewrite Tb eq | lift-== v k eq = refl

unfold-mymap' : ∀ n v vs k → ¬ T (neq v k) → mymap n (v :: vs) k ≡ mymap n vs k
unfold-mymap' n v vs k eq rewrite ¬Tb eq = refl

elim-mymap-c1 : ∀ s t v k → T (k isinℕ v) → mkenv (s ++ t) (mymap (length s) v k) ≡ mkenv t k
elim-mymap-c1 s        t []       k kinv = ⊥-elim kinv
elim-mymap-c1 s        t (v :: vs) k kinv with ex-mid (neq v k)
elim-mymap-c1 []       t (v :: vs) k kinv | inj₁ x rewrite Tb x | lift-== v k x = refl
elim-mymap-c1 (s :: ss) t (v :: vs) k kinv | inj₁ x rewrite Tb x
  = trans (cong (mkenv (ss ++ t)) (trans (sym (cong (λ j → length ss + j) (sym (lift-== v k x))))
                                         (sym (unfold-mymap (length ss) v vs k x))))
          (elim-mymap-c1 ss t (v :: vs) k kinv)
elim-mymap-c1 s        t (v :: vs) k kinv | inj₂ y rewrite ¬Tb y
  = elim-mymap-c1 s t vs k (∨-elim (⊥-elim ∘ y ∘ sym-== k v) id kinv)

elim-mymap-c2 : ∀ s t v k → T (k notinℕ v) → T (k < length s)
              → mkenv (s ++ t) (mymap (length s) v k) ≡ mkenv s k
elim-mymap-c2 s t [] k knotinv k<s = sym (lem-mkenv-++-eq s _ t k refl k<s)
elim-mymap-c2 s t (v :: vs) k knotinv k<s
  rewrite ¬Tb (lem-bool-neg-s _ (∧-elimₗ knotinv) ∘ sym-== v k)
  = elim-mymap-c2 s t vs k (∧-elimᵣ (not (neq k v)) knotinv) k<s

elim-mymap : ∀ s t v k → T (k < length s) → T (k isinℕ v) ⊎ ¬ T (k isinℕ v)
           → mkenv (s ++ t)(mymap (length s) v k) ≡ mkenv t k
             ⊎ mkenv (s ++ t)(mymap (length s) v k) ≡ mkenv s k
elim-mymap s t v k k<s = Sum.map (elim-mymap-c1 s t v k)
                                 (\ x → elim-mymap-c2 s t v k (lem-notisin k v x) k<s)

elim-mymap-in : ∀ n v k → T (k isinℕ v) → mymap n v k ≡ n + k
elim-mymap-in n [] k kinv = ⊥-elim kinv
elim-mymap-in n (x :: xs) k kinv with ex-mid (neq x k)
elim-mymap-in n (x :: xs) k kinv | inj₁ a rewrite Tb a | lift-== x k a = refl
elim-mymap-in n (x :: xs) k kinv | inj₂ a rewrite ¬Tb a | ¬Tb (a ∘ sym-== k x)
  = elim-mymap-in n xs k kinv

elim-mymap-notin : ∀ n v k → T (k notinℕ v) → mymap n v k ≡ k
elim-mymap-notin n []       k knotinv = refl
elim-mymap-notin n (x :: xs) k knotinv with ex-mid (neq x k)
elim-mymap-notin n (x :: xs) k knotinv | inj₁ x' rewrite Tb (sym-== x k x') = ⊥-elim knotinv
elim-mymap-notin n (x :: xs) k knotinv | inj₂ y rewrite ¬Tb y | ¬Tb (y ∘ sym-== k x)
  = elim-mymap-notin n xs k knotinv

open import Algebra
#elim-mymap-in : ∀ i j s t rungs1 rungs2 k
               → T (k isinℕ indicies rungs1)
               → mkenv (proj₁ s ++ proj₁ t ++ proj₁ (executeLadder i j (rungs1 ++ rungs2) s t))
                       (mymap (length (proj₁ s ++ proj₁ t)) (indicies rungs1) k)
                 ≡ mkenv (proj₁ (executeLadder i j (rungs1 ++ rungs2) s t)) k
#elim-mymap-in i j s t r1 r2 k isin
  = trans (cong (λ z → mkenv z (mymap (length (proj₁ s ++ proj₁ t)) (indicies r1) k))
                (sym (Monoid.assoc (monoid Bool) (proj₁ s) (proj₁ t)
                                   (proj₁ (executeLadder i j (r1 ++ r2) s t)))))
          (elim-mymap-c1 (proj₁ s ++ proj₁ t)
                         (proj₁ (executeLadder i j (r1 ++ r2) s t)) (indicies r1) k isin)

#elim-mymap-notin : ∀ i j s t rungs1 rungs2 k
                  → T (k < (length (proj₁ s) + length (proj₁ t)))
                  → T (k notinℕ indicies rungs1)
                  → mkenv (proj₁ s ++ proj₁ t ++ proj₁ (executeLadder i j (rungs1 ++ rungs2) s t))
                          (mymap (length (proj₁ s ++ proj₁ t)) (indicies rungs1) k)
                    ≡ mkenv (proj₁ s ++ proj₁ t) k
#elim-mymap-notin i j s t r1 r2 k k<s+t notin'
  = trans (cong (λ z → mkenv z (mymap (length (proj₁ s ++ proj₁ t)) (indicies r1) k))
                (sym (Monoid.assoc (monoid Bool) (proj₁ s) (proj₁ t)
                                   (proj₁ (executeLadder i j (r1 ++ r2) s t)))))
          (elim-mymap-c2 (proj₁ s ++ proj₁ t)
                         (proj₁ (executeLadder i j (r1 ++ r2) s t))
                         (indicies r1) k notin'
                         (subst (λ j' → T (k < j')) (lem-length (proj₁ s) (proj₁ t)) k<s+t))

#elim-mymap-in' : ∀ i j s t rungs1 rungs2 k
                → T (k isinℕ indicies rungs1)
                → FinMap i (rungs1 ++ rungs2)
                → mkenv (proj₁ s ++ proj₁ t ++ proj₁ (executeLadder i j (rungs1 ++ rungs2) s t))
                        (mymap (length (proj₁ s ++ proj₁ t)) (indicies rungs1) k)
                  ≡ mkenv (proj₁ (executeLadder i j rungs1 s t)) k
#elim-mymap-in' i j s t r1 r2 k isin' m = trans (#elim-mymap-in i j s t r1 r2 k isin')
                                               (trans (cong (λ z → mkenv (proj₁ z) k)
                                                            (elim++-executeLadder i j r1 r2 s t))
                                                      (lem-elim++-eq i j s t r1 r2 k isin' m))

#elim-mymap-notin' : ∀ i j s t rungs1 rungs2 k
```

```
                  → T (k < (length (proj₁ s) + length (proj₁ t)))
                  → T (k notin𝕟 indicies rungs1)
                  → mkenv (proj₁ s ++ proj₁ t ++ proj₁ (executeLadder i j (rungs1 ++ rungs2) s t))
                          (mymap (length (proj₁ s ++ proj₁ t)) (indicies rungs1) k)
                    ≡ mkenv (proj₁ (executeLadder i j rungs1 s t) ++ proj₁ t) k
#elim-mymap-notin' i j s t r1 r2 k k<s+t notin'
  = trans (#elim-mymap-notin i j s t r1 r2 k k<s+t notin')
          (sym (id-notin-executeLadder' i j r1 s t k notin'))

#elim-mymap : ∀ i j s t rungs1 rungs2 k → T (k < (length (proj₁ s) + length (proj₁ t)))
            → T (k isin𝕟 indicies rungs1) ⊎ ¬ T (k isin𝕟 indicies rungs1)
            → mkenv (proj₁ s ++ proj₁ t ++ proj₁ (executeLadder i j (rungs1 ++ rungs2) s t))
                    (mymap (length (proj₁ s ++ proj₁ t)) (indicies rungs1) k)
              ≡ mkenv (proj₁ (executeLadder i j (rungs1 ++ rungs2) s t)) k
             ⊎ mkenv (proj₁ s ++ proj₁ t ++ proj₁ (executeLadder i j (rungs1 ++ rungs2) s t))
                     (mymap (length (proj₁ s ++ proj₁ t)) (indicies rungs1) k)
               ≡ mkenv (proj₁ s ++ proj₁ t) k
#elim-mymap i j s t r1 r2 k k<s+t isin
  = Sum.map (#elim-mymap-in i j s t r1 r2 k)
            (#elim-mymap-notin i j s t r1 r2 k k<s+t ∘ lem-notisin k (indicies r1)) isin

weaken-isin-++ : ∀ k → (r1 r2 : List (𝕟 × PL-Formula))
               → T (k isin𝕟 indicies r1)
               → T (k isin𝕟 indicies (r1 ++ r2))
weaken-isin-++ k []       r2 = ⊥-elim
weaken-isin-++ k (x ∷ xs) r2 = f∨g {a = neq k (proj₁ x)} id (weaken-isin-++ k xs r2)

#elim-mymap'-aux : ∀ i (r : List (𝕟 × PL-Formula)) → ∀ k → T (k isin𝕟 indicies r) → FinMap i r
                 → T (k < i)
#elim-mymap'-aux i [] k kinr m = ⊥-elim kinr
#elim-mymap'-aux i (.(a , b) ∷ xs) k kinr ([_↦_]∷_ a b {._} {_} {z} y)
  = ∨-elim (λ k=a → subst (λ j → T (j < i)) (sym (lift-== k a k=a)) z)
           (λ z' → #elim-mymap'-aux i xs k z' y) kinr

#elim-mymap' : ∀ i j s t rungs1 rungs2 k → T (k < (length (proj₁ s) + length (proj₁ t)))
             → FinMap i (rungs1 ++ rungs2)
             → mkenv (proj₁ s ++ proj₁ t ++ proj₁ (executeLadder i j (rungs1 ++ rungs2) s t))
                     (mymap (length (proj₁ s ++ proj₁ t)) (indicies rungs1) k)
               ≡ mkenv (proj₁ (executeLadder i j rungs1 s t) ++ proj₁ t) k
#elim-mymap' i j s t r1 r2 k k<s+t m with ex-mid (k isin𝕟 indicies r1)
...| inj₁ x = trans (#elim-mymap-in' i j s t r1 r2 k x m)
                    (lem-mkenv-++-eq (proj₁ (executeLadder i j r1 s t)) _ (proj₁ t) k refl
                                     (#elim-mymap'-aux _ (r1 ++ r2) k (weaken-isin-++ k r1 r2 x)
                                                       (subst (λ z → FinMap z (r1 ++ r2))
                                                         (sym (proj₂ (executeLadder i j r1 s t))) m)))
...| inj₂ x = #elim-mymap-notin' i j s t r1 r2 k k<s+t (lem-notisin k (indicies r1) x)

lem-putat-eq' : ∀ s x φ → T (x < length s) → mkenv (s [ x == φ ]) x ≡ φ
lem-putat-eq' []       x       φ x<s = ⊥-elim x<s
lem-putat-eq' (x ∷ xs) zero    φ x<s = refl
lem-putat-eq' (x ∷ xs) (suc n) φ x<s = lem-putat-eq' xs n φ x<s

lemma5 : ∀ i j s t rungs a φ
       → T (a < length (proj₁ s))
       → mkenv (proj₁ (executeLadder i j (rungs ++ (a , φ) ∷ []) s t)) a
         ≡ eval-pl (mkenv (proj₁ (executeLadder i j rungs s t) ++ proj₁ t)) φ
lemma5 i j s t [] a φ a<s = lem-putat-eq' (proj₁ s) a (eval-pl (mkenv (proj₁ s ++ proj₁ t)) φ) a<s
lemma5 i j s t ((a' , φ') ∷ rungs) a φ a<s
  = lemma5 i j (executeRung i j (a' , φ') s t) t rungs a φ
           (subst (\ k → T (a < k))
                  (sym (lem-putat-length (proj₁ s) (eval-pl (mkenv (proj₁ s ++ proj₁ t)) φ')
                                         a')) a<s)

lem-apply-¥-∘ : ∀ ξ φ f → ⟦ ξ ⊨ apply-¥-pl φ f ⟧pl ≡ ⟦ ξ ∘ f ⊨ φ ⟧pl
lem-apply-¥-∘ ξ ¥true     f = refl
lem-apply-¥-∘ ξ ¥false    f = refl
lem-apply-¥-∘ ξ (y || y') f = cong₂ _⊎_ (lem-apply-¥-∘ ξ y f) (lem-apply-¥-∘ ξ y' f)
lem-apply-¥-∘ ξ (y && y') f = cong₂ _×_ (lem-apply-¥-∘ ξ y f) (lem-apply-¥-∘ ξ y' f)
lem-apply-¥-∘ ξ (y => y') f = cong₂ (λ a b → a → b) (lem-apply-¥-∘ ξ y f) (lem-apply-¥-∘ ξ y' f)
lem-apply-¥-∘ ξ (¥ y)     f = refl

lem-apply-¥-∘-eval : ∀ ξ φ f → eval-pl ξ (apply-¥-pl φ f) ≡ eval-pl (ξ ∘ f) φ
lem-apply-¥-∘-eval ξ ¥true     f = refl
lem-apply-¥-∘-eval ξ ¥false    f = refl
lem-apply-¥-∘-eval ξ (y || y') f = cong₂ _∨_ (lem-apply-¥-∘-eval ξ y f) (lem-apply-¥-∘-eval ξ y' f)
lem-apply-¥-∘-eval ξ (y && y') f = cong₂ _∧_ (lem-apply-¥-∘-eval ξ y f) (lem-apply-¥-∘-eval ξ y' f)
lem-apply-¥-∘-eval ξ (y => y') f
  = cong₂ (λ a b → not a ∨ b) (lem-apply-¥-∘-eval ξ y f) (lem-apply-¥-∘-eval ξ y' f)
lem-apply-¥-∘-eval ξ (¥ y)     f = refl

mktrans-aux' : 𝕟 → List (𝕟 × PL-Formula) → List (𝕟 × PL-Formula) → PL-Formula
mktrans-aux' n v []             = ¥true
mktrans-aux' n v ((a , b) ∷ as) = (¥ (n + a) <=> apply-¥-pl b (mymap n (indicies v))) &&
                                  mktrans-aux' n (v ++ [ (a , b) ]) as

sublist : List (𝕟 × PL-Formula) → List (𝕟 × PL-Formula) → Set
sublist x y = ∀ γ → T (γ isin𝕟 indicies x) → T (γ isin𝕟 indicies y)
```

```
id-sublist : ∀ l → sublist l l
id-sublist l = λ _ → id

elim++-isin : {A : Set} → ∀ k l x → T (x isinℕ indicies {A} (k ++ l))
            → T (x isinℕ indicies k) ⊎ T (x isinℕ indicies l)
elim++-isin []       l x  p = inj₂ p
elim++-isin (x :: xs) l x' p with ex-mid (neq x' (proj₁ x))
elim++-isin (x :: xs) l x' p | inj₁ x0 = inj₁ (V-intro _ _ x0)
elim++-isin (x :: xs) l x' p | inj₂ y rewrite ¬Tb y = elim++-isin xs l x' p

elim++-isin' : {A : Set} → ∀ k l x → T (x isinℕ indicies k) ⊎ T (x isinℕ indicies l)
            → T (x isinℕ indicies {A} (k ++ l))
elim++-isin' []       l x  p = [ ⊥-elim , id ]' p
elim++-isin' (x :: xs) l x' p with ex-mid (neq x' (proj₁ x))
elim++-isin' (x :: xs) l x' p | inj₁ x0 = V-intro _ _ x0
elim++-isin' (x :: xs) l x' p | inj₂ y rewrite ¬Tb y = elim++-isin' xs l x' p

sublist-extend : (k l : List (ℕ × PL-Formula)) → sublist k l → (x : List (ℕ × PL-Formula))
               → sublist (k ++ x) (l ++ x)
sublist-extend k l p x γ yink+x = elim++-isin' l x γ (Sum.map ( p γ ) id (elim++-isin k x γ yink+x))

_≈_ : List (ℕ × PL-Formula) → List (ℕ × PL-Formula) → Set
a ≈ b = sublist a b × sublist b a

<_≈_,_>++_ : (k l : List (ℕ × PL-Formula)) → k ≈ l → (x : List (ℕ × PL-Formula))
           → (k ++ x) ≈ (l ++ x)
< k ≈ l , p >++ x = (sublist-extend k l (proj₁ p) x) , (sublist-extend l k (proj₂ p) x)

lem-mymap-comm : ∀ (ξ : Env) n (rungs rungs' : List (ℕ × PL-Formula)) v → rungs ≈ rungs'
               → T (ξ (mymap n (indicies rungs) v)) → T (ξ (mymap n (indicies rungs') v))
lem-mymap-comm ξ n r r' v eqi with ex-mid (v isinℕ (indicies r))
lem-mymap-comm ξ n r r' v eqi | inj₁ x rewrite elim-mymap-in n (indicies r) v x
                                             | elim-mymap-in n (indicies r') v (proj₁ eqi v x) = id
lem-mymap-comm ξ n r r' v eqi | inj₂ y
  rewrite elim-mymap-notin n (indicies r) v (lem-notisin v (indicies r) y)
        | elim-mymap-notin n (indicies r') v (lem-notisin v (indicies r') (y ∘ (proj₂ eqi v))) = id

lem-ap-mymap-comm : ∀ ξ n rungs rungs' φ → rungs ≈ rungs'
                  → ⟦ ξ ⊧ apply-¥-pl φ (mymap n (indicies rungs)) ⟧pl
                  → ⟦ ξ ⊧ apply-¥-pl φ (mymap n (indicies rungs')) ⟧pl
lem-ap-mymap-comm ξ n r r' ¥true     eqi = id
lem-ap-mymap-comm ξ n r r' ¥false    eqi = id
lem-ap-mymap-comm ξ n r r' (y || y') eqi = Sum.map (lem-ap-mymap-comm ξ n r r' y eqi)
                                                  (lem-ap-mymap-comm ξ n r r' y' eqi)
lem-ap-mymap-comm ξ n r r' (y && y') eqi = Prod.map (lem-ap-mymap-comm ξ n r r' y eqi)
                                                   (lem-ap-mymap-comm ξ n r r' y' eqi)
lem-ap-mymap-comm ξ n r r' (y => y') eqi
  = λ x → lem-ap-mymap-comm ξ n r r' y' eqi ∘ x ∘ lem-ap-mymap-comm ξ n r' r y (swap eqi)
lem-ap-mymap-comm ξ n r r' (¥ y)     eqi = lem-mymap-comm ξ n r r' y eqi

lem-mktrans-comm : ∀ ξ n rungs1 rungs1' rungs2 → rungs1 ≈ rungs1'
                 → ⟦ ξ ⊧ mktrans-aux' n rungs1 rungs2 ⟧pl
                 → ⟦ ξ ⊧ mktrans-aux' n rungs1' rungs2 ⟧pl
lem-mktrans-comm ξ n r1 r1' []       eqi = id
lem-mktrans-comm ξ n r1 r1' (x :: xs) eqi
  = Prod.map (Prod.map (λ x' x0 → lem-ap-mymap-comm ξ n r1 r1' (proj₂ x) eqi (x' x0))
                       (λ x0 x1 → x0 (lem-ap-mymap-comm ξ n r1' r1 (proj₂ x) (swap eqi) x1)))
             (lem-mktrans-comm ξ n (r1 ++ [ x ]) (r1' ++ [ x ]) xs (< r1 ≈ r1' , eqi >++ [ x ]))

append-conc : ∀ r x → (r ++ [ x ]) ≈ (x :: r)
append-conc [] x = (\ _ → id) , (\ _ → id)
append-conc (x :: xs) x'
  = Prod.map (λ x0 γ → V-elim (V-intror (neq γ (proj₁ x')) _ ∘ V-introl (neq γ (proj₁ x)) _)
                              (λ x1 → fVg {a = neq γ (proj₁ x')}
                                          id (V-intror (neq γ (proj₁ x)) _) (x0 γ x1)))
             (λ x1 γ → V-elim {a = neq γ (proj₁ x')}
                              (V-intror (neq γ (proj₁ x)) _ ∘ x1 γ ∘ V-introl _ _)
                              (fVg {a = neq γ (proj₁ x)} id (x1 γ ∘ V-intror (neq γ (proj₁ x')) _)))
             (append-conc xs x')

lem-mktrans-aux' : ∀ ξ → (n : ℕ) → (r1 r2 : List (ℕ × PL-Formula)) → ⟦ ξ ⊧ mktrans-aux' n r1 r2 ⟧pl
                 → ⟦ ξ ⊧ mktrans-aux n (indicies r1) r2 ⟧pl
lem-mktrans-aux' ξ n r1 [] = id
lem-mktrans-aux' ξ n r1 (x :: xs)
  = Prod.map id (λ z → lem-mktrans-aux' ξ n (x :: r1) xs
                              (lem-mktrans-comm ξ n (r1 ++ [ x ]) (x :: r1) xs (append-conc r1 x) z))

lem-listbound : ∀ n l → ListProperty (λ _ k → T (k < n)) l → ∀ j → ¬ T (j < n) → T (j notinℕ l)
lem-listbound n []       lp                j ¬j<n = tt
lem-listbound n (x :: xs) lp                j ¬j<n with ex-mid (neq j x)
lem-listbound n (x :: xs) ([ ._ ]< q >:: lp) j ¬j<n | inj₁ x' rewrite lift-== j x x' = ⊥-elim (¬j<n q)
lem-listbound n (x :: xs) ([ ._ ]< q >:: lp) j ¬j<n | inj₂ y rewrite ¬Tb y
  = lem-listbound n xs lp j ¬j<n

lem-notin-mymap : ∀ n j l → T (j notinℕ l) → j ≡ mymap n l j
lem-notin-mymap n j []       jnotinl = refl
lem-notin-mymap n j (x :: xs) jnotinl rewrite ¬Tb ((lem-bool-neg-s _ (∧-eliml jnotinl)) ∘ sym-== x j)
```

```
  = lem-notin-mymap n j xs (∧-elimr (not (neq j x)) jnotinl)

module TransitionProof where

  listprop-proj : (r1 r2 : List (ℕ × PL-Formula)) → ∀ a φ n
                → ListProperty (λ _ x → PL-FormulaBound (proj₂ x) n) (r1 ++ (a , φ) :: r2)
                → PL-FormulaBound φ n
  listprop-proj [] r2 a φ n ([ .(a , φ) ]< pa >:: pas) = pa
  listprop-proj (x :: xs) r2 a φ n ([ .x ]< pa >:: pas) = listprop-proj xs r2 a φ n pas

  TransitionCorrect'-stepa : (r1 r2 : List (ℕ × PL-Formula))
                           → {sv : ℕ}
                           → (a : ℕ)
                           → (φ : PL-Formula)
                           → (rungs : FinMap sv (r1 ++ (a , φ) :: r2))
                           → (iv : ℕ)
                           → (rungbound : ListProperty (λ _ x → PL-FormulaBound (proj₂ x) (sv + iv))
                                                       (r1 ++ (a , φ) :: r2))
                           → (s : Σ[ m ∶ List Bool ] length m ≡ sv)
                           → (i : Σ[ m ∶ List Bool ] length m ≡ iv)
                           → ∀ k
                           → T (¥ k isSubFormula φ)
                           → (mkenv (proj₁ s ++ proj₁ i ++ proj₁ (executeLadder sv iv (r1 ++
                                     (a , φ) :: r2) s i)) ∘ (mymap (sv + iv) (indicies r1))) k
                             ≡ mkenv (proj₁ (executeLadder sv iv r1 s i) ++ proj₁ i) k
  TransitionCorrect'-stepa r1 r2 {sv} a φ rungs iv rb s i k ksub
    = trans (cong (λ z → mkenv (proj₁ s ++ proj₁ i ++ proj₁ (executeLadder sv iv (r1 ++
                                (a , φ) :: r2) s i)) (mymap z (indicies r1) k))
                  (trans (cong₂ _+_ (sym (proj₂ s)) (sym (proj₂ i)))
                         (lem-length (proj₁ s) (proj₁ i))))
            (#elim-mymap' sv iv s i r1 ((a , φ) :: r2) k
                          (subst (λ z → T (k < z)) (cong₂ _+_ (sym (proj₂ s)) (sym (proj₂ i)))
                              (lem-bound φ (sv + iv) k (listprop-proj r1 r2 a φ (sv + iv) rb) ksub))
                          rungs)

  lem-mkenv-+ : ∀ k l a → mkenv (k ++ l) (length k + a) ≡ mkenv l a
  lem-mkenv-+ []       l a = refl
  lem-mkenv-+ (x :: xs) l a = lem-mkenv-+ xs l a

  lem-mkenv-+² : ∀ k l m a → mkenv (k ++ l ++ m) (length k + length l + a) ≡ mkenv m a
  lem-mkenv-+² []       = lem-mkenv-+
  lem-mkenv-+² (x :: xs) = lem-mkenv-+²

  lem-finmap-proj< : ∀ n → (r1 r2 : List (ℕ × PL-Formula)) → ∀ a φ
                   → (rungs : FinMap n (r1 ++ ((a , φ) :: r2)))
                   → T (a < n)
  lem-finmap-proj< n [] r2 a φ ([_↦_]:: _ _ {._} {_} {z} y) = z
  lem-finmap-proj< n (x :: xs) r2 a φ ([_↦_]:: _ _ {._} {_} {z} y) = lem-finmap-proj< n xs r2 a φ y

  lem-isin : {A : Set} → ∀ r1 a φ → T (a isinℕ indicies {A} (r1 ++ [ (a , φ) ]))
  lem-isin [] a φ = ∨-introl (neq a a) _ (id-== a)
  lem-isin (x :: xs) a φ = ∨-intror (neq a (proj₁ x)) _ (lem-isin xs a φ)

  TransitionCorrect'-stepb : (r1 r2 : List (ℕ × PL-Formula))
                           → {sv : ℕ}
                           → (a : ℕ)
                           → (φ : PL-Formula)
                           → (rungs : FinMap sv (r1 ++ (a , φ) :: r2))
                           → (iv : ℕ)
                           → (rungbound : ListProperty (λ _ x → PL-FormulaBound (proj₂ x) (sv + iv))
                                                       (r1 ++ (a , φ) :: r2))
                           → (s : Σ[ m ∶ List Bool ] length m ≡ sv)
                           → (i : Σ[ m ∶ List Bool ] length m ≡ iv)
                           → mkenv (proj₁ s ++ proj₁ i ++ proj₁ (executeLadder sv iv (r1 ++
                                     (a , φ) :: r2) s i)) (sv + iv + a)
                             ≡ eval-pl (mkenv (proj₁ (executeLadder sv iv r1 s i) ++ proj₁ i)) φ
  TransitionCorrect'-stepb r1 r2 {sv} a φ rungs iv rb s i
    = trans (trans (cong₂  mkenv {y = proj₁ s ++ _} refl
                           (cong₂ (λ x y → x + y + a) (sym (proj₂ s)) (sym (proj₂ i))))
                   ((lem-mkenv-+² (proj₁ s) (proj₁ i) (proj₁ (executeLadder sv iv (r1 ++
                                     (a , φ) :: r2) s i)) a)))
            (trans (cong₂ mkenv (cong proj₁ (assoc-executeLadder sv iv r1 ((a , φ) :: [])
                                                                 r2 s i)) refl)
                   (trans (cong₂ mkenv (cong proj₁ (elim++-executeLadder sv iv (r1 ++
                                                                 [ (a , φ) ]) r2 s i)) refl)
                          (trans (lem-elim++-eq sv iv s i (r1 ++ [ (a , φ) ]) r2 a (lem-isin r1 a φ)
                                    (subst (FinMap sv) (sym (Monoid.assoc (monoid (ℕ × PL-Formula))
                                                r1 [ (a , φ) ] r2)) rungs))
                                 (lemma5 sv iv s i r1 a φ (subst (T ∘ _<_ a) (sym (proj₂ s))
                                           (lem-finmap-proj< sv r1 r2 a φ rungs)))))))

  TransitionCorrect'-step : (r1 r2 : List (ℕ × PL-Formula))
                          → {sv : ℕ}
                          → (a : ℕ)
                          → (φ : PL-Formula)
                          → (rungs : FinMap sv (r1 ++ (a , φ) :: r2))
                          → (iv : ℕ)
                          → (rungbound : ListProperty (λ _ x → PL-FormulaBound (proj₂ x) (sv + iv))
```

```
                                                  (r1 ++ (a , φ) ∷ r2))
                          → (s : Σ[ m ∶ List Bool ] length m ≡ sv)
                          → (i : Σ[ m ∶ List Bool ] length m ≡ iv)
                          → ⟦ mkenv (proj₁ s ++ proj₁ i ++ proj₁ (executeLadder sv iv (r1 ++
                                       (a , φ) ∷ r2) s i))
                              ⊧ ¥ (sv + iv + a) <=> apply-¥-pl φ (mymap (sv + iv) (indicies r1)) ⟧pl
  TransitionCorrect'-step r1 r2 {sv} a φ rungs iv rungbound s i
    rewrite trans (lem-apply-¥-∘ (mkenv (proj₁ s ++ proj₁ i ++ proj₁ (executeLadder sv iv (r1 ++
                                          (a , φ) ∷ r2) s i))) φ (mymap (sv + iv) (indicies r1)))
                  (env-eq-guard _ _ φ (TransitionCorrect'-stepa r1 r2 a φ rungs iv rungbound s i))
          | TransitionCorrect'-stepb r1 r2 a φ rungs iv rungbound s i
    = lem-eval' (mkenv (proj₁ (executeLadder sv iv r1 s i) ++ proj₁ i)) φ ,
      lem-eval (mkenv (proj₁ (executeLadder sv iv r1 s i) ++ proj₁ i)) φ

  #TransitionCorrect' : (r1 r2 : List (ℕ × PL-Formula))
                      → {sv : ℕ}
                      → (rungs : FinMap sv (r1 ++ r2))
                      → (iv : ℕ)
                      → (rungbound : ListProperty (λ _ x → PL-FormulaBound (proj₂ x) (sv + iv))
                                                  (r1 ++ r2))
                      → (s : Σ[ m ∶ List Bool ] length m ≡ sv)
                      → (i : Σ[ m ∶ List Bool ] length m ≡ iv)
                      → ⟦ mkenv (proj₁ s ++ proj₁ i ++ proj₁ (executeLadder sv iv (r1 ++ r2) s i))
                                ⊧ mktrans-aux' (sv + iv) r1 r2 ⟧pl
  #TransitionCorrect' r1 [] rungs iv rungbound s i = tt
  #TransitionCorrect' r1 ((a , φ) ∷ r2) {sv} rungs iv rungbound s i
    = TransitionCorrect'-step r1 r2 a φ rungs iv rungbound s i ,
      env-subst-guard _ _ (mktrans-aux' (sv + iv) (r1 ++ (a , φ) ∷ []) r2)
          (λ n p → cong₂ mkenv (cong₂ _++_ {x = proj₁ s} refl (cong₂ _++_ {x = proj₁ i} refl
                     (cong proj₁ (sym (assoc-executeLadder sv iv r1 [ (a , φ) ] r2 s i))))) refl)
          (#TransitionCorrect' (r1 ++ [ (a , φ) ]) r2 (subst (FinMap sv) (sym (Monoid.assoc (monoid
            (ℕ × PL-Formula)) r1 [ (a , φ) ] r2)) rungs) iv (subst (ListProperty
              (λ _ x → PL-FormulaBound (proj₂ x) (sv + iv))) (sym (Monoid.assoc (monoid
              (ℕ × PL-Formula)) r1 [ (a , φ) ] r2)) rungbound) s i)

  TransitionCorrect' : (r1 r2 : List (ℕ × PL-Formula))
                     → {sv : ℕ}
                     → (rungs : FinMap sv (r1 ++ r2))
                     → (iv : ℕ)
                     → (rungbound : ListProperty (λ _ x → PL-FormulaBound (proj₂ x) (sv + iv))
                                                 (r1 ++ r2))
                     → (s : Σ[ m ∶ List Bool ] length m ≡ sv)
                     → (i : Σ[ m ∶ List Bool ] length m ≡ iv)
                     → ⟦ mkenv (proj₁ s ++ proj₁ i ++ proj₁ (executeLadder sv iv (r1 ++ r2) s i))
                               ⊧ mktrans-aux (sv + iv) (indicies r1) r2 ⟧pl
  TransitionCorrect' r1 r2 {sv} rungs iv rungbound s i
    = lem-mktrans-aux' (mkenv (proj₁ s ++ proj₁ i ++ proj₁ (executeLadder sv iv (r1 ++ r2) s i)))
                       (sv + iv) r1 r2 (#TransitionCorrect' r1 r2 rungs iv rungbound s i)

constructTransitionFunction : (l : ∃ LadderWellFormed)
                            → Ladder.State (proj₁ l)
                            → Ladder.Input (proj₁ l)
                            → Ladder.State (proj₁ l)
constructTransitionFunction (l , p) s i = executeLadder (Ladder.statevars l) (Ladder.inputvars l)
                                                        (Ladder.rungs l) s (proj₁ i)

TransitionCorrect : (l : ∃ LadderWellFormed)
                  → (s : Ladder.State (proj₁ l))
                  → (i : Ladder.Input (proj₁ l))
                  → ⟦ mkenv (proj₁ s ++ proj₁ (proj₁ i) ++
                             proj₁ (constructTransitionFunction l s i)) ⊧ mktrans (proj₁ l) ⟧pl
TransitionCorrect l s i = TransitionProof.TransitionCorrect' [] _
                            (LadderWellFormed.rungsmap (proj₂ l)) (Ladder.inputvars (proj₁ l))
                            (LadderWellFormed.rungvarbound (proj₂ l)) s (proj₁ i)

abstract
  mkDecTransitionSystem : (l : ∃ LadderWellFormed)
                        → DecidableTransitionSystem (mkTransitionSystem (proj₁ l))
  mkDecTransitionSystem l = record {
                              initialState = constructInitialState l;
                              initialCorrectness = InitialCorrect l;
                              transitionFunction = constructTransitionFunction l;
                              transitionCorrectness = TransitionCorrect l }

  mkDecTrans-init-eq : ∀ l
                     → DecidableTransitionSystem.initialState (mkDecTransitionSystem l)
                       ≡ constructInitialState l
  mkDecTrans-init-eq l = refl

open DecidableTransitionSystem

nthLadderStateCorrect : {l : Ladder}
                      → (dts : DecidableTransitionSystem (mkTransitionSystem l))
                      → ∀ φ
                      → LadderCorrectness l φ
                      → ∀ n inputs
                      → ⟦safety⟧π φ (nthState dts inputs n) (inputs n) (nthState dts inputs (suc n))
nthLadderStateCorrect dts φ = nthStateCorrect dts (⟦safety⟧π φ)
```

```
-- some usefull functions for constructing LadderWellFormed records
isfinmap : ∀ {B} c → (l : List (ℕ × B)) → Bool
isfinmap c []            = true
isfinmap c ((a , b) ∷ l) = (a notinℕ indicies l ∧ (a < c)) ∧ isfinmap c l

mkfinmap : ∀ {B} c → (l : List (ℕ × B)) → T (isfinmap c l) → FinMap c l
mkfinmap c []            = \ _ → []
mkfinmap c ((a , b) ∷ l) = ∧-elim (λ p q → [_↦_]∷_ a b {p = ∧-eliml p}
          {q = ∧-elimr (a notinℕ indicies l) p} (mkfinmap c l q))

isbound : (b : ℕ) → List (ℕ × PL-Formula) → Bool
isbound b []            = true
isbound b ((a , φ) ∷ l) = bound b φ ∧ isbound b l

mkbound : (b : ℕ) → (l : List (ℕ × PL-Formula))
        → T (isbound b l) → ListProperty (const (πPL-FormulaBound b)) l
mkbound b [] = const []
mkbound b ((a , φ) ∷ l) = ∧-elim (λ p q → [ (a , φ) ]< p >∷ (mkbound b l q))
```

```
module CTL.Ladder where

open import CTL.RecordSystem hiding (toState;fromState)
open import CTL.ListGen

open import Coinduction

open import Data.Nat hiding (_<_)
open import Data.Bool renaming (_∧_ to _∧ᵇ_ ; _∨_ to _∨ᵇ_)
open import Data.Vec hiding ([_];lookup;foldr) renaming (_++_ to _++v_)
open import Data.List as List hiding ([_];_++_)
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.List.Util using (repeat;lookup)
open import Data.Fin hiding (_+_;_<_)
open import Data.Fin.Record
open import Data.Fin.EqReasoning
open import Data.String hiding (_==_)

open import Relation.Binary.PropositionalEquality hiding ([_])

open import PropIso renaming (_==_ to nateq)

infixr 7 _∧_
infixr 8 _∨_
infix 9 ~_

data BooleanFormula (n m : ℕ) : Set where
  false true : BooleanFormula n m
  _∧_ _∨_ : (φ ψ : BooleanFormula n m) → BooleanFormula n m
  ~_ : (φ : BooleanFormula n m) → BooleanFormula n m
  $_ : Fin n ⊎ Fin m  → BooleanFormula n m

record Ladder : Set where
  constructor
    ladder
  field
    inp     : ℕ
    state   : ℕ
    initial : Vec Bool state
    rungs   : List (Fin state × BooleanFormula inp state)

  Input : Set
  Input = Vec Bool inp

  State : Set
  State = Vec Bool state

data LadderCTL (n : ℕ) : Set where
  false : LadderCTL n
  _∨_ _∧_ E[_U_] : (φ : LadderCTL n) → (ψ : LadderCTL n) → LadderCTL n
  P[_] : (i : Fin n) → LadderCTL n
  ~ EX EG : (φ : LadderCTL n) → LadderCTL n

data LadderProblem : Set where
  _,_⊧_ : (l : Ladder) → (s : Ladder.State l) → (φ : LadderCTL (Ladder.state l)) → LadderProblem

_[_:=_] : ∀ {n} {A : Set} → Vec A n → Fin n → A → Vec A n
[] [ () := b ]
(x ∷ v) [ zero  := b ] = b ∷ v
(x ∷ v) [ suc a := b ] = x ∷ v [ a := b ]

_[_] : ∀ {n} {A : Set} → Vec A n → Fin n → A
[] [ () ]
(x ∷ v) [ zero  ] = x
(x ∷ v) [ suc i ] = v [ i ]

eval : ∀ {n m} → BooleanFormula n m → Vec Bool n → Vec Bool m → Bool
eval false ξ ζ         = false
eval true ξ ζ          = true
eval (φ ∧ φ₁) ξ ζ      = eval φ ξ ζ ∧ᵇ eval φ₁ ξ ζ
eval (φ ∨ φ₁) ξ ζ      = eval φ ξ ζ ∨ᵇ eval φ₁ ξ ζ
eval (~ φ) ξ ζ         = not (eval φ ξ ζ)
eval ($ (inj₁ x)) ξ ζ = ξ [ x ]
eval ($ (inj₂ x)) ξ ζ = ζ [ x ]

executeRungs : ∀ {n m} → List (Fin n × BooleanFormula m n) → Vec Bool n → Vec Bool m → Vec Bool n
executeRungs [] s i             = s
executeRungs ((a , φ) ∷ r) s i = executeRungs r (s [ a := eval φ i s ]) i

mkTransition : (l : Ladder) → Ladder.State l → Ladder.Input l → Ladder.State l
mkTransition l = executeRungs (Ladder.rungs l)

data LadderRun (l : Ladder) (s : Ladder.State l) : Set where
  next : (i : Ladder.Input l) → ∞ (LadderRun l (mkTransition l s i)) → LadderRun l s

run-decompˡ : (l : Ladder)
```

```
           → {s : Ladder.State l}
           → LadderRun l s
           → Σ (Ladder.Input l) (λ x → LadderRun l (mkTransition l s x))
run-decompˡ l (next i y) = i , ♭ y

run-headˡ : (l : Ladder) → {s : Ladder.State l} → LadderRun l s → Ladder.Input l
run-headˡ l r = proj₁ (run-decompˡ l r)

run-tailˡ : (l : Ladder)
          → {s : Ladder.State l}
          → (r : LadderRun l s)
          → LadderRun l (mkTransition l s (run-headˡ l r))
run-tailˡ l r = proj₂ (run-decompˡ l r)

nthˡ : (l : Ladder) → {s : Ladder.State l} → ℕ → LadderRun l s → Ladder.State l
nthˡ l {s} zero r = s
nthˡ l (suc n) r = nthˡ l n (run-tailˡ l r)

⟦_⟧ˡ  : LadderProblem → Set
⟦ l , s ⊨ false ⟧ˡ       = ⊥
⟦ l , s ⊨ ~ φ ⟧ˡ          = ¬ ⟦ l , s ⊨ φ ⟧ˡ
⟦ l , s ⊨ (φ ∨ φ₁) ⟧ˡ     = ⟦ l , s ⊨ φ ⟧ˡ ⊎ ⟦ l , s ⊨ φ₁ ⟧ˡ
⟦ l , s ⊨ (φ ∧ φ₁) ⟧ˡ     = ⟦ l , s ⊨ φ ⟧ˡ × ⟦ l , s ⊨ φ₁ ⟧ˡ
⟦ l , s ⊨ P[ i ] ⟧ˡ       = T (s [ i ])
⟦ l , s ⊨ EX φ ⟧ˡ         = Σ[ run ∶ LadderRun l s ] ⟦ l , nthˡ l 1 run ⊨ φ ⟧ˡ
⟦ l , s ⊨ EG φ ⟧ˡ         = Σ[ run ∶ LadderRun l s ] (∀ i → ⟦ l , nthˡ l i run ⊨ φ ⟧ˡ)
⟦ l , s ⊨ E[ φ U φ₁ ] ⟧ˡ = Σ[ run ∶ LadderRun l s ] Σ[ k ∶ ℕ ]
                                  (((j : ℕ) → T (j < k) → ⟦ l , nthˡ l j run ⊨ φ ⟧ˡ)
                                      × ⟦ l , nthˡ l k run ⊨ φ₁ ⟧ˡ)

toRecord : ∀ {n} → Vec Bool n → Record (repeat 2 n)
toRecord []       = tt
toRecord (x ∷ v) = (fromBool x) , (toRecord v)

fromRecord : ∀ {n} → Record (repeat 2 n) → Vec Bool n
fromRecord {zero} r         = []
fromRecord {suc n} (x , y) = (toBool x) ∷ (fromRecord y)

record-iso1 : ∀ {n} → (v : Vec Bool n) → fromRecord (toRecord v) ≡ v
record-iso1 []       = refl
record-iso1 (x ∷ v) = cong₂ _∷_ (tobool-iso1 x) (record-iso1 v)

record-iso2 : ∀ {n} → (v : Record (repeat 2 n)) → toRecord (fromRecord {n} v) ≡ v
record-iso2 {0} tt          = refl
record-iso2 {suc n} (x , v) = cong₂ _,_ (tobool-iso2 x) (record-iso2 {n} v)

fliprecord1 : ∀ {n} → {v : Vec Bool n}
            → {w :  Record (repeat 2 n)}
            → v ≡ fromRecord w → toRecord v ≡ w
fliprecord1 {._} {[]} {tt} eq = refl
fliprecord1 {._} {x ∷ v} {y , w} eq
  = cong₂ _,_ (flip-toBool1 _ _ (cong head eq)) (fliprecord1 (cong tail eq))

fliprecord2 : ∀ {n} → {v : Vec Bool n}
            → {w :  Record (repeat 2 n)}
            → toRecord v ≡ w → v ≡ fromRecord w
fliprecord2 {._} {[]} {w} eq = refl
fliprecord2 {._} {x ∷ v} {y , w} eq
  = cong₂ _∷_ (flip-toBool2 _ _ (cong proj₁ eq)) (fliprecord2 (cong proj₂ eq))

toState : (l : Ladder) → Ladder.State l → Record (repeat 2 (Ladder.state l))
toState l = toRecord

fromState : (l : Ladder) → Record (repeat 2 (Ladder.state l)) → Ladder.State l
fromState l = fromRecord

toInput : (l : Ladder) → Ladder.Input l → Record (repeat 2 (Ladder.inp l))
toInput l = toRecord

fromInput : (l : Ladder) → Record (repeat 2 (Ladder.inp l)) → Ladder.Input l
fromInput l = fromRecord

toSymFSM : (l : Ladder) → FSMʳ
toSymFSM l = frm (repeat 2 (Ladder.state l))
                 (const (repeat 2 (Ladder.inp l)))
                 (toState l (Ladder.initial l) ∷ [])
                 ((λ s i → toState l (mkTransition l (fromState l s) (fromInput l i))))

toRunˡ'' : (l : Ladder)
         → (s t : Ladder.State l)
         → (fromState l (toState l t)) ≡ s
         → LadderRun l s
         → Runʳ (toSymFSM l) (toState l t)
toRunˡ'' l  ._ t refl (next i x)
  = next (toState l t)
         (toInput l i)
         (♯ toRunˡ'' l _ _ (trans (record-iso1 (mkTransition l (fromRecord (toRecord t))
                                                               (fromRecord (toRecord i))))
```

```
                                (cong (mkTransition l (fromRecord (toRecord t))) (record-iso1 i)))
                          (♭ x))

toRunˡ : (l : Ladder) → (s : Ladder.State l) → LadderRun l s → Runʳ (toSymFSM l) (toState l s)
toRunˡ l s r = toRunˡ' l _ _ (record-iso1 s) r

fromRunˡ' : (l : Ladder)
          → (s t : Ladder.State l)
          → (fromState l (toState l t) ≡ s)
          → Runʳ (toSymFSM l) (toState l t)
          → LadderRun l s
fromRunˡ' l ._ t refl (next ._ i x)
  = next (fromInput l i)
         (♯ fromRunˡ' l _ _ (record-iso1 (mkTransition l (fromRecord (toRecord t))
                                                         (fromRecord i)))
                            (♭ x))

fromRunˡ : (l : Ladder)
         → (s : Ladder.State l)
         → Runʳ (toSymFSM l) (toState l s)
         → LadderRun l s
fromRunˡ l s r = fromRunˡ' l _ _ (record-iso1 s) r

lem-nth'ˡ : (l : Ladder)
          → {s t : Ladder.State l}
          → (r : LadderRun l s)
          → (n : ℕ)
          → (eq : fromState l (toState l t) ≡ s)
          → nthˡ l n r ≡ fromState l (nthʳ (toSymFSM l) n (toRunˡ' l s t eq r))
lem-nth'ˡ l r zero refl = refl
lem-nth'ˡ l {._} {t} (next i r) (suc n) refl = lem-nth'ˡ l (♭ r) n _

lem-nthˡ : (l : Ladder)
         → {s : Ladder.State l}
         → (r : LadderRun l s)
         → (n : ℕ)
         → nthˡ l n r ≡ fromState l (nthʳ (toSymFSM l) n (toRunˡ l s r))
lem-nthˡ l {s} r n = lem-nth'ˡ l {s} {s} r n (record-iso1 s)

lem-nth''ˡ' : (l : Ladder)
           → {s t : Ladder.State l}
           → (r : Runʳ (toSymFSM l) (toState l t))
           → (n : ℕ)
           → (eq : fromState l (toState l t) ≡ s)
           → fromState l (nthʳ (toSymFSM l) n r) ≡ nthˡ l n (fromRunˡ' l s t eq r)
lem-nth''ˡ' l r zero refl = refl
lem-nth''ˡ' l {._} {t} (next ._ a r) (suc n) refl = lem-nth''ˡ' l (♭ r) n _

lem-nth''ˡ : (l : Ladder)
           → {s : Ladder.State l}
           → (r : Runʳ (toSymFSM l) (toState l s))
           → (n : ℕ)
           → fromState l (nthʳ (toSymFSM l) n r) ≡ nthˡ l n (fromRunˡ l _ r)
lem-nth''ˡ l r n = lem-nth''ˡ' l r n _

const-two : ∀ {n} → (i : Fin n) → [ id , const 0 ]′ (lookup (repeat 2 n) (toℕ i)) ≡ 2
const-two zero    = refl
const-two (suc i) = const-two i

one : ∀ {n} → (i : Fin n) → Fin ([ id , const 0 ]′ (lookup (repeat 2 n) (toℕ i)))
one i = subst Fin (sym (const-two i)) (suc zero)

toSymCTL : {n : ℕ} → LadderCTL n → CTLʳ (repeat 2 n)
toSymCTL false       = false
toSymCTL (~ φ)       = ~ (toSymCTL φ)
toSymCTL (φ ∨ φ₁)    = toSymCTL φ ∨ toSymCTL φ₁
toSymCTL (φ ∧ φ₁)    = toSymCTL φ ∧ toSymCTL φ₁
toSymCTL P[ i ]      = P[ toℕ i == one i ]
toSymCTL (EX φ)      = EX (toSymCTL φ)
toSymCTL (EG φ)      = EG (toSymCTL φ)
toSymCTL E[ φ U φ₁ ] = E[ toSymCTL φ U toSymCTL φ₁ ]

toSymCTLProblem : LadderProblem → CTLProblemʳ
toSymCTLProblem (l , s ⊧ φ) = (toSymFSM l) , (toState l s) ⊧ʳ (toSymCTL φ)

evalˡ : LadderProblem → Bool
evalˡ l = evalʳ (toSymCTLProblem l)

private
  ladder-correct-proposition1 : ∀ {n}
                              → (s : Vec Bool n)
                              → (i : Fin n)
                              → T (s [ i ])
                              → T (record-lookup (repeat 2 n) (toRecord s)
                                                 (toℕ i , subst Fin (sym (const-two i)) (suc zero)))
  ladder-correct-proposition1 [] () p
  ladder-correct-proposition1 (x ∷ s) zero p rewrite Tb p = tt
  ladder-correct-proposition1 (x ∷ s) (suc i) p = ladder-correct-proposition1 s i p
```

```
  ladder-correct-proposition2 : ∀ {n}
                              → (s : Vec Bool n)
                              → (i : Fin n)
                              → T (record-lookup (repeat 2 n) (toRecord s)
                                                 (toℕ i , subst Fin (sym (const-two i)) (suc zero)))
                              → T (s [ i ])
  ladder-correct-proposition2 [] () p
  ladder-correct-proposition2 (true :: s) zero p = tt
  ladder-correct-proposition2 (false :: s) zero p = p
  ladder-correct-proposition2 (x :: s) (suc i) p = ladder-correct-proposition2 s i p

mutual
  ladder-correct1 : ∀ (P : LadderProblem) → ⟦ P ⟧ˡ → ⟦ toSymCTLProblem P ⟧ʳ
  ladder-correct1 (l , s ⊧ false) q     = q
  ladder-correct1 (l , s ⊧ ~ φ) q       = q ∘ ladder-correct2 (l , s ⊧ φ)
  ladder-correct1 (l , s ⊧ (φ ∨ φ₁)) q = Sum.map (ladder-correct1 (l , s ⊧ φ))
                                                 (ladder-correct1 (l , s ⊧ φ₁)) q
  ladder-correct1 (l , s ⊧ (φ ∧ φ₁)) q = Prod.map (ladder-correct1 (l , s ⊧ φ))
                                                  (ladder-correct1 (l , s ⊧ φ₁)) q
  ladder-correct1 (l , s ⊧ P[ i ]) q    = ladder-correct-proposition1 s i q
  ladder-correct1 (l , s ⊧ EX φ) q
    = Prod.map (toRunˡ l s)
               (λ {r} → subst (λ s' → ⟦ toSymFSM l , s' ⊧ʳ toSymCTL φ ⟧ʳ)
                              (cong (toState l) (trans (lem-nthˡ l r 1) (record-iso1 _))) ∘
                         ladder-correct1 (l , nthˡ l 1 r ⊧ φ)) q
  ladder-correct1 (l , s ⊧ EG φ) q
    = Prod.map (toRunˡ l s)
               (λ {r} → λ x n → subst (λ s' → ⟦ toSymFSM l , s' ⊧ʳ toSymCTL φ ⟧ʳ)
                                      (fliprecord1 (lem-nthˡ l r n))
                                      (ladder-correct1 (l , nthˡ l n r ⊧ φ) (x n))) q
  ladder-correct1 (l , s ⊧ E[ φ U φ₁ ]) q
    = Prod.map (toRunˡ l s)
               (λ {r} → Prod.map id (λ {i} → Prod.map (λ x j x₁ →
               subst (λ s' → ⟦ toSymFSM l , s' ⊧ʳ toSymCTL φ ⟧ʳ)
                     (fliprecord1 (lem-nthˡ l r j))
                     (ladder-correct1 (l , nthˡ l j r ⊧ φ) (x j x₁)))
               (subst (λ s' → ⟦ toSymFSM l , s' ⊧ʳ toSymCTL φ₁ ⟧ʳ)
                      (fliprecord1 (lem-nthˡ l r i)) ∘ ladder-correct1 (l , nthˡ l i r ⊧ φ₁)))) q

  ladder-correct2 : ∀ (P : LadderProblem) → ⟦ toSymCTLProblem P ⟧ʳ → ⟦ P ⟧ˡ
  ladder-correct2 (l , s ⊧ false) q     = q
  ladder-correct2 (l , s ⊧ ~ φ) q       = q ∘ ladder-correct1 (l , s ⊧ φ)
  ladder-correct2 (l , s ⊧ (φ ∨ φ₁)) q = Sum.map (ladder-correct2 (l , s ⊧ φ))
                                                 (ladder-correct2 (l , s ⊧ φ₁)) q
  ladder-correct2 (l , s ⊧ (φ ∧ φ₁)) q = Prod.map (ladder-correct2 (l , s ⊧ φ))
                                                  (ladder-correct2 (l , s ⊧ φ₁)) q
  ladder-correct2 (l , s ⊧ P[ i ]) q    = ladder-correct-proposition2 s i q
  ladder-correct2 (l , s ⊧ EX φ) q
    = Prod.map (fromRunˡ l s)
               (λ {r} → subst (λ s' → ⟦ l , s' ⊧ φ ⟧ˡ)
                              (trans (sym (record-iso1 _)) (lem-nthˡ' l r 1)) ∘
                         ladder-correct2 (l , _ ⊧ φ)) q
  ladder-correct2 (l , s ⊧ EG φ) q
    = Prod.map (fromRunˡ l s)
               (λ {r} → λ x i → subst (λ s' → ⟦ l , s' ⊧ φ ⟧ˡ)
                                      (lem-nthˡ' l r i)
                                      (ladder-correct2 (l , _ ⊧ φ)
                                        (subst (λ s' → ⟦ toSymFSM l , s' ⊧ʳ toSymCTL φ ⟧ʳ)
                                               (sym (record-iso2 {Ladder.state l} _)) (x i)))) q
  ladder-correct2 (l , s ⊧ E[ φ U φ₁ ]) q
    = Prod.map (fromRunˡ l s)
               (λ {r} → Prod.map id (λ {i} → Prod.map (λ x j x₁ →
               subst (λ s' → ⟦ l , s' ⊧ φ ⟧ˡ)
                     (lem-nthˡ' l r j)
                     (ladder-correct2 (l , _ ⊧ φ)
                                      (subst (λ s' → ⟦ toSymFSM l , s' ⊧ʳ toSymCTL φ ⟧ʳ)
                                             (sym (record-iso2 {Ladder.state l} _))
                                             (x j x₁))))
               (λ x → subst (λ s' → ⟦ l , s' ⊧ φ₁ ⟧ˡ)
                            (lem-nthˡ' l r i)
                            (ladder-correct2 (l , _ ⊧ φ₁)
                                             (subst (λ s' → ⟦ toSymFSM l , s' ⊧ʳ toSymCTL φ₁ ⟧ʳ)
                                                    (sym (record-iso2 {Ladder.state l} _)) x))))) q

soundnessˡ : (P : LadderProblem) → (T (evalˡ P)) → ⟦ P ⟧ˡ
soundnessˡ P p = ladder-correct2 P (soundnessʳ (toSymCTLProblem P) p)

completenessˡ : (P : LadderProblem) → ⟦ P ⟧ˡ → (T (evalˡ P))
completenessˡ P p = completenessʳ (toSymCTLProblem P) (ladder-correct1 P p)

AG : {n : ℕ} → LadderCTL n → LadderCTL n
AG φ = ~ E[ ~ false U ~ φ ]

from_toplus_ : ℕ → ℕ → List ℕ
from n toplus zero    = []
from n toplus (suc m) = n :: from suc n toplus m
```

```
private primitive primShowNat : ℕ → String

showfin : {n : ℕ} → Fin n → String
showfin = primShowNat ∘ toℕ

_elem_by_ : {A : Set} → A → List A → (A → A → Bool) → Bool
x elem [] by f      = false
x elem a ∷ as by f = f x a ∨ᵇ (x elem as by f)

showformula : {n m : ℕ} → List ℕ → BooleanFormula n m → String
showformula v (φ ∧ ψ)              = "(" ++ showformula v φ ++ " & " ++ showformula v ψ ++ ")"
showformula v (φ ∨ ψ)              = "(" ++ showformula v φ ++ " | " ++ showformula v ψ ++ ")"
showformula v (~ φ)                = "!(" ++ showformula v φ ++ ")"
showformula v false                = "FALSE"
showformula v true                 = "TRUE"
showformula v ($ (inj₁ y))         = "ivar" ++ showfin y
showformula v ($ (inj₂ y)) with toℕ y elem v by nateq
showformula v ($ (inj₂ y)) | true  = "next(var" ++ showfin y ++ ")"
showformula v ($ (inj₂ y)) | false = "var" ++ showfin y

showladder : {n m : ℕ} → List ℕ → List (Fin n × BooleanFormula m n) → String
showladder v []              = ""
showladder v ((a , y) ∷ as) = "    next(var" ++
                                showfin a ++ ") := " ++ showformula v y ++ ";\n" ++
                                showladder (toℕ a ∷ v) as
showbool : Bool → String
showbool true  = "TRUE"
showbool false = "FALSE"

showinit : (n : ℕ) → ℕ → Vec Bool n → String
showinit zero m []                = ""
showinit (suc zero) m (b ∷ [])    = "    init(var" ++ primShowNat m ++ ") := " ++ showbool b ++ ";\n"
showinit (suc (suc n)) m (b ∷ y) = "    init(var" ++ primShowNat m ++ ") := " ++ showbool b ++
                                   ";\n" ++ showinit (suc n) (suc m) y

showspec : {n : ℕ} → LadderCTL n → String
showspec false      = "FALSE"
showspec (~ φ)      = "!(" ++ showspec φ ++ ")"
showspec (φ ∨ ψ)    = "(" ++ showspec φ ++ ") | (" ++ showspec ψ ++ ")"
showspec (φ ∧ ψ)    = "(" ++ showspec φ ++ ") & (" ++ showspec ψ ++ ")"
showspec P[ i ]     = "var" ++ showfin i
showspec (EX φ)     = "EX (" ++ showspec φ ++ ")"
showspec (EG φ)     = "EG (" ++ showspec φ ++ ")"
showspec E[ φ U ψ ] = "E[" ++ showspec φ ++ " U " ++ showspec ψ ++ "]"

genSMVFromLadder : LadderProblem → String
genSMVFromLadder (ladder inp state init rungs , s ⊨ φ) = header ++ ivar ++ var ++ assign ++ spec
  where
    header = "MODULE main\n"

    ivar = "IVAR\n" ++ foldr _++_ "" (List.map (λ n → "    ivar" ++ primShowNat n ++
              " : boolean;\n") (from 0 toplus inp))

    var = "VAR\n" ++ foldr _++_ "" (List.map (λ n → "    var" ++ primShowNat n ++
              " : boolean;\n") (from 0 toplus state))

    assign = "ASSIGN\n" ++ showinit state 0 init ++ showladder [] rungs

    spec = "SPEC\n    " ++ showspec φ ++ ";\n"

{-# BUILTIN UNIT ⊤ #-}
{-# BUILTIN TRIV tt #-}
{-# BUILTIN EMPTY ⊥ #-}
{-# BUILTIN ATOM T #-}

TOOL : String
TOOL = "nusmv"

{-# BUILTIN ATPTOOL TOOL #-}
{-# BUILTIN ATPPROBLEM LadderProblem #-}
{-# BUILTIN ATPINPUT genSMVFromLadder #-}
{-# BUILTIN ATPDECPROC evalᴸ #-}
{-# BUILTIN ATPSEMANTICS ⟦_⟧ᴸ #-}
{-# BUILTIN ATPSOUND soundnessᴸ #-}
{-# BUILTIN ATPCOMPLETE completenessᴸ #-}
```

```
module CTL.Pelicon where

open import CTL.Ladder
open import Data.Fin
open import Data.Fin.Arithmetic
open import Data.Vec
open import Data.Nat
open import Data.Product
open import Data.List
open import Data.Sum
open import Data.Bool hiding (_∧_;_∨_)

open import PropIso hiding (_$_)

inputlatchcount : ℕ
inputlatchcount = 1

Input : Set
Input = Fin inputlatchcount

statelatchcount : ℕ
statelatchcount = 4

State : Set
State = Fin statelatchcount

Pressed : Input
Pressed = fromℕ< 0 tt

Requested : State
Requested = fromℕ< 0 tt

Crossing : State
Crossing = fromℕ< 1 tt

PLightG : State
PLightG = fromℕ< 2 tt

TLightG : State
TLightG = fromℕ< 3 tt

private
  infix 6 _:=_
  _:=_ : State → BooleanFormula 1 4 → State × BooleanFormula 1 4
  _:=_ = _,_

rungs : List (State × BooleanFormula 1 4)
rungs = Crossing  := (~ ($ (inj₂ Crossing)) ∧ $ (inj₂ Requested))
      ∷ Requested := ($ (inj₁ Pressed) ∧ ~ ($ (inj₂ Crossing)))
      ∷ PLightG   := ($ (inj₂ Crossing))
      ∷ TLightG   := (~ ($ (inj₂ Crossing)) ∧ ~ ($ (inj₂ Requested)))
      ∷ []

pelicon-ladder : Ladder
pelicon-ladder = ladder 1 4 (false ∷ false ∷ false ∷ false ∷ []) rungs

φ : LadderCTL 4
φ = AG (~ P[ PLightG ] ∨ ~ P[ TLightG ])

prob : LadderProblem
prob = pelicon-ladder , Ladder.initial pelicon-ladder ⊨ φ

X : T (evalᴸ prob)
X = tt
```

```
module Pelicon.PeliconModel where

open import Data.Nat
open import Data.Product
open import Data.Sum
open import Data.Nat.Properties

open import Relation.Binary.PropositionalEquality
open import Relation.Decidable

open import Function

open import Algebra.Structures

{-
  Pedestrial Light Controlled Corssing -- Simulator

              P1
  -------------------------
      T1    = MUX =    T2
  -------------------------
              P2
-}


-- some needed lemmata
+-id : ∀ a b → a + b ∸ a ≡ b
+-id zero b = refl
+-id (suc a) b = +-id a b

stream-simplify : ∀ (X Y : ℕ → ℕ) → X 0 ≡ Y 0
                → (∀ n → X (suc n) ≡ X n + Y (suc n) ∸ Y n) → ∀ n → X n ≡ Y n
stream-simplify X Y base-eq stream-def zero = base-eq
stream-simplify X Y base-eq stream-def (suc n) rewrite stream-def n
                                                     | stream-simplify X Y base-eq stream-def n
                                                         = +-id (Y n) (Y (suc n))

simplify : ∀ (V W X Y Z : ℕ → ℕ)
         → V 0 ≡ 0 → W 0 ≡ 0 → X 0 ≡ 0
         → (∀ t → W t ≡ Z (suc t))
         → (∀ t → X t ≡ Y (suc t))
         → (∀ t → V (suc t) ≡ V t + (W (suc t) + X (suc t)) ∸ (Y (suc t) + Z (suc t)))
         → ∀ t → V t ≡ W t + X t
simplify V W X Y Z baseV baseW baseX WZeq XYeq stream-def
  = stream-simplify V (λ t → W t + X t)
                    (trans baseV (sym (cong₂ _+_ baseW baseX)))
                    (λ t → trans (stream-def t) (cong (λ k → V t + (W (suc t) + X (suc t)) ∸ k)
                            (trans (cong₂ _+_ (sym (XYeq t)) (sym (WZeq t)))
                              (IsCommutativeMonoid.comm
                                (IsCommutativeSemiring.+-isCommutativeMonoid isCommutativeSemiring)
                                (X t) (W t)))))

data Aspect : Set where
  red green : Aspect

aspect-dec : Decidable (λ x y → type-signature Aspect x ≡ y)
aspect-dec red red = yes refl
aspect-dec red green = no (λ ())
aspect-dec green red = no (λ ())
aspect-dec green green = yes refl

data Area : Set where
  P1 P2 T1 T2 MUX : Area

record State : Set where
  field
    numcars : Area → ℕ
    numpeds : Area → ℕ
    movingcars : Area → Area → ℕ
    movingpeds : Area → Area → ℕ
    traffic : Aspect
    pedest : Aspect

open State

record Controller : Set where
  field
    nthState : ℕ → State

    taxm1 : ∀ t → traffic (nthState t) ≡ red
          → movingcars (nthState (suc t)) T1 MUX ≡ 0
            × movingcars (nthState (suc t)) T2 MUX ≡ 0
    taxm2 : ∀ t → movingcars (nthState t) T1 MUX ≡ movingcars (nthState (suc t)) MUX T2
    taxm3 : ∀ t → movingcars (nthState t) T2 MUX ≡ movingcars (nthState (suc t)) MUX T1
    taxm4 : ∀ t → movingcars (nthState (suc t)) T1 MUX ≤ numcars (nthState t) T1
    taxm5 : ∀ t → movingcars (nthState (suc t)) T2 MUX ≤ numcars (nthState t) T2
    taxm6 : ∀ t → numcars (nthState (suc t)) MUX ≡ numcars (nthState t) MUX
```

```
                  + (movingcars (nthState (suc t)) T1 MUX + movingcars (nthState (suc t)) T2 MUX)
                  ≐ (movingcars (nthState (suc t)) MUX T1 + movingcars (nthState (suc t)) MUX T2)


    paxm1 : ∀ t → pedest (nthState t) ≡ red
          → movingpeds (nthState (suc t)) P1 MUX ≡ 0 × movingpeds (nthState (suc t)) P2 MUX ≡ 0
    paxm2 : ∀ t → movingpeds (nthState t) P1 MUX ≡ movingpeds (nthState (suc t)) MUX P2
    paxm3 : ∀ t → movingpeds (nthState t) P2 MUX ≡ movingpeds (nthState (suc t)) MUX P1
    paxm4 : ∀ t → movingpeds (nthState (suc t)) P1 MUX ≤ numpeds (nthState t) P1
    paxm5 : ∀ t → movingpeds (nthState (suc t)) P2 MUX ≤ numpeds (nthState t) P2
    paxm6 : ∀ t → numpeds (nthState (suc t)) MUX ≡ numpeds (nthState t) MUX
                  + (movingpeds (nthState (suc t)) P1 MUX + movingpeds (nthState (suc t)) P2 MUX)
                  ≐ (movingpeds (nthState (suc t)) MUX P1 + movingpeds (nthState (suc t)) MUX P2)

    safetyp : ∀ t → (movingcars (nthState t) T1 MUX ≡ 0 × movingcars (nthState t) T2 MUX ≡ 0)
                  ⊎ (movingpeds (nthState t) P1 MUX ≡ 0 × movingpeds (nthState t) P2 MUX ≡ 0)

open Controller

record Safe (c : Controller) : Set where
  ψ : ℕ → Set
  ψ n = numcars (nthState c n) MUX ≡ 0 ⊎ numpeds (nthState c n) MUX ≡ 0

  field
    initnousers : (movingcars (nthState c 0) T1 MUX ≡ 0 × movingcars (nthState c 0) T2 MUX ≡ 0)
                × (movingpeds (nthState c 0) P1 MUX ≡ 0 × movingpeds (nthState c 0) P2 MUX ≡ 0)
                × numcars (nthState c 0) MUX ≡ 0 × numpeds (nthState c 0) MUX ≡ 0

  CarMuxStream : ∀ n
               → numcars (nthState c n) MUX
                 ≡ movingcars (nthState c n) T1 MUX + movingcars (nthState c n) T2 MUX
  CarMuxStream = simplify (λ t → numcars (nthState c t) MUX)
                          (λ t → movingcars (nthState c t) T1 MUX)
                          (λ t → movingcars (nthState c t) T2 MUX)
                          (λ t → movingcars (nthState c t) MUX T1)
                          (λ t → movingcars (nthState c t) MUX T2)
                          (proj₁ $ proj₂ $ proj₂ initnousers)
                          (proj₁ $ proj₁ initnousers)
                          (proj₂ $ proj₁ initnousers)
                          (taxm2 c) (taxm3 c) (taxm6 c)

  PedMuxStream : ∀ n
               → numpeds (nthState c n) MUX
                 ≡ movingpeds (nthState c n) P1 MUX + movingpeds (nthState c n) P2 MUX
  PedMuxStream = simplify (λ t → numpeds (nthState c t) MUX)
                          (λ t → movingpeds (nthState c t) P1 MUX)
                          (λ t → movingpeds (nthState c t) P2 MUX)
                          (λ t → movingpeds (nthState c t) MUX P1)
                          (λ t → movingpeds (nthState c t) MUX P2)
                          (proj₂ $ proj₂ $ proj₂ initnousers)
                          (proj₁ $ proj₁ $ proj₂ initnousers)
                          (proj₂ $ proj₁ $ proj₂ initnousers)
                          (paxm2 c) (paxm3 c) (paxm6 c)

  IsSafe : ∀ t → ψ t
  IsSafe t = [ (λ x → inj₁ (trans (CarMuxStream t) (cong₂ _+_ (proj₁ x) (proj₂ x)))) ,
               (λ x → inj₂ (trans (PedMuxStream t) (cong₂ _+_ (proj₁ x) (proj₂ x))))
             ]′ (safetyp c t)
```

```
module Pelicon.Ladder where

open import Data.List
open import Data.Nat
open import Data.Product
open import Data.Bool
open import Data.Unit

open import Ladder.Decidable
open import Ladder.Core

open import TransitionSystem
open import TransitionSystem.Decidable

open import Boolean.Formula

CROSSING = 0
REQ = 1
TLIGHT = 2
PLIGHT = 3
PRESSED = 4

rungs : List (ℕ × PL-Formula)
rungs = (CROSSING , ~ (¥ CROSSING) && ¥ REQ)
      ∷ (REQ , ¥ PRESSED && ~ (¥ CROSSING))
      ∷ (TLIGHT , ~ (¥ CROSSING) && ~ (¥ REQ))
      ∷ (PLIGHT , ¥ CROSSING)
      ∷ []

initstate : List (ℕ × Bool)
initstate = (CROSSING , false)
          ∷ (REQ , false)
          ∷ (TLIGHT , false)
          ∷ (PLIGHT , false)
          ∷ []

pelicon-ladder : Ladder
pelicon-ladder = ladder 4 1 rungs initstate ¥true

pelicon-fsm : TransitionSystem (Ladder.Input pelicon-ladder)
pelicon-fsm = mkTransitionSystem pelicon-ladder


PeliconLadderWellFormed : LadderWellFormed pelicon-ladder
PeliconLadderWellFormed =
  record { initialmap = mkfinmap (Ladder.statevars pelicon-ladder)
                                 (Ladder.initialstate pelicon-ladder) tt;
           rungsmap = mkfinmap (Ladder.statevars pelicon-ladder)
                               (Ladder.rungs pelicon-ladder) tt;
           rungvarbound = mkbound (Ladder.statevars pelicon-ladder + Ladder.inputvars pelicon-ladder)
                                  (Ladder.rungs pelicon-ladder) tt
         }

PeliconDecidableLadder : DecidableTransitionSystem pelicon-fsm
PeliconDecidableLadder = mkDecTransitionSystem (pelicon-ladder , PeliconLadderWellFormed)
```

```
module Pelicon.State where

open import Pelicon.Ladder
open import Pelicon.PeliconModel

open import Boolean.Formula

open import Ladder.Decidable
open import Ladder.Core

open import TransitionSystem.Decidable renaming (DecidableTransitionSystem to DTS)
open import TransitionSystem

open import Data.List
open import Data.Nat
open import Data.Product
open import Data.Bool
open import Data.Sum

open import Relation.Binary.PropositionalEquality

open import PropIso

≤-id : ∀ n → n ≤ n
≤-id zero = z≤n
≤-id (suc n) = s≤s (≤-id n)

Input : Set
Input = ℕ → Ladder.Input pelicon-ladder

UsersInput : Set
UsersInput = ℕ → ℕ × ℕ × ℕ × ℕ

πT1 : ℕ × ℕ × ℕ × ℕ → ℕ
πT1 = proj₁

πT2 : ℕ × ℕ × ℕ × ℕ → ℕ
πT2 = proj₁ ∘ proj₂

πP1 : ℕ × ℕ × ℕ × ℕ → ℕ
πP1 = proj₁ ∘ proj₂ ∘ proj₂

πP2 : ℕ × ℕ × ℕ × ℕ → ℕ
πP2 = proj₂ ∘ proj₂ ∘ proj₂

sigmap : Bool → Aspect
sigmap true = green
sigmap false = red

lem-sigmap : ∀ b → T b → sigmap b ≡ green
lem-sigmap true _ = refl
lem-sigmap false ()

lem-sigmap' : ∀ b → ¬ T b → sigmap b ≡ red
lem-sigmap' false p = refl
lem-sigmap' true p = ⊥-elim $ p _

archCarAspect : Ladder.State pelicon-ladder → Aspect
archCarAspect s = sigmap $ eval-pl (mkenv (proj₁ s)) $ ¥ TLIGHT

archPedAspect : Ladder.State pelicon-ladder → Aspect
archPedAspect s = sigmap $ eval-pl (mkenv (proj₁ s)) $ ¥ PLIGHT

carmoving' : (Area → Area → ℕ) → (Area → ℕ)→ Ladder.State pelicon-ladder → Area → Area → ℕ
carmoving' moving position s P1  a'  = 0
carmoving' moving position s P2  a'  = 0
carmoving' moving position s T1  P1  = 0
carmoving' moving position s T1  P2  = 0
carmoving' moving position s T1  T1  = 0
carmoving' moving position s T1  T2  = 0
carmoving' moving position s T1  MUX with archCarAspect s
carmoving' moving position s T1  MUX | red = 0
carmoving' moving position s T1  MUX | green = position T1
carmoving' moving position s T2  P1  = 0
carmoving' moving position s T2  P2  = 0
carmoving' moving position s T2  T1  = 0
carmoving' moving position s T2  T2  = 0
carmoving' moving position s T2  MUX with archCarAspect s
carmoving' moving position s T2  MUX | red = 0
carmoving' moving position s T2  MUX | green = position T2
carmoving' moving position s MUX P1  = 0
carmoving' moving position s MUX P2  = 0
carmoving' moving position s MUX T1  = moving T2 MUX
carmoving' moving position s MUX T2  = moving T1 MUX
carmoving' moving position s MUX MUX = 0

carposition' : (ℕ × ℕ × ℕ × ℕ) → (Area → Area → ℕ) → (Area → ℕ) → Area → ℕ
```

```
carposition' inp moving position P1  = 0
carposition' inp moving position P2  = 0
carposition' inp moving position T1  = position T1 + πT1 inp ∸ moving T1 MUX
carposition' inp moving position T2  = position T2 + πT2 inp ∸ moving T2 MUX
carposition' inp moving position MUX = position MUX + (moving T1 MUX + moving T2 MUX)
                                                    ∸ (moving MUX T1 + moving MUX T2)
mutual
   carmoving : Input → UsersInput → ℕ → Area → Area → ℕ
   carmoving i ui zero    = \ _ _ → 0
   carmoving i ui (suc t) = carmoving' (carmoving i ui t)
                                       (carposition i ui t)
                                       (DTS.nthState PeliconDecidableLadder i t)

   carposition : Input → UsersInput → ℕ → Area → ℕ
   carposition i ui zero    = λ _ → 0
   carposition i ui (suc t) = carposition' (ui t) (carmoving i ui (suc t))
                                           (carposition i ui t)

pedestmoving' : (Area → Area → ℕ) → (Area → ℕ) → Ladder.State pelicon-ladder → Area → Area → ℕ
pedestmoving' moving position s T1  a'  = 0
pedestmoving' moving position s T2  a'  = 0
pedestmoving' moving position s P1  P1  = 0
pedestmoving' moving position s P1  P2  = 0
pedestmoving' moving position s P1  T1  = 0
pedestmoving' moving position s P1  T2  = 0
pedestmoving' moving position s P1  MUX with archPedAspect s
pedestmoving' moving position s P1  MUX | red = 0
pedestmoving' moving position s P1  MUX | green = position P1
pedestmoving' moving position s P2  P1  = 0
pedestmoving' moving position s P2  P2  = 0
pedestmoving' moving position s P2  T1  = 0
pedestmoving' moving position s P2  T2  = 0
pedestmoving' moving position s P2  MUX with archPedAspect s
pedestmoving' moving position s P2  MUX | red = 0
pedestmoving' moving position s P2  MUX | green = position P2
pedestmoving' moving position s MUX T1  = 0
pedestmoving' moving position s MUX T2  = 0
pedestmoving' moving position s MUX P1  = moving P2 MUX
pedestmoving' moving position s MUX P2  = moving P1 MUX
pedestmoving' moving position s MUX MUX = 0

pedestposition' : (ℕ × ℕ × ℕ × ℕ) → (Area → Area → ℕ) → (Area → ℕ) → Area → ℕ
pedestposition' inp moving position T1  = 0
pedestposition' inp moving position T2  = 0
pedestposition' inp moving position P1  = position P1 + πP1 inp ∸ moving P1 MUX
pedestposition' inp moving position P2  = position P2 + πP2 inp ∸ moving P2 MUX
pedestposition' inp moving position MUX = position MUX + (moving P1 MUX + moving P2 MUX)
                                                       ∸ (moving MUX P1 + moving MUX P2)
mutual
   pedestmoving : Input → UsersInput → ℕ → Area → Area → ℕ
   pedestmoving i ui zero = \ _ _ → 0
   pedestmoving i ui (suc t) = pedestmoving' (pedestmoving i ui t)
                                             (pedestposition i ui t)
                                             (DTS.nthState PeliconDecidableLadder i t)

   pedestposition : Input → UsersInput → ℕ → Area → ℕ
   pedestposition i ui zero = λ _ → 0
   pedestposition i ui (suc t) = pedestposition' (ui t) (pedestmoving i ui (suc t))
                                                        (pedestposition i ui t)

SimState : Set
SimState = (Area → ℕ) × (Area → ℕ) × (Area → Area → ℕ) × (Area → Area → ℕ)

nc : SimState → Area → ℕ
nc = proj₁

np : SimState → Area → ℕ
np = proj₁ ∘ proj₂

mc : SimState → Area → Area → ℕ
mc = proj₁ ∘ proj₂ ∘ proj₂

mp : SimState → Area → Area → ℕ
mp = proj₂ ∘ proj₂ ∘ proj₂

nextState : SimState × Ladder.State pelicon-ladder
          → Ladder.Input pelicon-ladder
          → (ℕ × ℕ × ℕ × ℕ)
          → SimState × Ladder.State pelicon-ladder
nextState s i ui
  = (carposition' ui (carmoving' (mc $ proj₁ s) (nc $ proj₁ s) (proj₂ s)) (nc $ proj₁ s)
     , pedestposition' ui (pedestmoving' (mp $ proj₁ s) (np $ proj₁ s) (proj₂ s)) (np $ proj₁ s)
     , carmoving' (mc $ proj₁ s) (nc $ proj₁ s) (proj₂ s)
     , pedestmoving' (mp $ proj₁ s) (np $ proj₁ s) (proj₂ s))
    , (DTS.transitionFunction PeliconDecidableLadder (proj₂ s) i)

initSimState : SimState
initSimState = ( (const 0) , (const 0) , (const $ const 0) , (const $ const 0) )
```

```
nthState : Input → UsersInput → ℕ → SimState × Ladder.State pelicon-ladder
nthState i ui zero = initSimState , DTS.initialState PeliconDecidableLadder
nthState i ui (suc t) = nextState (nthState i ui t) (i t) (ui t)

archState : Input → UsersInput → ℕ → State
archState i ui t = record {
                    numcars = nc $ proj₁ $ nthState i ui t;
                    numpeds = np $ proj₁ $ nthState i ui t;
                    movingcars = mc $ proj₁ $ nthState i ui t;
                    movingpeds = mp $ proj₁ $ nthState i ui t;
                    traffic = archCarAspect $ proj₂ $ nthState i ui t;
                    pedest = archPedAspect $ proj₂ $ nthState i ui t }

lem-carmoving : ∀ i ui t → State.traffic (archState i ui t) ≡ red
              → State.movingcars (archState i ui (suc t)) T1 MUX ≡ 0
                × State.movingcars (archState i ui (suc t)) T2 MUX ≡ 0
lem-carmoving i ui t tred with sigmap (mkenv (proj₁ (proj₂ (nthState i ui t))) TLIGHT)
lem-carmoving i ui t tred | red = refl , refl
lem-carmoving i ui t () | green

lem-carmoving2 : ∀ i ui t
               → State.movingcars (archState i ui (suc t)) T1 MUX
                   ≤ State.numcars (archState i ui t) T1
lem-carmoving2 i ui t with sigmap (mkenv (proj₁ (proj₂ (nthState i ui t))) TLIGHT)
lem-carmoving2 i ui t | red = z≤n
lem-carmoving2 i ui t | green = ≤-id _

lem-carmoving3 : ∀ i ui t
               → State.movingcars (archState i ui (suc t)) T2 MUX
                   ≤ State.numcars (archState i ui t) T2
lem-carmoving3 i ui t with sigmap (mkenv (proj₁ (proj₂ (nthState i ui t))) TLIGHT)
lem-carmoving3 i ui t | red = z≤n
lem-carmoving3 i ui t | green = ≤-id _

lem-pedestmoving : ∀ i ui t → State.pedest (archState i ui t) ≡ red
              → State.movingpeds (archState i ui (suc t)) P1 MUX ≡ 0
                × State.movingpeds (archState i ui (suc t)) P2 MUX ≡ 0
lem-pedestmoving i ui t tred with sigmap (mkenv (proj₁ (proj₂ (nthState i ui t))) PLIGHT)
lem-pedestmoving i ui t tred | red = refl , refl
lem-pedestmoving i ui t () | green

lem-pedestmoving2 : ∀ i ui t
                  → State.movingpeds (archState i ui (suc t)) P1 MUX
                      ≤ State.numpeds (archState i ui t) P1
lem-pedestmoving2 i ui t with sigmap (mkenv (proj₁ (proj₂ (nthState i ui t))) PLIGHT)
lem-pedestmoving2 i ui t | red = z≤n
lem-pedestmoving2 i ui t | green = ≤-id _

lem-pedestmoving3 : ∀ i ui t
                  → State.movingpeds (archState i ui (suc t)) P2 MUX
                      ≤ State.numpeds (archState i ui t) P2
lem-pedestmoving3 i ui t with sigmap (mkenv (proj₁ (proj₂ (nthState i ui t))) PLIGHT)
lem-pedestmoving3 i ui t | red = z≤n
lem-pedestmoving3 i ui t | green = ≤-id _

nthReachable' : ∀ i ui n → Reachable pelicon-fsm (proj₂ $ nthState i ui n)
nthReachable' i ui zero = initial (DTS.initialState PeliconDecidableLadder)
                                  (DTS.initialCorrectness PeliconDecidableLadder)
nthReachable' i ui (suc n) = next (proj₂ $ nthState i ui n) (nthReachable' i ui n)
                                  (i n)
                                  (proj₂ $ nthState i ui (suc n))
                                  (DTS.transitionCorrectness PeliconDecidableLadder
                                                             (proj₂ $ nthState i ui n) (i n))

nthLadderStateCorrect' : ∀ φ
                       → LadderCorrectness pelicon-ladder φ
                       → ∀ n i ui
                       → ⟦safety⟧π φ (proj₂ $ nthState i ui n) (i n) (proj₂ $ nthState i ui (suc n))
nthLadderStateCorrect' φ lc n i ui = lc (proj₂ $ nthState i ui n)
                                        (nthReachable' i ui n)
                                        (i n)
                                        (proj₂ $ nthState i ui (suc n))
                                        (DTS.transitionCorrectness PeliconDecidableLadder
                                                                   (proj₂ $ nthState i ui n) (i n))

private
  φ = (~ (¥ TLIGHT) || ~ (¥ PLIGHT))

open import Boolean.TPTP

peliconproof : LadderCorrectness pelicon-ladder φ
peliconproof = inductiveProof pelicon-ladder φ

peliconProof : ∀ t (i : Input) (ui : UsersInput) → ⟦ mkenv (proj₁ $ proj₂ $ nthState i ui t) ⊧ φ ⟧pl
peliconProof t i ui = lem-mkenv-++-pl' φ (proj₁ $ proj₂ $ nthState i ui t)
                        (proj₁ (proj₁ (i t)) ++ proj₁ (proj₂ (nthState i ui (suc t))))
                        (subst (λ k → T (bound k φ)) (sym (proj₂ (proj₂ (nthState i ui t))))  )
```

```
                (nthLadderStateCorrect' φ peliconproof t i ui)

safety : ∀ i ui t
       → State.movingcars (archState i ui t) T1 MUX ≡ 0
           × State.movingcars (archState i ui t) T2 MUX ≡ 0
         ⊎ State.movingpeds (archState i ui t) P1 MUX ≡ 0
             × State.movingpeds (archState i ui t) P2 MUX ≡ 0
safety i ui zero = inj₁ (refl , refl)
safety i ui (suc t) with peliconProof t i ui
safety i ui (suc t) | inj₁ x rewrite lem-sigmap' _ x = inj₁ (refl , refl)
safety i ui (suc t) | inj₂ y rewrite lem-sigmap' _ y = inj₂ (refl , refl)

pelicon-controller : Input → UsersInput → Controller
pelicon-controller i ui = record {
                      nthState = archState i ui;
                      taxm1 = lem-carmoving i ui;
                      taxm4 = lem-carmoving2 i ui;
                      taxm5 = lem-carmoving3 i ui;
                      paxm1 = lem-pedestmoving i ui;
                      paxm4 = lem-pedestmoving2 i ui;
                      paxm5 = lem-pedestmoving3 i ui;
                      safetyp = safety i ui;
                      taxm2 = λ t → refl; taxm3 = λ t → refl; taxm6 = λ t → refl;
                      paxm2 = λ t → refl; paxm3 = λ t → refl; paxm6 = λ t → refl }
```

```
module Pelicon.Safe where

open import Pelicon.PeliconModel
open import Pelicon.State
open import Data.Sum
open import Data.Product
open import Relation.Binary.PropositionalEquality

pelicon-safe : (i : Input) → (ui : UsersInput) → Safe (pelicon-controller i ui)
pelicon-safe i ui = record { initnousers = (refl , refl) , (refl , refl) , refl , refl }
```

```
module Pelicon.Simulator where

open import IO.Console
open import Foreign.Haskell

open import Data.String as String hiding (_==_) renaming (_++_ to _+++_)
open import Data.Char
open import Data.Bool
open import Data.List as List
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Nat hiding (_<_)
open import Data.Maybe

open import Relation.Binary.PropositionalEquality

open import Coinduction
open import coparse as P hiding (choice)

open import PropIso

open import TransitionSystem.Decidable

open import Pelicon.Ladder

showBool : Bool → String
showBool true  = "True"
showBool false = "False"

showState : Σ[ l ∶ List Bool ] (List.length l ≡ 4) → String
showState (crossing ∷ req ∷ tlight ∷ plight ∷ [] , proj₂) =
  "Crossing: "  +++ showBool crossing      +++ "\n" +++
  "Requested: " +++ showBool req           +++ "\n" +++
  "Car Green: " +++ showBool tlight        +++ "\n" +++
  "Pedestrian Green: " +++ showBool plight +++ "\n\n"
showState _ = "IMPOSSIBLE ERROR"

parseRequested : Parser Bool
parseRequested s = maybe′ just (P.choice' (parseDataSp "no" false) (parseDataSp "n" false) s)
                               (P.choice' (parseDataSp "yes" true) (parseDataSp "r" true) s)

readRequested : IOConsole Bool
readRequested = do (putStrLn (toCostring "Request Crossing [yes/no] ")) \ _ →
              ♯ do getLine \ l →
              ♯ choice (parseRequested l) (return ∘ proj₁) readRequested

readInput : IOConsole (Σ[ l ∶ List Bool ] (List.length l ≡ 1))
readInput = readRequested >>= λ b →
          ♯ return ( b ∷ [] , refl)

readExit : IOConsole Choice
readExit = do (putStrLn (toCostring "do you want to exit? [yes/no]")) \ _ →
         ♯ do getLine \ l1 →
         ♯ choice (parseChoice l1) (return ∘ proj₁) readExit

mainloop : Σ[ l ∶ List Bool ] (List.length l ≡ 4) → IOConsole Unit
mainloop s = (do (putStrLn (toCostring (showState s))) \ _ → ♯ return unit) >>= const
             (♯ readInput) >>= λ inp →
             ♯ mainloop
               (DecidableTransitionSystem.transitionFunction PeliconDecidableLadder s (inp , _))

program : IOConsole Unit
program = (do (putStrLn (toCostring "Pelicon Crossing Simulator 0.01\nentering main loop...")) \ _ →
        ♯ mainloop (DecidableTransitionSystem.initialState PeliconDecidableLadder))

main : PrimIO Unit
main = translateIOConsole program
```

```
module Pelicon.SimulatorFull where

open import IO.Console
open import Foreign.Haskell

open import Data.String as String hiding (_==_) renaming (_++_ to _+++_)
open import Data.Char
open import Data.Bool
open import Data.List as List
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.Nat hiding (_<_)
open import Data.Maybe

open import Relation.Binary.PropositionalEquality

open import Coinduction
open import coparse as P hiding (choice)

open import PropIso

open import TransitionSystem.Decidable

open import Pelicon.Ladder
open import Pelicon.State
open import Pelicon.PeliconModel

showBool : Bool → String
showBool true  = "True"
showBool false = "False"

showState : Σ[ l ∶ List Bool ] (List.length l ≡ 4) → String
showState (crossing ∷ req ∷ tlight ∷ plight ∷ [] , proj₂) =
  "Crossing: " +++ showBool crossing +++ "\n" +++
  "Requested: " +++ showBool req +++ "\n" +++
  "Car Green: " +++ showBool tlight +++ "\n" +++
  "Pedestrian Green: " +++ showBool plight +++ "\n"
showState _ = "IMPOSSIBLE ERROR"

primitive
  primShowNat : ℕ → String

-- append the moving users
showSimState : SimState → String
showSimState (nc , np , mc , mp) =
 "Area\tPeople\tCars\n" +++
 "P1\t" +++ primShowNat (np P1) +++ "\t" +++ primShowNat (nc P1) +++ "\n" +++
 "P2\t" +++ primShowNat (np P2) +++ "\t" +++ primShowNat (nc P2) +++ "\n" +++
 "T1\t" +++ primShowNat (np T1) +++ "\t" +++ primShowNat (nc T1) +++ "\n" +++
 "T2\t" +++ primShowNat (np T2) +++ "\t" +++ primShowNat (nc T2) +++ "\n" +++
 "MUX\t" +++ primShowNat (np MUX) +++ "\t" +++ primShowNat (nc MUX) +++ "\n\n" +++
 "P1->MUX: " +++ primShowNat (mp P1 MUX) +++ "   P2->MUX: " +++ primShowNat (mp P2 MUX) +++ "\n" +++
 "MUX->P1: " +++ primShowNat (mp MUX P1) +++ "   MUX->P2: " +++ primShowNat (mp MUX P2) +++ "\n" +++
 "T1->MUX: " +++ primShowNat (mc T1 MUX) +++ "   T2->MUX: " +++ primShowNat (mc T2 MUX) +++ "\n" +++
 "MUX->T1: " +++ primShowNat (mc MUX T1) +++ "   MUX->T2: " +++ primShowNat (mc MUX T2) +++ "\n\n"

parseRequested : Parser Bool
parseRequested s = maybe′ just (P.choice' (parseDataSp "no" false) (parseDataSp "n" false) s)
                               (P.choice' (parseDataSp "yes" true) (parseDataSp "r" true) s)

readRequested : IOConsole Bool
readRequested = do (putStrLn (toCostring "Request Crossing [yes/no] ")) \ _ →
              # do getLine \ l →
              # choice (parseRequested l) (return ∘ proj₁) readRequested

readApproachT1 : IOConsole ℕ
readApproachT1 = do (putStrLn (toCostring "Enter no. cars embarking T1")) \ _ →
                  # do getLine \ l1 →
                  # choice (parseSpaced parseNat l1) (return ∘ proj₁) readApproachT1

readApproachT2 : IOConsole ℕ
readApproachT2 = do (putStrLn (toCostring "Enter no. cars embarking T2")) \ _ →
                  # do getLine \ l1 →
                  # choice (parseSpaced parseNat l1) (return ∘ proj₁) readApproachT2

readApproachP1 : IOConsole ℕ
readApproachP1 = do (putStrLn (toCostring "Enter no. people embarking P1")) \ _ →
                  # do getLine \ l1 →
                  # choice (parseSpaced parseNat l1) (return ∘ proj₁) readApproachP1

readApproachP2 : IOConsole ℕ
readApproachP2 = do (putStrLn (toCostring "Enter no. people embarking P2")) \ _ →
                  # do getLine \ l1 →
                  # choice (parseSpaced parseNat l1) (return ∘ proj₁) readApproachP2

readInput : IOConsole (Σ[ l ∶ List Bool ] (List.length l ≡ 1))
readInput = readRequested >>= λ b →
```

```
          ♯ return ( b ∷ [] , refl)

readExit : IOConsole Choice
readExit = do (putStrLn (toCostring "do you want to exit? [yes/no]")) \ _ →
         ♯ do getLine \ l1 →
         ♯ choice (parseChoice l1) (return ∘ proj₁) readExit

mainloop : SimState × (Σ[ l ∶ List Bool ] (List.length l ≡ 4)) → IOConsole Unit
mainloop s = (do (putStrLn (toCostring (showState (proj₂ s)))) \ _ →
             ♯ do (putStrLn (toCostring (showSimState (proj₁ s)))) \ _ →
             ♯ return unit) >>= const
             (♯ readInput) >>= λ inp →
             ♯ (readApproachT1 >>= λ t1 →
             ♯ (readApproachT2 >>= λ t2 →
             ♯ (readApproachP1 >>= λ p1 →
             ♯ (readApproachP2 >>= λ p2 →
             ♯ mainloop (nextState s (inp , _) (t1 , t2 , p1 , p2))))))

program : IOConsole Unit
program = (do (putStrLn (toCostring "Pelicon Crossing Simulator \nentering main loop...\n")) \ _ →
        ♯ mainloop (initSimState , DecidableTransitionSystem.initialState PeliconDecidableLadder))

main : PrimIO Unit
main = translateIOConsole program
```

```
module Gwili.Layout where

open import RDM.RailYard
open import RDM.fixedtrains

open import Data.Fin using (Fin;suc;zero;#_)
open import Data.Fin.Arithmetic
open import Data.Fin.Decidable
open import Data.Nat as Nat hiding (_<_)
open import Data.Bool
open import Data.List
open import Data.Product as Prod
open import Data.Sum as Sum
open import Data.List.Inhabitence

open import Relation.Decidable
open import Relation.Binary.PropositionalEquality hiding ([_])

open import TransitionSystem.Decidable

open import PropIso

gwiliConnections : Fin 9 → List (Fin 9)
gwiliConnections zero                                                = [ # 1 ]
gwiliConnections (suc zero)                                          = # 0 :: # 2 :: []
gwiliConnections (suc (suc zero))                                    = # 1 :: # 3 :: # 6 :: []
gwiliConnections (suc (suc (suc zero)))                              = # 2 :: # 4 :: []
gwiliConnections (suc (suc (suc (suc zero))))                        = # 3 :: # 7 :: # 5 :: []
gwiliConnections (suc (suc (suc (suc (suc zero)))))                  = [ # 4 ]
gwiliConnections (suc (suc (suc (suc (suc (suc zero))))))            = # 2 :: # 7 :: []
gwiliConnections (suc (suc (suc (suc (suc (suc (suc zero)))))))      = # 4 :: # 6 :: # 8 :: []
gwiliConnections (suc (suc (suc (suc (suc (suc (suc (suc zero))))))))= [ # 7 ]
gwiliConnections (suc (suc (suc (suc (suc (suc (suc (suc (suc ()))))))))))

data Route (l : List ℕ) : Set where
  <_> : (n : ℕ) → {q : T (n isin ℕ l)} → Route l

unRoute : ∀ {l} → Route l → ℕ
unRoute < n > = n

decroute : ∀ {l} → Decidable (_≡_ {_} {Route l})
decroute {l} (<_> m {mp}) (<_> n {np}) with Nat._≟_ m n
...| yes x rewrite x = yes (cong (λ p → < n > {p}) (uip-isinℕ n l mp np))
...| no x = no (x ∘ cong unRoute)

gwiliRoutes = 3 :: 2 :: 6 :: 4 :: 19 :: 17 :: 16 :: 7 :: 5 :: 18 :: 20 :: []

elimRoute : {A : Set} → ∀ l → (f : (n : ℕ) → T (n isin ℕ l) → A) → Route l → A
elimRoute l f (<_> p {q}) = f p q

record ElimGwiliRoute (A : Route gwiliRoutes → Set) : Set where
  field
    f3  : A < 3 >
    f2  : A < 2 >
    f6  : A < 6 >
    f4  : A < 4 >
    f19 : A < 19 >
    f17 : A < 17 >
    f16 : A < 16 >
    f7  : A < 7 >
    f5  : A < 5 >
    f18 : A < 18 >
    f20 : A < 20 >

record ElimGwiliRouteInspect (A : Route gwiliRoutes → Set) (rt : Route gwiliRoutes) : Set where
  field
    f3  : (eq : rt ≡ < 3 >) → A < 3 >
    f2  : (eq : rt ≡ < 2 >) → A < 2 >
    f6  : (eq : rt ≡ < 6 >) → A < 6 >
    f4  : (eq : rt ≡ < 4 >) → A < 4 >
    f19 : (eq : rt ≡ < 19 >) → A < 19 >
    f17 : (eq : rt ≡ < 17 >) → A < 17 >
    f16 : (eq : rt ≡ < 16 >) → A < 16 >
    f7  : (eq : rt ≡ < 7 >) → A < 7 >
    f5  : (eq : rt ≡ < 5 >) → A < 5 >
    f18 : (eq : rt ≡ < 18 >) → A < 18 >
    f20 : (eq : rt ≡ < 20 >) → A < 20 >

elimGwiliRoute : (A : Route gwiliRoutes → Set) → ElimGwiliRoute A → (r : Route gwiliRoutes) → A r
elimGwiliRoute A egr (<_> 0  {()})
elimGwiliRoute A egr (<_> 1  {()})
elimGwiliRoute A egr (<_> 2  {p}) = ElimGwiliRoute.f2 egr
elimGwiliRoute A egr (<_> 3  {p}) = ElimGwiliRoute.f3 egr
elimGwiliRoute A egr (<_> 4  {p}) = ElimGwiliRoute.f4 egr
elimGwiliRoute A egr (<_> 5  {p}) = ElimGwiliRoute.f5 egr
elimGwiliRoute A egr (<_> 6  {p}) = ElimGwiliRoute.f6 egr
```

```
elimGwiliRoute A egr (<_> 7  {p}) = ElimGwiliRoute.f7 egr
elimGwiliRoute A egr (<_> 8  {()})
elimGwiliRoute A egr (<_> 9  {()})
elimGwiliRoute A egr (<_> 10 {()})
elimGwiliRoute A egr (<_> 11 {()})
elimGwiliRoute A egr (<_> 12 {()})
elimGwiliRoute A egr (<_> 13 {()})
elimGwiliRoute A egr (<_> 14 {()})
elimGwiliRoute A egr (<_> 15 {()})
elimGwiliRoute A egr (<_> 16 {p}) = ElimGwiliRoute.f16 egr
elimGwiliRoute A egr (<_> 17 {p}) = ElimGwiliRoute.f17 egr
elimGwiliRoute A egr (<_> 18 {p}) = ElimGwiliRoute.f18 egr
elimGwiliRoute A egr (<_> 19 {p}) = ElimGwiliRoute.f19 egr
elimGwiliRoute A egr (<_> 20 {p}) = ElimGwiliRoute.f20 egr
elimGwiliRoute A egr (<_> ((suc (suc (suc (suc (suc (suc (suc
                           (suc (suc (suc (suc (suc (suc (suc (suc
                           (suc (suc (suc (suc (suc n))))))))))))))))))))) {()})

elimGwiliRouteInspect : (A : Route gwiliRoutes → Set) → (r : Route gwiliRoutes)
                      → ElimGwiliRouteInspect A r → A r
elimGwiliRouteInspect A (<_> 0 {()})  egr
elimGwiliRouteInspect A (<_> 1 {()})  egr
elimGwiliRouteInspect A (<_> 2 {p})   egr = ElimGwiliRouteInspect.f2 egr refl
elimGwiliRouteInspect A (<_> 3 {p})   egr = ElimGwiliRouteInspect.f3 egr refl
elimGwiliRouteInspect A (<_> 4 {p})   egr = ElimGwiliRouteInspect.f4 egr refl
elimGwiliRouteInspect A (<_> 5 {p})   egr = ElimGwiliRouteInspect.f5 egr refl
elimGwiliRouteInspect A (<_> 6 {p})   egr = ElimGwiliRouteInspect.f6 egr refl
elimGwiliRouteInspect A (<_> 7 {p})   egr = ElimGwiliRouteInspect.f7 egr refl
elimGwiliRouteInspect A (<_> 8 {()})  egr
elimGwiliRouteInspect A (<_> 9 {()})  egr
elimGwiliRouteInspect A (<_> 10 {()}) egr
elimGwiliRouteInspect A (<_> 11 {()}) egr
elimGwiliRouteInspect A (<_> 12 {()}) egr
elimGwiliRouteInspect A (<_> 13 {()}) egr
elimGwiliRouteInspect A (<_> 14 {()}) egr
elimGwiliRouteInspect A (<_> 15 {()}) egr
elimGwiliRouteInspect A (<_> 16 {p})  egr = ElimGwiliRouteInspect.f16 egr refl
elimGwiliRouteInspect A (<_> 17 {p})  egr = ElimGwiliRouteInspect.f17 egr refl
elimGwiliRouteInspect A (<_> 18 {p})  egr = ElimGwiliRouteInspect.f18 egr refl
elimGwiliRouteInspect A (<_> 19 {p})  egr = ElimGwiliRouteInspect.f19 egr refl
elimGwiliRouteInspect A (<_> 20 {p})  egr = ElimGwiliRouteInspect.f20 egr refl
elimGwiliRouteInspect A (<_> ((suc (suc (suc (suc (suc (suc (suc (suc
                             (suc (suc (suc (suc (suc (suc (suc (suc
                             (suc (suc (suc (suc (suc n))))))))))))))))))))) {()}) egr

gwiliPhysicalLayout : PhysicalLayout
gwiliPhysicalLayout = record {
                        Segment = Fin 9;
                        Signal = Route gwiliRoutes;
                        connections = gwiliConnections;
                        signalLocation = elimGwiliRoute
                                           (const
                                            (SignalLocation (Fin 9)
                                                            (λ a b → b isin gwiliConnections a)))
                                           (record {
                                              f3  = sigloc (# 0) (# 1) (inj₁ refl);
                                              f2  = sigloc (# 1) (# 2) (inj₂ (inj₁ refl));
                                              f6  = sigloc (# 1) (# 2) (inj₂ (inj₁ refl));
                                              f4  = sigloc (# 1) (# 2) (inj₂ (inj₁ refl));
                                              f19 = sigloc (# 3) (# 2) (inj₁ refl);
                                              f17 = sigloc (# 6) (# 2) (inj₁ refl);
                                              f16 = sigloc (# 1) (# 0) (inj₁ refl);
                                              f7  = sigloc (# 6) (# 7) (inj₂ (inj₁ refl));
                                              f5  = sigloc (# 3) (# 4) (inj₂ (inj₁ refl));
                                              f18 = sigloc (# 5) (# 4) (inj₁ refl);
                                              f20 = sigloc (# 5) (# 4) (inj₁ refl) }) }

gwiliRoute : Route gwiliRoutes → ControlTableEntry gwiliPhysicalLayout
gwiliRoute =
  elimGwiliRoute (const (ControlTableEntry gwiliPhysicalLayout))
    (record {
       f3  = record {
               start         = < 3 >;
               segments      = [ # 1 ];
               normalpoints  = [];
               reversepoints = [];
               facing        = []
             };
       f2  = record {
               start         = < 2 >;
               segments      = # 2 :: [ # 3 ];
               normalpoints  = [ # 2 ];
               reversepoints = [];
               facing        = [ # 2 ]
             };
       f6  = record {
               start         = < 6 >;
```

```
                segments      = # 2 :: [ # 6 ];
                normalpoints  = [];
                reversepoints = [ # 2 ];
                facing        = [ # 2 ]
              };
        f4  = record {
                start         = < 4 >;
                segments      = # 2 :: [ # 3 ];
                normalpoints  = [ # 2 ];
                reversepoints = [];
                facing        = [ # 2 ]
              };
        f19 = record {
                start         = < 19 >;
                segments      = # 2 :: [ # 1 ];
                normalpoints  = [ # 2 ];
                reversepoints = [];
                facing        = []
              };
        f17 = record {
                start         = < 17 >;
                segments      = # 2 :: [ # 1 ];
                normalpoints  = [];
                reversepoints = [ # 2 ];
                facing        = []
              };
        f16 = record {
                start         = < 16 >;
                segments      = [ # 0 ];
                normalpoints  = [];
                reversepoints = [];
                facing        = []
              };
        f7  = record {
                start         = < 7 >;
                segments      = # 7 :: # 4 :: [ # 5 ];
                normalpoints  = [];
                reversepoints = # 4 :: [ # 7 ];
                facing        = [ # 7 ]
              };
        f5  = record {
                start         = < 5 >;
                segments      = # 4 :: [ # 5 ];
                normalpoints  = [ # 4 ];
                reversepoints = [];
                facing        = []
              };
        f18 = record {
                start         = < 18 >;
                segments      = # 4 :: # 7 :: [ # 6 ];
                normalpoints  = [];
                reversepoints = # 4 :: [ # 7 ];
                facing        = [ # 4 ]
              };
        f20 = record {
                start         = < 20 >;
                segments      = # 4 :: [ # 3 ];
                normalpoints  = [ # 4 ];
                reversepoints = [];
                facing        = [ # 4 ]
              }})

gwiliPoints : List (Fin 9)
gwiliPoints = # 2 :: # 4 :: [ # 7 ]

-- basic connections, does not allow for trains to be reversed
gwiliRouteConnections : Route gwiliRoutes → List (Route gwiliRoutes)
gwiliRouteConnections = elimGwiliRoute (const $ List $ Route gwiliRoutes)
                          (record {
                             f3  = < 2 > :: < 6 > :: [ < 4 > ];
                             f2  = [ < 5 > ];
                             f6  = [ < 7 > ];
                             f4  = [ < 5 > ];
                             f19 = [ < 16 > ];
                             f17 = [ < 16 > ];
                             f16 = [];
                             f7  = [];
                             f5  = [];
                             f18 = [ < 17 > ];
                             f20 = [ < 19 > ] })

GwiliRouteConnections : Route gwiliRoutes → Route gwiliRoutes → Set
GwiliRouteConnections rt₁ rt₂ = rt₂ isin gwiliRouteConnections rt₁

subst' : ∀ {a b x} → (eq : a ≡ b) → x isin gwiliRouteConnections a → x isin gwiliRouteConnections b
subst' {a} {b} {x} = (subst (λ k → x isin gwiliRouteConnections k))
```

```
 private
   ψ' : (rt₁ rt₃ : Route gwiliRoutes) → Set
   ψ' rt₁ rt₃ =  ∃ (λ ts → (ts isin (ControlTableEntry.segments (gwiliRoute rt₁)))
                          × (ts isin (ControlTableEntry.segments (gwiliRoute rt₃))))

   ψ : (rt₁ rt₂ rt₃ : Route gwiliRoutes) → Set
   ψ rt₁ rt₂ rt₃ = rt₂ isin gwiliRouteConnections rt₁
                 → rt₂ isin gwiliRouteConnections rt₃
                 → ψ' rt₁ rt₃

   ψ'' : (rt : Route gwiliRoutes) → Set
   ψ'' rt₄ = {rt₁ rt₂ : Route gwiliRoutes}
                 → (rt₃ : Route gwiliRoutes)
                 → (rt₁ ≡ rt₄)
                 → rt₂ isin gwiliRouteConnections rt₁
                 → rt₂ isin gwiliRouteConnections rt₃
                 → ψ' rt₄ rt₃

 abstract
   GwiliWellFormedRoutes2 : ψ'' < 2 >
   GwiliWellFormedRoutes2 rt₃ refl (inj₁ refl) rt₂▸rt₃
     = elimGwiliRouteInspect (\ rt₃ →  ψ' (< 2 >) rt₃) rt₃
       (record {
          f3 = λ eq → [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]' (subst' eq rt₂▸rt₃);
          f2 = const (, (inj₁ refl , inj₁ refl));
          f6 = const (, (inj₁ refl , inj₁ refl));
          f4 = const (, (inj₁ refl , inj₁ refl));
          f19 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f17 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
   GwiliWellFormedRoutes2 rt₃ refl (inj₂ y)     rt₂▸rt₃ = ⊥-elim y

   GwiliWellFormedRoutes3 : ψ'' < 3 >
   GwiliWellFormedRoutes3 rt₃ refl (inj₁ refl) rt₂▸rt₃
     = elimGwiliRouteInspect (\ rt₃ → ψ' (< 3 >) rt₃) rt₃
       (record {
         f3 = const (, (inj₁ refl , inj₁ refl));
         f2 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f6 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f4 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f19 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f17 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
   GwiliWellFormedRoutes3 rt₃ refl (inj₂ (inj₁ refl)) rt₂▸rt₃
     = elimGwiliRouteInspect (\ rt₃ → ψ' (< 3 >) rt₃) rt₃
       (record {
         f3 = const (, (inj₁ refl , inj₁ refl));
         f2 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f6 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f4 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f19 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f17 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
   GwiliWellFormedRoutes3 rt₃ refl (inj₂ (inj₂ (inj₁ refl))) rt₂▸rt₃
     = elimGwiliRouteInspect (\ rt₃ → ψ' (< 3 >) rt₃) rt₃
       (record {
         f3 = const (, (inj₁ refl , inj₁ refl));
         f2 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f6 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f4 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f19 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f17 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
   GwiliWellFormedRoutes3 rt₃ refl (inj₂ (inj₂ (inj₂ y))) rt₂▸rt₃ = ⊥-elim y

   GwiliWellFormedRoutes4 : ψ'' < 4 >
   GwiliWellFormedRoutes4 rt₃ refl (inj₁ refl) rt₂▸rt₃
     = elimGwiliRouteInspect (\ rt₃ → ψ' (< 4 >) rt₃) rt₃
       (record {
          f3 = λ eq → [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]' (subst' eq rt₂▸rt₃);
          f2 = const (, (inj₁ refl , inj₁ refl));
```

```
          f6 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f4 = const (, (inj₁ refl , inj₁ refl));
          f19 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f17 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
  GwiliWellFormedRoutes4 rt₃ refl (inj₂ y)    rt₂▸rt₃ = ⊥-elim y

GwiliWellFormedRoutes5 : ψ'' < 5 >
GwiliWellFormedRoutes5 rt₃ refl eq rt₂▸rt₃ = ⊥-elim eq

abstract
  GwiliWellFormedRoutes6 : ψ'' < 6 >
  GwiliWellFormedRoutes6 rt₃ refl (inj₁ refl) rt₂▸rt₃
    = elimGwiliRouteInspect (\ rt₃ → ψ' (< 6 >) rt₃) rt₃
      (record {
          f3 = λ eq → [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]' (subst' eq rt₂▸rt₃);
          f2 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f6 = const (, (inj₁ refl , inj₁ refl));
          f4 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f19 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f17 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
  GwiliWellFormedRoutes6 rt₃ refl (inj₂ y) rt₂▸rt₃ = ⊥-elim y

GwiliWellFormedRoutes7 : ψ'' < 7 >
GwiliWellFormedRoutes7 rt₃ refl eq rt₂▸rt₃ = ⊥-elim eq

GwiliWellFormedRoutes16 : ψ'' < 16 >
GwiliWellFormedRoutes16 rt₃ refl eq rt₂▸rt₃ = ⊥-elim eq

abstract
  GwiliWellFormedRoutes17 : ψ'' < 17 >
  GwiliWellFormedRoutes17 rt₃ refl (inj₁ refl) rt₂▸rt₃
    = elimGwiliRouteInspect (\ rt₃ → ψ' (< 17 >) rt₃) rt₃
      (record {
          f3 = λ eq → [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]' (subst' eq rt₂▸rt₃);
          f2 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f6 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f4 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f19 = const (, (inj₁ refl , inj₁ refl));
          f17 = const (, (inj₁ refl , inj₁ refl));
          f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
  GwiliWellFormedRoutes17 rt₃ refl (inj₂ y) rt₂▸rt₃ = ⊥-elim y

  GwiliWellFormedRoutes18 : ψ'' < 18 >
  GwiliWellFormedRoutes18 rt₃ refl (inj₁ refl) rt₂▸rt₃
    = elimGwiliRouteInspect (\ rt₃ → ψ' (< 18 >) rt₃) rt₃
      (record {
          f3 = λ eq → [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]' (subst' eq rt₂▸rt₃);
          f2 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f6 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f4 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f19 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f17 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f7 =  λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f5 =  λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f18 = const (, (inj₁ refl , inj₁ refl));
          f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
  GwiliWellFormedRoutes18 rt₃ refl (inj₂ y) rt₂▸rt₃ = ⊥-elim y

  GwiliWellFormedRoutes19 : ψ'' < 19 >
  GwiliWellFormedRoutes19 rt₃ refl (inj₁ refl) rt₂▸rt₃
    = elimGwiliRouteInspect (\ rt₃ → ψ' (< 19 >) rt₃) rt₃
      (record {
          f3 = λ eq → [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]' (subst' eq rt₂▸rt₃);
          f2 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f6 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f4 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
          f19 = const (, (inj₁ refl , inj₁ refl));
          f17 = const (, (inj₁ refl , inj₁ refl));
          f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
          f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
```

```
          f20 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃) })
  GwiliWellFormedRoutes19 rt₃ refl (inj₂ y) rt₂▸rt₃ = ⊥-elim y

  GwiliWellFormedRoutes20 : ψ'' < 20 >
  GwiliWellFormedRoutes20 rt₃ refl (inj₁ refl) rt₂▸rt₃
    = elimGwiliRouteInspect (\ rt₃ → ψ' (< 20 >) rt₃) rt₃
      (record {
         f3 = λ eq → [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]' (subst' eq rt₂▸rt₃);
         f2 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f6 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f4 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f19 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f17 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f16 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f7 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f5 = λ eq → ⊥-elim (subst' eq rt₂▸rt₃);
         f18 = λ eq → [ (λ ()) , (λ ()) ]' (subst' eq rt₂▸rt₃);
         f20 = const (, (inj₁ refl , inj₁ refl)) })
  GwiliWellFormedRoutes20 rt₃ refl (inj₂ y) rt₂▸rt₃ = ⊥-elim y

GwiliWellFormedRoutes : (rt₁ rt₂ rt₃ : Route gwiliRoutes) → ψ rt₁ rt₂ rt₃
GwiliWellFormedRoutes rt₁ rt₂ rt₃ rt₂▸rt₁ rt₂▸rt₃ =
  elimGwiliRouteInspect (\ rt₁ → ψ' rt₁ rt₃) rt₁
    (record {
       f3 = λ eq → GwiliWellFormedRoutes3 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f2 = λ eq → GwiliWellFormedRoutes2 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f6 = λ eq → GwiliWellFormedRoutes6 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f4 = λ eq → GwiliWellFormedRoutes4 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f19 = λ eq → GwiliWellFormedRoutes19 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f17 = λ eq → GwiliWellFormedRoutes17 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f16 = λ eq → GwiliWellFormedRoutes16 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f7 = λ eq → GwiliWellFormedRoutes7 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f5 = λ eq → GwiliWellFormedRoutes5 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f18 = λ eq → GwiliWellFormedRoutes18 rt₃ eq rt₂▸rt₁ rt₂▸rt₃;
       f20 = λ eq → GwiliWellFormedRoutes20 rt₃ eq rt₂▸rt₁ rt₂▸rt₃ }
    )

GwiliNonEmptyRoutes : ∀ rt → Σ[ ts ∶ Fin 9 ] (ts isin (ControlTableEntry.segments $ gwiliRoute rt))
GwiliNonEmptyRoutes = elimGwiliRoute _ (record {
                                          f3 = , inj₁ refl;
                                          f2 = , inj₁ refl;
                                          f6 = , inj₁ refl;
                                          f4 = , inj₁ refl;
                                          f19 = , inj₁ refl;
                                          f17 = , inj₁ refl;
                                          f16 = , inj₁ refl;
                                          f7 = , inj₁ refl;
                                          f5 = , inj₁ refl;
                                          f18 = , inj₁ refl;
                                          f20 = , inj₁ refl })

private
  φ : (rt₁ rt₂ : Route gwiliRoutes) → Set
  φ rt₁ rt₂ = rt₂ isin gwiliRouteConnections rt₁
              → (SignalLocation.facing (PhysicalLayout.signalLocation gwiliPhysicalLayout
                                                                      (ControlTableEntry.start $ gwiliRoute rt₂)))
                ≡ last (ControlTableEntry.segments $ gwiliRoute rt₁) (GwiliNonEmptyRoutes rt₁)
              × (SignalLocation.trailing (PhysicalLayout.signalLocation gwiliPhysicalLayout
                                                                      (ControlTableEntry.start $ gwiliRoute rt₂)))
                ≡ head (ControlTableEntry.segments $ gwiliRoute rt₂) (GwiliNonEmptyRoutes rt₂)

  φ' : (rt₁ : Route gwiliRoutes) → Set
  φ' rt₁ = (rt₂ : Route gwiliRoutes) → φ rt₁ rt₂

GwiliRoutesConnected3 : φ' < 3 >
GwiliRoutesConnected3 .(< 2 >) (inj₁ refl) = refl , refl
GwiliRoutesConnected3 .(< 6 >) (inj₂ (inj₁ refl)) = refl , refl
GwiliRoutesConnected3 .(< 4 >) (inj₂ (inj₂ (inj₁ refl))) = refl , refl
GwiliRoutesConnected3 rt₂ (inj₂ (inj₂ (inj₂ ())))

GwiliRoutesConnected2 : φ' < 2 >
GwiliRoutesConnected2 .(< 5 >) (inj₁ refl) = refl , refl
GwiliRoutesConnected2 rt₂ (inj₂ ())

GwiliRoutesConnected6 : φ' < 6 >
GwiliRoutesConnected6 .(< 7 >) (inj₁ refl) = refl , refl
GwiliRoutesConnected6 rt₂ (inj₂ ())

GwiliRoutesConnected4 : φ' < 4 >
GwiliRoutesConnected4 .(< 5 >) (inj₁ refl) = refl , refl
GwiliRoutesConnected4 rt₂ (inj₂ ())

GwiliRoutesConnected19 : φ' < 19 >
GwiliRoutesConnected19 .(< 16 >) (inj₁ refl) = refl , refl
GwiliRoutesConnected19 rt₂ (inj₂ ())

GwiliRoutesConnected17 : φ' < 17 >
```

```
GwiliRoutesConnected17 .(< 16 >) (inj₁ refl) = refl , refl
GwiliRoutesConnected17 rt₂ (inj₂ ())

GwiliRoutesConnected18 : φ' < 18 >
GwiliRoutesConnected18 .(< 17 >) (inj₁ refl) = refl , refl
GwiliRoutesConnected18 rt₂ (inj₂ ())

GwiliRoutesConnected20 : φ' < 20 >
GwiliRoutesConnected20 .(< 19 >) (inj₁ refl) = refl , refl
GwiliRoutesConnected20 rt₂ (inj₂ ())

GwiliRoutesConnected : ∀ rt₁ rt₂ → φ rt₁ rt₂
GwiliRoutesConnected rt₁ rt₂ = elimGwiliRoute (\ rt₁ → φ rt₁ rt₂)
                                  (record {
                                     f3 =  GwiliRoutesConnected3 rt₂;
                                     f2 =  GwiliRoutesConnected2 rt₂;
                                     f6 =  GwiliRoutesConnected6 rt₂;
                                     f4 =  GwiliRoutesConnected4 rt₂;
                                     f19 = GwiliRoutesConnected19 rt₂;
                                     f17 = GwiliRoutesConnected17 rt₂;
                                     f16 = ⊥-elim;
                                     f7 =  ⊥-elim;
                                     f5 =  ⊥-elim;
                                     f18 = GwiliRoutesConnected18 rt₂;
                                     f20 = GwiliRoutesConnected20 rt₂ }) rt₁

gwiliControlTable : ControlTable gwiliPhysicalLayout
gwiliControlTable = record {
                      Route = Route gwiliRoutes;
                      DecidableRoute = decroute;
                      entries = gwiliRoute;
                      connections = gwiliRouteConnections;
                      NonEmptyRoutes = GwiliNonEmptyRoutes;
                      WellFormed = GwiliWellFormedRoutes;
                      RoutesConnected = GwiliRoutesConnected }

GwiliLayout : Layout
GwiliLayout = toLayout gwiliPhysicalLayout gwiliControlTable (Fin 2 , findecidableeq)
```

```
module Ladder.LockingTable where

open import Data.Nat
open import Data.List hiding (reverse)
open import Data.Product as Prod
open import Data.Bool

open import Relation.Binary.PropositionalEquality hiding ([_])

open import Boolean.Formula

open import Ladder.Core

open import PropIso

data Leaver : Set where
  _R  _N _B : ℕ → Leaver

record TableEntry : Set where
  constructor releasedby_locks_bothways_
  field
    releasedby : List ℕ
    locks      : List ℕ
    bothways   : List ℕ

Table : Set
Table = List TableEntry

♭leaver : Leaver → PL-Formula
♭leaver (y R) = ¥ y
♭leaver (y N) = ~ (¥ y)
♭leaver (y B) = ¥true

_elem_ : ℕ → List ℕ → Bool
_elem_ l []       = false
_elem_ l (x ∷ xs) = x == l ∨ l elem xs

requirenormal : List ℕ → PL-Formula
requirenormal []       = ¥true
requirenormal (x ∷ xs) = ♭leaver (x N) && requirenormal xs

requirereverse : List ℕ → PL-Formula
requirereverse []       = ¥true
requirereverse (x ∷ xs) = ♭leaver (x R) && requirereverse xs

-- a state is wellformed iff it does not violate constraints in the locking table
WellFormedState : Table → ℕ → PL-Formula
WellFormedState [] n = ¥true
WellFormedState (releasedby r locks l bothways b ∷ xs) n
  = (♭leaver (n N) || requirereverse r && requirenormal l) && WellFormedState xs (suc n)

{- any leavers that locks this one (snd arg) bothways are normal -}
bothwayslocks-aux : List ℕ → ℕ → ℕ → PL-Formula
bothwayslocks-aux bw n l = if l elem bw then ♭leaver (n N) else ¥true

bothwayslocks : Table → ℕ → ℕ → PL-Formula
bothwayslocks []       n l = ¥true
bothwayslocks (x ∷ xs) n l = bothwayslocks-aux (TableEntry.bothways x) n l &&
                             bothwayslocks xs (suc n) l

{- any backlock on this leaver (3rd arg) are normal -}
backlocks-aux : List ℕ → ℕ → ℕ → PL-Formula
backlocks-aux relby n l = if l elem relby then ♭leaver (n N) else ¥true

backlocks : Table → ℕ → ℕ → PL-Formula
backlocks []       n l = ¥true
backlocks (x ∷ xs) n l = backlocks-aux (TableEntry.releasedby x) n l &&
                         backlocks xs (suc n) l

{- any leaver that locks this leaver (3rd arg) are normal -}
notlocked-aux : List ℕ → ℕ → ℕ → PL-Formula
notlocked-aux loks n l = if l elem loks then ♭leaver (n N) else ¥true

notlocked : Table → ℕ → ℕ → PL-Formula
notlocked []       n l = ¥true
notlocked (x ∷ xs) n l = notlocked-aux (TableEntry.locks x) n l &&
```

```
                          notlocked xs (suc n) l

{-
a leaver is freetomove (r) when:
  all leavers that lock this leaver (N) are N
  all leavers that relsease this leaver to go R are N
  all leavers that lock this B are N
  all leavers that this leaver locks (N) are N, without
    conditionals this is eqiv to first condition
-}

freetomoveentryʳ : TableEntry → PL-Formula
freetomoveentryʳ te = requirereverse (TableEntry.releasedby te) &&
                      requirenormal (TableEntry.locks te)

freetomoveʳ-aux : Table → ℕ → PL-Formula
freetomoveʳ-aux []       l       = ¥false
freetomoveʳ-aux (x :: xs) zero    = freetomoveentryʳ x
freetomoveʳ-aux (x :: xs) (suc n) = freetomoveʳ-aux xs n

freetomoveʳ : Table → ℕ → PL-Formula
freetomoveʳ t l = freetomoveʳ-aux t l && notlocked t 0 l && bothwayslocks t 0 l

{-
a leaver is freetomove (n) when
  all leavers that lock this leaver B are N
  all leavers that this leaver releases are N
-}
freetomoveⁿ : Table → ℕ → PL-Formula
freetomoveⁿ t l = backlocks t 0 l && bothwayslocks t 0 l

-- the input for the ladder is: i0 i1 ... in
-- where n is the number of leavers.
-- i0 idicates to move normal/reverse true == reverse, false == normal
-- i1 .. in idicates which leaver to move, need input invariance.

-- the state of the ladder is given by s0 ... s(n-1),
-- normal / reverse for each leaver.

-- i₀ : Table → ℕ
-- i₀ t = length t

-- i : Table → ℕ → ℕ -- input variable
-- i t zero = suc (i₀ t)
-- i t (suc n) = suc (i t n)

mkRungs-aux : ℕ → Table → Table → ℕ → List (ℕ × PL-Formula)
mkRungs-aux l _ [] n = []
-- true == reverse, false == normal
mkRungs-aux l t (x :: xs) n = (n , (¥ (n + l) && ((~ (¥ n)) && (freetomoveʳ t n) ||
                                    (¥ n) && (~ (freetomoveⁿ t n))))
                              || ~ (¥ (n + l)) && ¥ n) :: mkRungs-aux l t xs (suc n)

mkRungs : Table → List (ℕ × PL-Formula)
mkRungs t = mkRungs-aux (length t) t t 0

mkInitial : ℕ → List (ℕ × Bool)
mkInitial zero    = []
mkInitial (suc n) = (n , false) :: mkInitial n

mkDisj : ℕ → ℕ → PL-Formula
mkDisj l zero    = ¥false
mkDisj l (suc n) = ¥ (n + l) || mkDisj l n

mkInvariant-aux : ℕ → ℕ → PL-Formula
mkInvariant-aux l zero    = ¥true
mkInvariant-aux l (suc n) = (¥ (n + l) => ~ (mkDisj l n)) && mkInvariant-aux l n

-- at most one leaver can go high at a time, need this over inputs.
mkInvariant : ℕ → PL-Formula
mkInvariant n = (mkInvariant-aux n n)

mkLadder : Table → Ladder
mkLadder t = ladder (length t) (length t) (mkRungs t) (mkInitial (length t))
                    (mkInvariant-aux 0 (length t))
```

```
module Gwili.Ladder where

open import Ladder.LockingTable
open import Ladder.Core
open import Ladder.Decidable

open import Data.List
open import Data.Product as Prod
open import Data.Nat hiding (_<_)
open import Data.Bool

open import PropIso

open import Boolean.Formula using (PL-Formula;bound)

open import TransitionSystem.Decidable

-- private
--   points = 0
--   lock = 1
--   signal = 2
--   direction = 3
--   pointsrq = 4
--   lockrq = 5
--   signalrq = 6
--   points' = 7
--   lock' = 8
--   signal' = 9

-- pointstable : Table
-- pointstable = releasedby []         locks [] bothways []
--             ∷ releasedby []         locks [] bothways [ points ]
--             ∷ releasedby [ lock ]   locks [] bothways []
--             ∷ []


{-
need mapping from leavers into the railyard equipment they control

0 ↦ ⊥ -- added by me to preserve numbering
1 ↦ ⊥ -- space
2 ↦ sig
3 ↦ sig
4 ↦ sig
5 ↦ sig
6 ↦ sig
7 ↦ sig
8 ↦ ? -- spare
9 ↦ ⊥ -- space
10 ↦ ? -- space
11 ↦ ? -- space
12 ↦ fpl
13 ↦ points
14 ↦ points
15 ↦ fpl
16 ↦ sig
17 ↦ sig
18 ↦ sig
19 ↦ sig
20 ↦ sig
21 ↦ ? -- space
-}

--original version
gwilitable : Table
gwilitable = releasedby []            locks []                                              bothways []
{- 1  -}   ∷ releasedby []            locks []                                              bothways []
{- 2  -}   ∷ releasedby [ 12 ]        locks (3 ∷ 4 ∷ 5 ∷ 7 ∷ 8  ∷ 13 ∷ 16 ∷ 19 ∷ [ 20 ]) bothways []
{- 3  -}   ∷ releasedby [ 12 ]        locks (2 ∷ 4 ∷ 5 ∷ 7 ∷ 8  ∷ 13 ∷ 16 ∷ 19 ∷ [ 20 ]) bothways []
{- 4  -}   ∷ releasedby [ 12 ]        locks (2 ∷ 5 ∷ 7 ∷ 8 ∷ 13 ∷ 14 ∷ 16 ∷ 19 ∷ [ 20 ]) bothways []
{- 5  -}   ∷ releasedby []            locks (2 ∷ 3 ∷ 4 ∷ 6 ∷ 11 ∷ 13 ∷ 14 ∷ 19 ∷ [ 20 ]) bothways []
{- 6  -}   ∷ releasedby (12 ∷ [ 13 ]) locks (5 ∷ 7 ∷ 8 ∷ 16 ∷ 17 ∷ [ 18 ])                 bothways []
 -- fails fpl for shunt signal, so added 15 to be pulled
{- 7  -}   ∷ releasedby (14 ∷ [ 15 ]) locks (2 ∷ 3 ∷ 4 ∷ 6 ∷ 8 ∷ 11 ∷ 13 ∷ 17 ∷ [ 18 ])  bothways []
{- 8  -}   ∷ releasedby []            locks (10 ∷ 14 ∷ 18 ∷ 20 ∷ [ 21 ])                    bothways []
{- 9  -}   ∷ releasedby []            locks []                                              bothways []
{- 10 -}   ∷ releasedby []            locks (8 ∷ [ 11 ])                                    bothways []
{- 11 -}   ∷ releasedby []            locks (5 ∷ 7 ∷ 8 ∷ 10 ∷ [ 21 ])                       bothways []
{- 12 -}   ∷ releasedby []            locks []                                        bothways [ 13 ]
{- 13 -}   ∷ releasedby []            locks (2 ∷ 3 ∷ 4 ∷ 19 ∷ [ 20 ])                    bothways []
{- 14 -}   ∷ releasedby []            locks (4 ∷ 5 ∷ [ 20 ])                             bothways []
{- 15 -}   ∷ releasedby []            locks []                                        bothways [ 14 ]
{- 16 -}   ∷ releasedby []            locks (2 ∷ 3 ∷ 4 ∷ 6 ∷ [ 13 ])                     bothways []
{- 17 -}   ∷ releasedby [ 13 ]        locks (6 ∷ 7 ∷ [ 14 ])                             bothways []
{- 18 -}   ∷ releasedby (14 ∷ [ 15 ]) locks (6 ∷ 7 ∷ 8 ∷ 13 ∷ [ 19 ])                     bothways []
{- 19 -}   ∷ releasedby []            locks (2 ∷ 3 ∷ 4 ∷ 13 ∷ 14 ∷ [ 18 ])                bothways []
{- 20 -}   ∷ releasedby [ 15 ]        locks (2 ∷ 3 ∷ 4 ∷ 8 ∷ 13 ∷ [ 14 ])                 bothways []
```

```
 {- 21 -}    :: releasedby []            locks (8 :: [ 11 ])                bothways []
             :: []

gwiliLadder = mkLadder gwilitable
GwiliLadder = mkTransitionSystem gwiliLadder

GwiliLadderWellFormed : LadderWellFormed gwiliLadder
GwiliLadderWellFormed = record { initialmap   = mkfinmap 22 (Ladder.initialstate gwiliLadder) _;
                                 rungsmap     = mkfinmap 22 (Ladder.rungs gwiliLadder) _;
                                 rungvarbound = mkbound 44 (Ladder.rungs gwiliLadder) _ }

GwiliDecidableLadder : DecidableTransitionSystem GwiliLadder
GwiliDecidableLadder = mkDecTransitionSystem (, GwiliLadderWellFormed)

gstatesWellFormed : LadderCorrectness gwiliLadder (WellFormedState gwilitable 0)
gstatesWellFormed = inductiveProof gwiliLadder (WellFormedState gwilitable 0)
```

```
module Gwili.State where

open import RDM.fixedtrains
open import RDM.RailYard

open import Data.Nat hiding (_<_)
open import Data.Fin hiding (_<_;pred)
open import Data.Product as Prod
open import Data.List
open import Data.List.Inhabitence
open import Data.Bool
open import Data.Sum as Sum

open import Relation.Binary.PropositionalEquality hiding ([_])
open import Relation.Decidable

open import Gwili.Layout
open import Gwili.Abstract
open import Gwili.Ladder

open import Ladder.Core

open import Boolean.Formula renaming (¥ to var_)

open import PropIso

data TrainDirection : Set where
  moving : (Route gwiliRoutes) → TrainDirection
  stationary : TrainDirection

Inputs : Set
Inputs = ℕ → Ladder.Input gwiliLadder

TrainInputs : Set
TrainInputs = ℕ → Fin 2 → TrainDirection

aspmap : Bool → Aspect
aspmap true  = Proceed
aspmap false = Stop

lockmap : Bool → Locking
lockmap true  = Locked
lockmap false = Unlocked

archSignalState : Ladder.State gwiliLadder → Route gwiliRoutes → Aspect
archSignalState s rt = aspmap $ mkenv (proj₁ s) (unRoute rt)

archSegmentLock : Ladder.State gwiliLadder → Fin 9 → Locking
archSegmentLock s (suc (suc zero)) = lockmap $ mkenv (proj₁ s) 12
archSegmentLock s (suc (suc (suc (suc zero)))) = lockmap $ mkenv (proj₁ s) 15
archSegmentLock s (suc (suc (suc (suc (suc (suc (suc zero))))))) = lockmap $ mkenv (proj₁ s) 15
archSegmentLock s _ = Locked

initTrainPosition : Fin 2 → Route gwiliRoutes
initTrainPosition zero = < 3 >
initTrainPosition _    = < 16 >

decasp : Decidable (_≡_ {_} {Aspect})
decasp Proceed Proceed = yes refl
decasp Stop Proceed    = no (λ ())
decasp Proceed Stop    = no (λ ())
decasp Stop Stop       = yes refl

mutual
  trainPosition-aux : (inp : Inputs)
                    → TrainInputs
                    → (t : ℕ)
                    → Fin 2
                    → Route gwiliRoutes
                    → List (Route gwiliRoutes)
                    → Route gwiliRoutes
  trainPosition-aux inp tinp t tr rt [] = trainPosition inp tinp t tr
  trainPosition-aux inp tinp t tr rt (rt' :: rts)
    = elim-Dec (_≡_ {A = Route gwiliRoutes}) (decroute rt rt')
               (elim-Dec (_≡_ {A = Aspect}) (decasp (archSignalState (nthState inp t) rt) Proceed)
                         (λ _ _ → rt) (λ _ _ → trainPosition inp tinp t tr))
               (λ _ → trainPosition-aux inp tinp t tr rt rts)

  trainPosition : Inputs → TrainInputs → ℕ → Fin 2 → Route gwiliRoutes
  trainPosition inp tinp 0       tr = initTrainPosition tr
  trainPosition inp tinp (suc t) tr with tinp t tr
  trainPosition inp tinp (suc t) tr | moving rt
    = trainPosition-aux inp tinp t tr rt (gwiliRouteConnections (trainPosition inp tinp t tr))
  trainPosition inp tinp (suc t) tr | stationary = trainPosition inp tinp t tr
```

```
toArchState : ℕ → Inputs → TrainInputs → LayoutState GwiliLayout
toArchState n inputs tinp = record {
                              trainRoute    = trainPosition inputs tinp n;
                              signalAspect = archSignalState (nthState inputs n);
                              locked       = archSegmentLock (nthState inputs n)
                            }
```

```
module Gwili.ControlTableCorrect where

open import RDM.RailYard

open import Gwili.Layout
open import Gwili.Ladder

open import Boolean.Formula
open import Boolean.SatSolver

open import Ladder.Core

open import Data.Nat
open import Data.Bool
open import Data.Unit
open import Data.List
open import Data.Fin hiding (_+_)

open import Function

open ControlTableEntry

rtSet : Route gwiliRoutes → PL-Formula
rtSet < n > = ¥ n

segNormal : Fin 9 → PL-Formula
segNormal (suc (suc zero)) = ~ (¥ 13)
segNormal (suc (suc (suc (suc zero)))) = ~ (¥ 14)
segNormal (suc (suc (suc (suc (suc (suc (suc zero))))))) = ~ (¥ 14)
segNormal _ = ¥true

segReverse : Fin 9 → PL-Formula
segReverse = ~ ∘ segNormal

segLocked : Fin 9 → PL-Formula
segLocked (suc (suc zero)) = ¥ 12
segLocked (suc (suc (suc (suc zero)))) = ¥ 15
segLocked (suc (suc (suc (suc (suc (suc (suc zero))))))) = ¥ 15
segLocked _ = ¥true

ControlTableCorrectness : Route gwiliRoutes → PL-Formula
ControlTableCorrectness rt
  = rtSet rt => (foldr (λ s x → segNormal s && x) ¥true
                       (normalpoints (ControlTable.entries gwiliControlTable rt)) &&
                 foldr (λ s x → segReverse s && x) ¥true
                       (reversepoints (ControlTable.entries gwiliControlTable rt)) &&
                 foldr (λ s x → segLocked s && x) ¥true
                       (facing (ControlTable.entries gwiliControlTable rt)))

private
  ψ : ∀ rt → PL-Formula
  ψ rt = (ControlTableCorrectness rt)

  allψ = ψ < 3 >   && ψ < 2 >   && ψ < 6 >   && ψ < 4 >   &&
         ψ < 19 >  && ψ < 17 >  && ψ < 16 >  && ψ < 7 >   &&
         ψ < 5 >   && ψ < 18 >  && ψ < 20 >

ControlTableCorrect : LadderCorrectness gwiliLadder allψ
ControlTableCorrect = inductiveProof gwiliLadder allψ
```

```
module Gwili.Abstract where

open import Data.Nat
open import Data.Bool
open import Data.Product
open import Data.Sum
open import Data.List

open import Relation.Binary.PropositionalEquality

open import Boolean.Formula

open import Ladder.Core
open import Ladder.Decidable

open import TransitionSystem.Decidable

open import Gwili.Ladder

abstract
  rung' : List (ℕ × PL-Formula)
  rung' = Ladder.rungs gwiliLadder

  initialcfg' : List (ℕ × Bool)
  initialcfg' = Ladder.initialstate gwiliLadder

gwiliLadder' : Ladder
gwiliLadder' = ladder (Ladder.statevars gwiliLadder) (Ladder.inputvars gwiliLadder)
                      rung' initialcfg' (Ladder.inp-correct gwiliLadder)

abstract
  gwili-eq : gwiliLadder' ≡ gwiliLadder
  gwili-eq = refl

  nthState : (ℕ → Ladder.Input gwiliLadder') → ℕ → Ladder.State gwiliLadder'
  nthState = DecidableTransitionSystem.nthState GwiliDecidableLadder

  nthState-eq : nthState ≡ DecidableTransitionSystem.nthState GwiliDecidableLadder
  nthState-eq = refl

  nthState0-eq : ∀ inp
               → nthState inp 0 ≡ constructInitialState (gwiliLadder , GwiliLadderWellFormed)
  nthState0-eq inp = mkDecTrans-init-eq (gwiliLadder , GwiliLadderWellFormed)
```

```
module Gwili.Abstract-level2 where

open import Data.Nat

open import Boolean.Formula

open import Ladder.Core
open import Ladder.Decidable

open import TransitionSystem.Decidable

open import Gwili.Ladder
open import Gwili.Abstract

open import Relation.Binary.PropositionalEquality

abstract
  nthLadderStateCorrect' : (φ : PL-Formula)
                         → LadderCorrectness gwiliLadder' φ
                         → (n : ℕ)
                         → (inputs : ℕ → Ladder.Input gwiliLadder')
                         → ⟦safety⟧π φ (nthState inputs n) (inputs n) (nthState inputs (suc n))
  nthLadderStateCorrect' = subst (\ l → (φ : PL-Formula)
                                      → LadderCorrectness l φ
                                      → (n : ℕ)
                                      → (i : ℕ → Ladder.Input gwiliLadder')
                                      → ⟦safety⟧π φ (nthState i n) (i n) (nthState i (suc n)))
                                 (sym gwili-eq)
                                 (subst (\ f → (φ : PL-Formula)
                                             → LadderCorrectness gwiliLadder φ
                                             → (n : ℕ)
                                             → (i : ℕ → Ladder.Input gwiliLadder)
                                             → ⟦safety⟧π φ (f i n) (i n) (f i (suc n)))
                                        (sym nthState-eq)
                                        (nthLadderStateCorrect {gwiliLadder} GwiliDecidableLadder))
```

```
module Gwili.Ladder.OpposingRoutes where

open import Boolean.Formula

open import Gwili.Ladder
open import Gwili.Abstract
open import Gwili.Abstract-level2

open import Ladder.Core
open import Ladder.Decidable

open import Data.Nat
open import Data.Product
open import Data.Sum
open import Data.List
open import Data.Unit
open import Data.Bool

open import Relation.Binary.PropositionalEquality

open import Ladder.LockingTable

open import Function

φopp : ℕ → ℕ → PL-Formula
φopp n m = ~ (¥ n) || ~ (¥ m)

private
  φ : PL-Formula
  φ = φopp 3  19   && φopp 3  17   &&
      φopp 2  6    && φopp 2  19   &&
      φopp 2  17   && φopp 2  20   &&
      φopp 6  4    && φopp 6  17   &&
      φopp 6  18   && φopp 6  19   &&
      φopp 4  17   && φopp 4  19   &&
      φopp 4  20   && φopp 17 19   &&
      φopp 5  7    && φopp 5  18   &&
      φopp 5  20   && φopp 7  18   &&
      φopp 7  20   && φopp 18 20   &&
      φopp 2  4

  abstract
    OpposingProof : LadderCorrectness gwiliLadder' φ
    OpposingProof = subst (\ l → LadderCorrectness l φ)
                          (sym gwili-eq)
                          (ladderapply {(WellFormedState gwilitable 0)} {φ}
                                gwiliLadder
                                (inductiveProof gwiliLadder ((WellFormedState gwilitable 0) => φ))
                                gstatesWellFormed)

  #OpposingProof : ∀ t inp → ⟦safety⟧π φ (nthState inp t) (inp t) (nthState inp (suc t))
  #OpposingProof t inp = nthLadderStateCorrect' φ OpposingProof t inp

  ##OpposingProof : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ φ ⟧pl
  ##OpposingProof t inp = lem-mkenv-++-pl' φ
                   (proj₁ (nthState inp t))
                   (proj₁ (proj₁ (inp t)) ++
                    proj₁ (nthState inp (suc t)))
                  (subst (λ n → T (bound n φ))
                      (sym (proj₂ (nthState inp t))) tt) (#OpposingProof t inp)

Opp-3-19 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 3 19) ⟧pl
Opp-3-19 t inp = proj₁ $ ##OpposingProof t inp

Opp-3-17 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 3 17) ⟧pl
Opp-3-17 t inp = proj₁ $ proj₂ $ ##OpposingProof t inp

Opp-2-6 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 2 6) ⟧pl
Opp-2-6 t inp = proj₁ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-2-19 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 2 19) ⟧pl
Opp-2-19 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-2-17 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 2 17) ⟧pl
Opp-2-17 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-2-20 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 2 20) ⟧pl
Opp-2-20 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-6-4 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 6 4) ⟧pl
Opp-6-4 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-6-17 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 6 17) ⟧pl
Opp-6-17 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 ##OpposingProof t inp
```

```
Opp-6-18 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 6 18) ⟧pl
Opp-6-18 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ ##OpposingProof t inp

Opp-6-19 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 6 19) ⟧pl
Opp-6-19 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-4-17 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 4 17) ⟧pl
Opp-4-17 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-4-19 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 4 19) ⟧pl
Opp-4-19 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-4-20 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 4 20) ⟧pl
Opp-4-20 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-17-19 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 17 19) ⟧pl
Opp-17-19 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                  proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-5-7 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 5 7) ⟧pl
Opp-5-7 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-5-18 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 5 18) ⟧pl
Opp-5-18 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 ##OpposingProof t inp

Opp-5-20 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 5 20) ⟧pl
Opp-5-20 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ ##OpposingProof t inp

Opp-7-18 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 7 18) ⟧pl
Opp-7-18 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-7-20 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 7 20) ⟧pl
Opp-7-20 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                 proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-18-20 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 18 20) ⟧pl
Opp-18-20 t inp = proj₁ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                  proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                  proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-2-4 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 2 4) ⟧pl
Opp-2-4 t inp = proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $ proj₂ $
                proj₂ $ proj₂ $ proj₂ $ proj₂ $ ##OpposingProof t inp

Opp-19-3 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 19 3) ⟧pl
Opp-19-3 t inp = [ inj₂ , inj₁ ]′ (Opp-3-19 t inp)

Opp-17-3 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 17 3) ⟧pl
Opp-17-3 t inp = [ inj₂ , inj₁ ]′ (Opp-3-17 t inp)

Opp-6-2 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 6 2) ⟧pl
Opp-6-2 t inp = [ inj₂ , inj₁ ]′ (Opp-2-6 t inp)

Opp-19-2 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 19 2) ⟧pl
Opp-19-2 t inp = [ inj₂ , inj₁ ]′ (Opp-2-19 t inp)

Opp-17-2 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 17 2) ⟧pl
Opp-17-2 t inp = [ inj₂ , inj₁ ]′ (Opp-2-17 t inp)

Opp-20-2 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 20 2) ⟧pl
Opp-20-2 t inp = [ inj₂ , inj₁ ]′ (Opp-2-20 t inp)

Opp-4-6 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 4 6) ⟧pl
Opp-4-6 t inp = [ inj₂ , inj₁ ]′ (Opp-6-4 t inp)

Opp-17-6 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 17 6) ⟧pl
Opp-17-6 t inp = [ inj₂ , inj₁ ]′ (Opp-6-17 t inp)

Opp-18-6 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 18 6) ⟧pl
Opp-18-6 t inp = [ inj₂ , inj₁ ]′ (Opp-6-18 t inp)
```

```
Opp-19-6 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 19 6) ⟧pl
Opp-19-6 t inp = [ inj₂ , inj₁ ]′ (Opp-6-19 t inp)

Opp-17-4 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 17 4) ⟧pl
Opp-17-4 t inp = [ inj₂ , inj₁ ]′ (Opp-4-17 t inp)

Opp-19-4 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 19 4) ⟧pl
Opp-19-4 t inp = [ inj₂ , inj₁ ]′ (Opp-4-19 t inp)

Opp-20-4 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 20 4) ⟧pl
Opp-20-4 t inp = [ inj₂ , inj₁ ]′ (Opp-4-20 t inp)

Opp-19-17 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 19 17) ⟧pl
Opp-19-17 t inp = [ inj₂ , inj₁ ]′ (Opp-17-19 t inp)

Opp-7-5 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 7 5) ⟧pl
Opp-7-5 t inp = [ inj₂ , inj₁ ]′ (Opp-5-7 t inp)

Opp-18-5 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 18 5) ⟧pl
Opp-18-5 t inp = [ inj₂ , inj₁ ]′ (Opp-5-18 t inp)

Opp-20-5 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 20 5) ⟧pl
Opp-20-5 t inp = [ inj₂ , inj₁ ]′ (Opp-5-20 t inp)

Opp-18-7 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 18 7) ⟧pl
Opp-18-7 t inp = [ inj₂ , inj₁ ]′ (Opp-7-18 t inp)

Opp-20-7 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 20 7) ⟧pl
Opp-20-7 t inp = [ inj₂ , inj₁ ]′ (Opp-7-20 t inp)

Opp-20-18 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 20 18) ⟧pl
Opp-20-18 t inp = [ inj₂ , inj₁ ]′ (Opp-18-20 t inp)

Opp-4-2 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ (φopp 4 2) ⟧pl
Opp-4-2 t inp = [ inj₂ , inj₁ ]′ (Opp-2-4 t inp)
```

```
module Gwili.OpposingSignals where

open import RDM.fixedtrains
open import RDM.RailYard

open import Gwili.Layout
open import Gwili.State
open import Gwili.Abstract
open import Gwili.Ladder
open import Gwili.Ladder.OpposingRoutes

open import Boolean.Formula renaming (¥ to var)

open import Data.Nat
open import Data.Sum
open import Data.Product
open import Data.List
open import Data.List.Inhabitence
open import Data.Fin using (Fin;zero;suc;#_)

open import Relation.Binary.PropositionalEquality

open import PropIso

-- Assemble opposing signal proof
#OppSig : (inputs : Inputs)
        → ∀ t
        → (rt₁ rt₂ : Route gwiliRoutes)
        → ⟦ mkenv (proj₁ (nthState inputs t)) ⊧ (φopp (unRoute rt₁) (unRoute rt₂)) ⟧pl
        → archSignalState (nthState inputs t) rt₁ ≡ Stop
          ⊎ archSignalState (nthState inputs t) rt₂ ≡ Stop
#OppSig inputs t < rt₁ > < rt₂ > (inj₁ x) rewrite ¬Tb x = inj₁ refl
#OppSig inputs t < rt₁ > < rt₂ > (inj₂ y) rewrite ¬Tb y = inj₂ refl

private
  ψ' : (inputs : Inputs) (t : ℕ) (rt₁ : Route gwiliRoutes) → Set
  ψ' inputs t rt₁
    = ∀ rt₂ ts
       → rt₁ ≢ rt₂
       → ts isin (ControlTableEntry.segments $ gwiliRoute rt₁)
       → ts isin (ControlTableEntry.segments $ gwiliRoute rt₂)
       → archSignalState (nthState inputs t) rt₁ ≡ Stop
         ⊎ archSignalState (nthState inputs t) rt₂ ≡ Stop

  ψ'' : (rt₁ : Route gwiliRoutes) → Set
  ψ'' rt₁ = (inputs : Inputs) (t : ℕ) → ψ' inputs t rt₁

  ψ : Set
  ψ = (inputs : Inputs) (t : ℕ) (rt₁ : Route gwiliRoutes) → ψ' inputs t rt₁

  ψ''' : (inputs : Inputs) (t : ℕ) (ts : Fin 9) → (rt₁ rt₂ : Route gwiliRoutes) → Set
  ψ''' inputs t ts rt₁ rt₂ = ts isin (ControlTableEntry.segments $ gwiliRoute rt₂)
   → archSignalState (nthState inputs t) rt₁ ≡ Stop ⊎ archSignalState (nthState inputs t) rt₂ ≡ Stop

OppSig3 : ψ''  < 3 >
OppSig3 inputs t rt₂ ts eq (inj₂ ())
OppSig3 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 1) (< 3 >)) rt₂
      (record {
         f3  = ⊥-elim ∘ eq ∘ sym;
         f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f19 = const $ const $ #OppSig inputs t (< 3 >) (< 19 >) (Opp-3-19 t inputs);
         f17 = const $ const $ #OppSig inputs t (< 3 >) (< 17 >) (Opp-3-17 t inputs);
         f16 = const [ (λ ()) , (λ ()) ]';
         f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f20 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]'})

OppSig2 : ψ'' < 2 >
OppSig2 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 2) (< 2 >)) rt₂
      (record {
         f3  = const [ (λ ()) , (λ ()) ]';
         f2  = ⊥-elim ∘ eq ∘ sym;
         f6  = const $ const $ #OppSig inputs t (< 2 >) (< 6 >) (Opp-2-6 t inputs);
         f4  = const $ const $ #OppSig inputs t (< 2 >) (< 4 >) (Opp-2-4 t inputs);
         f19 = const $ const $ #OppSig inputs t (< 2 >) (< 19 >) (Opp-2-19 t inputs);
         f17 = const $ const $ #OppSig inputs t (< 2 >) (< 17 >) (Opp-2-17 t inputs);
         f16 = const [ (λ ()) , (λ ()) ]';
         f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f20 = const $ const $ #OppSig inputs t (< 2 >) (< 20 >) (Opp-2-20 t inputs)})
OppSig2 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
```

```
  = elimGwiliRouteInspect (ψ''' inputs t (# 3) (< 2 >)) rt₂
      (record {
         f3  = const [ (λ ()) , (λ ()) ]';
         f2  = ⊥-elim ∘ eq ∘ sym;
         f6  = const $ const $ #OppSig inputs t (< 2 >) (< 6 >) (Opp-2-6 t inputs);
         f4  = const $ const $ #OppSig inputs t (< 2 >) (< 4 >) (Opp-2-4 t inputs);
         f19 = const $ const $ #OppSig inputs t (< 2 >) (< 19 >) (Opp-2-19 t inputs);
         f17 = const $ const $ #OppSig inputs t (< 2 >) (< 17 >) (Opp-2-17 t inputs);
         f16 = const [ (λ ()) , (λ ()) ]';
         f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f20 = const $ const $ #OppSig inputs t (< 2 >) (< 20 >) (Opp-2-20 t inputs)})
OppSig2 inputs t rt₂ ts eq (inj₂ (inj₂ ()))

OppSig6 : ψ'' < 6 >
OppSig6 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 2) (< 6 >)) rt₂
      (record {
         f3  = const [ (λ ()) , (λ ()) ]';
         f2  = const $ const $ #OppSig inputs t (< 6 >) (< 2 >) (Opp-6-2 t inputs);
         f6  = ⊥-elim ∘ eq ∘ sym;
         f4  = const $ const $ #OppSig inputs t (< 6 >) (< 4 >) (Opp-6-4 t inputs);
         f19 = const $ const $ #OppSig inputs t (< 6 >) (< 19 >) (Opp-6-19 t inputs);
         f17 = const $ const $ #OppSig inputs t (< 6 >) (< 17 >) (Opp-6-17 t inputs);
         f16 = const [ (λ ()) , (λ ()) ]';
         f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f18 = const $ const $ #OppSig inputs t (< 6 >) (< 18 >) (Opp-6-18 t inputs);
         f20 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]'})
OppSig6 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
  = elimGwiliRouteInspect (ψ''' inputs t (# 6) (< 6 >)) rt₂
      (record {
         f3  = const [ (λ ()) , (λ ()) ]';
         f2  = const $ const $ #OppSig inputs t (< 6 >) (< 2 >) (Opp-6-2 t inputs);
         f6  = ⊥-elim ∘ eq ∘ sym;
         f4  = const $ const $ #OppSig inputs t (< 6 >) (< 4 >) (Opp-6-4 t inputs);
         f19 = const $ const $ #OppSig inputs t (< 6 >) (< 19 >) (Opp-6-19 t inputs);
         f17 = const $ const $ #OppSig inputs t (< 6 >) (< 17 >) (Opp-6-17 t inputs);
         f16 = const [ (λ ()) , (λ ()) ]';
         f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f18 = const $ const $ #OppSig inputs t (< 6 >) (< 18 >) (Opp-6-18 t inputs);
         f20 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]'})
OppSig6 inputs t rt₂ ts eq (inj₂ (inj₂ ()))

OppSig4 : ψ'' < 4 >
OppSig4 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 2) (< 4 >)) rt₂
      (record {
         f3  = const [ (λ ()) , (λ ()) ]';
         f2  = const $ const $ #OppSig inputs t (< 4 >) (< 2 >) (Opp-4-2 t inputs);
         f6  = const $ const $ #OppSig inputs t (< 4 >) (< 6 >) (Opp-4-6 t inputs);
         f4  = ⊥-elim ∘ eq ∘ sym;
         f19 = const $ const $ #OppSig inputs t (< 4 >) (< 19 >) (Opp-4-19 t inputs);
         f17 = const $ const $ #OppSig inputs t (< 4 >) (< 17 >) (Opp-4-17 t inputs);
         f16 = const [ (λ ()) , (λ ()) ]';
         f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f20 = const $ const $ #OppSig inputs t (< 4 >) (< 20 >) (Opp-4-20 t inputs)})
OppSig4 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
  = elimGwiliRouteInspect (ψ''' inputs t (# 3) (< 4 >)) rt₂
      (record {
         f3  = const [ (λ ()) , (λ ()) ]';
         f2  = const $ const $ #OppSig inputs t (< 4 >) (< 2 >) (Opp-4-2 t inputs);
         f6  = const $ const $ #OppSig inputs t (< 4 >) (< 6 >) (Opp-4-6 t inputs);
         f4  = ⊥-elim ∘ eq ∘ sym;
         f19 = const $ const $ #OppSig inputs t (< 4 >) (< 19 >) (Opp-4-19 t inputs);
         f17 = const $ const $ #OppSig inputs t (< 4 >) (< 17 >) (Opp-4-17 t inputs);
         f16 = const [ (λ ()) , (λ ()) ]';
         f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
         f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
         f20 = const $ const $ #OppSig inputs t (< 4 >) (< 20 >) (Opp-4-20 t inputs)})
OppSig4 inputs t rt₂ ts eq (inj₂ (inj₂ ()))

OppSig19 : ψ'' < 19 >
OppSig19 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 2) (< 19 >)) rt₂
      (record {
         f3  = const $ const $ #OppSig inputs t (< 19 >) (< 3 >) (Opp-19-3 t inputs);
         f2  = const $ const $ #OppSig inputs t (< 19 >) (< 2 >) (Opp-19-2 t inputs);
         f6  = const $ const $ #OppSig inputs t (< 19 >) (< 6 >) (Opp-19-6 t inputs);
         f4  = const $ const $ #OppSig inputs t (< 19 >) (< 4 >) (Opp-19-4 t inputs);
         f19 = ⊥-elim ∘ eq ∘ sym;
         f17 = const $ const $ #OppSig inputs t (< 19 >) (< 17 >) (Opp-19-17 t inputs);
         f16 = const [ (λ ()) , (λ ()) ]';
```

```
          f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f20 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]'})
OppSig19 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
  = elimGwiliRouteInspect (ψ''' inputs t (# 1) (< 19 >)) rt₂
      (record {
          f3  = const $ const $ #OppSig inputs t (< 19 >) (< 3 >) (Opp-19-3 t inputs);
          f2  = const $ const $ #OppSig inputs t (< 19 >) (< 2 >) (Opp-19-2 t inputs);
          f6  = const $ const $ #OppSig inputs t (< 19 >) (< 6 >) (Opp-19-6 t inputs);
          f4  = const $ const $ #OppSig inputs t (< 19 >) (< 4 >) (Opp-19-4 t inputs);
          f19 = ⊥-elim ∘ eq ∘ sym;
          f17 = const $ const $ #OppSig inputs t (< 19 >) (< 17 >) (Opp-19-17 t inputs);
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f20 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]'})
OppSig19 inputs t rt₂ ts eq (inj₂ (inj₂ ()))

OppSig17 : ψ'' < 17 >
OppSig17 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 2) (< 17 >)) rt₂
      (record {
          f3  = const $ const $ #OppSig inputs t (< 17 >) (< 3 >) (Opp-17-3 t inputs);
          f2  = const $ const $ #OppSig inputs t (< 17 >) (< 2 >) (Opp-17-2 t inputs);
          f6  = const $ const $ #OppSig inputs t (< 17 >) (< 6 >) (Opp-17-6 t inputs);
          f4  = const $ const $ #OppSig inputs t (< 17 >) (< 4 >) (Opp-17-4 t inputs);
          f19 = const $ const $ #OppSig inputs t (< 17 >) (< 19 >) (Opp-17-19 t inputs);
          f17 = ⊥-elim ∘ eq ∘ sym;
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f20 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]'})
OppSig17 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
  = elimGwiliRouteInspect (ψ''' inputs t (# 1) (< 17 >)) rt₂
      (record {
          f3  = const $ const $ #OppSig inputs t (< 17 >) (< 3 >) (Opp-17-3 t inputs);
          f2  = const $ const $ #OppSig inputs t (< 17 >) (< 2 >) (Opp-17-2 t inputs);
          f6  = const $ const $ #OppSig inputs t (< 17 >) (< 6 >) (Opp-17-6 t inputs);
          f4  = const $ const $ #OppSig inputs t (< 17 >) (< 4 >) (Opp-17-4 t inputs);
          f19 = const $ const $ #OppSig inputs t (< 17 >) (< 19 >) (Opp-17-19 t inputs);
          f17 = ⊥-elim ∘ eq ∘ sym;
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f20 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]'})
OppSig17 inputs t rt₂ ts eq (inj₂ (inj₂ ()))

OppSig16 : ψ'' < 16 >
OppSig16 inputs t rt₂ ts eq (inj₂ ())
OppSig16 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 0) (< 16 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = ⊥-elim ∘ eq ∘ sym;
          f7  = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f5  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f18 = const [ (λ ()) , [ (λ ()) , [ (λ ()) , (λ ()) ]' ]' ]';
          f20 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]'})

OppSig5 : ψ'' < 5 >
OppSig5 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 4) (< 5 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const $ const $ #OppSig inputs t (< 5 >) (< 7 >) (Opp-5-7 t inputs);
          f5  = ⊥-elim ∘ eq ∘ sym;
          f18 = const $ const $ #OppSig inputs t (< 5 >) (< 18 >) (Opp-5-18 t inputs);
          f20 = const $ const $ #OppSig inputs t (< 5 >) (< 20 >) (Opp-5-20 t inputs)})
OppSig5 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
  = elimGwiliRouteInspect (ψ''' inputs t (# 5) (< 5 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
```

```
          f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const $ const $ #OppSig inputs t (< 5 >) (< 7 >) (Opp-5-7 t inputs);
          f5  = ⊥-elim ∘ eq ∘ sym;
          f18 = const $ const $ #OppSig inputs t (< 5 >) (< 18 >) (Opp-5-18 t inputs);
          f20 = const $ const $ #OppSig inputs t (< 5 >) (< 20 >) (Opp-5-20 t inputs)})
OppSig5 inputs t rt₂ ts eq (inj₂ (inj₂ ()))

OppSig7 : ψ'' < 7 >
OppSig7 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 7) (< 7 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = ⊥-elim ∘ eq ∘ sym;
          f5  = const $ const $ #OppSig inputs t (< 7 >) (< 5 >) (Opp-7-5 t inputs);
          f18 = const $ const $ #OppSig inputs t (< 7 >) (< 18 >) (Opp-7-18 t inputs);
          f20 = const $ const $ #OppSig inputs t (< 7 >) (< 20 >) (Opp-7-20 t inputs)})
OppSig7 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
  = elimGwiliRouteInspect (ψ''' inputs t (# 4) (< 7 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = ⊥-elim ∘ eq ∘ sym;
          f5  = const $ const $ #OppSig inputs t (< 7 >) (< 5 >) (Opp-7-5 t inputs);
          f18 = const $ const $ #OppSig inputs t (< 7 >) (< 18 >) (Opp-7-18 t inputs);
          f20 = const $ const $ #OppSig inputs t (< 7 >) (< 20 >) (Opp-7-20 t inputs)})
OppSig7 inputs t rt₂ ._  eq (inj₂ (inj₂ (inj₁ refl)))
  = elimGwiliRouteInspect (ψ''' inputs t (# 5) (< 7 >)) rt₂
      (record {
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';  f3  = const [ (λ ()) , (λ ()) ]';
          f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';  f16 = const [ (λ ()) , (λ ()) ]';
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';  f7  = ⊥-elim ∘ eq ∘ sym;
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f5  = const $ const $ #OppSig inputs t (< 7 >) (< 5 >) (Opp-7-5 t inputs);
          f18 = const $ const $ #OppSig inputs t (< 7 >) (< 18 >) (Opp-7-18 t inputs);
          f20 = const $ const $ #OppSig inputs t (< 7 >) (< 20 >) (Opp-7-20 t inputs)})
OppSig7 inputs t rt₂ ts eq (inj₂ (inj₂ (inj₂ ())))

OppSig18 : ψ'' < 18 >
OppSig18 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 4) (< 18 >)) rt₂
      (record {
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';  f3  = const [ (λ ()) , (λ ()) ]';
          f6  = const $ const $ #OppSig inputs t (< 18 >) (< 6 >) (Opp-18-6 t inputs);
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const $ const $ #OppSig inputs t (< 18 >) (< 7 >) (Opp-18-7 t inputs);
          f5  = const $ const $ #OppSig inputs t (< 18 >) (< 5 >) (Opp-18-5 t inputs);
          f18 = ⊥-elim ∘ eq ∘ sym;
          f20 = const $ const $ #OppSig inputs t (< 18 >) (< 20 >) (Opp-18-20 t inputs)})
OppSig18 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
  = elimGwiliRouteInspect (ψ''' inputs t (# 7) (< 18 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f6  = const $ const $ #OppSig inputs t (< 18 >) (< 6 >) (Opp-18-6 t inputs);
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const $ const $ #OppSig inputs t (< 18 >) (< 7 >) (Opp-18-7 t inputs);
          f5  = const $ const $ #OppSig inputs t (< 18 >) (< 5 >) (Opp-18-5 t inputs);
          f18 = ⊥-elim ∘ eq ∘ sym;
          f20 = const $ const $ #OppSig inputs t (< 18 >) (< 20 >) (Opp-18-20 t inputs)})
OppSig18 inputs t rt₂ ._  eq (inj₂ (inj₂ (inj₁ refl)))
  = elimGwiliRouteInspect (ψ''' inputs t (# 6) (< 18 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f6  = const $ const $ #OppSig inputs t (< 18 >) (< 6 >) (Opp-18-6 t inputs);
```

```
          f4  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const $ const $ #OppSig inputs t (< 18 >) (< 7 >) (Opp-18-7 t inputs);
          f5  = const $ const $ #OppSig inputs t (< 18 >) (< 5 >) (Opp-18-5 t inputs);
          f18 = ⊥-elim ∘ eq ∘ sym;
          f20 = const $ const $ #OppSig inputs t (< 18 >) (< 20 >) (Opp-18-20 t inputs)})
OppSig18 inputs t rt₂ ts eq (inj₂ (inj₂ (inj₂ ())))

OppSig20 : ψ'' < 20 >
OppSig20 inputs t rt₂ ._ eq (inj₁ refl)
  = elimGwiliRouteInspect (ψ''' inputs t (# 4) (< 20 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const $ const $ #OppSig inputs t (< 20 >) (< 2 >) (Opp-20-2 t inputs);
          f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f4  = const $ const $ #OppSig inputs t (< 20 >) (< 4 >) (Opp-20-4 t inputs);
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const $ const $ #OppSig inputs t (< 20 >) (< 7 >) (Opp-20-7 t inputs);
          f5  = const $ const $ #OppSig inputs t (< 20 >) (< 5 >) (Opp-20-5 t inputs);
          f18 = const $ const $ #OppSig inputs t (< 20 >) (< 18 >) (Opp-20-18 t inputs);
          f20 = ⊥-elim ∘ eq ∘ sym})
OppSig20 inputs t rt₂ ._  eq (inj₂ (inj₁ refl))
  = elimGwiliRouteInspect (ψ''' inputs t (# 3) (< 20 >)) rt₂
      (record {
          f3  = const [ (λ ()) , (λ ()) ]';
          f2  = const $ const $ #OppSig inputs t (< 20 >) (< 2 >) (Opp-20-2 t inputs);
          f6  = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f4  = const $ const $ #OppSig inputs t (< 20 >) (< 4 >) (Opp-20-4 t inputs);
          f19 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f17 = const [ (λ ()) , [ (λ ()) , (λ ()) ]' ]';
          f16 = const [ (λ ()) , (λ ()) ]';
          f7  = const $ const $ #OppSig inputs t (< 20 >) (< 7 >) (Opp-20-7 t inputs);
          f5  = const $ const $ #OppSig inputs t (< 20 >) (< 5 >) (Opp-20-5 t inputs);
          f18 = const $ const $ #OppSig inputs t (< 20 >) (< 18 >) (Opp-20-18 t inputs);
          f20 = ⊥-elim ∘ eq ∘ sym})
OppSig20 inputs t rt₂ ts eq (inj₂ (inj₂ ()))

OppSig : ψ
OppSig inputs t
  = elimGwiliRoute (\ rt₁ → ψ' inputs t rt₁)
      (record {
        f3  = OppSig3 inputs t;
        f2  = OppSig2 inputs t;
        f6  = OppSig6 inputs t;
        f4  = OppSig4 inputs t;
        f19 = OppSig19 inputs t;
        f17 = OppSig17 inputs t;
        f16 = OppSig16 inputs t;
        f7  = OppSig7 inputs t;
        f5  = OppSig5 inputs t;
        f18 = OppSig18 inputs t;
        f20 = OppSig20 inputs t})
```

```
module Gwili.Ladder.Facing where

open import Boolean.Formula

open import Gwili.Ladder
open import Gwili.Abstract
open import Gwili.Abstract-level2

open import Ladder.Core
open import Ladder.Decidable
open import Ladder.LockingTable

open import Data.Nat
open import Data.Product
open import Data.Sum
open import Data.List
open import Data.Unit
open import Data.Bool

open import Relation.Binary.PropositionalEquality

φ-points13/fpl12 : PL-Formula
φ-points13/fpl12 = ((¥ 4 || ¥ 2 || ¥ 6) => ¥ 12)

φ-points14/fpl15 : PL-Formula
φ-points14/fpl15 = ((¥ 18 || ¥ 20 || ¥ 7) => ¥ 15)

points13/fpl12 : ∀ t inp
               → ⟦safety⟧π φ-points13/fpl12 (nthState inp t) (inp t) (nthState inp (suc t))
points13/fpl12 t inp = subst (λ f → ⟦safety⟧π φ-points13/fpl12 (f inp t) (inp t) (f inp (suc t)))
                             (sym nthState-eq)
                             (nthLadderStateCorrect {gwiliLadder} GwiliDecidableLadder
                                φ-points13/fpl12 (inductiveProof gwiliLadder φ-points13/fpl12) t
                                inp)

points14/fpl15 : ∀ t inp
               → ⟦safety⟧π φ-points14/fpl15 (nthState inp t) (inp t) (nthState inp (suc t))
points14/fpl15 t inp = subst (λ f → ⟦safety⟧π φ-points14/fpl15 (f inp t) (inp t) (f inp (suc t)))
                             (sym nthState-eq)
                             (nthLadderStateCorrect {gwiliLadder} GwiliDecidableLadder
                                φ-points14/fpl15 (inductiveProof gwiliLadder φ-points14/fpl15) t
                                inp)

Faceing-13 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ φ-points13/fpl12 ⟧pl
Faceing-13 t inp = lem-mkenv-++-pl' ((¥ 4 || ¥ 2 || ¥ 6) => ¥ 12)
                     (proj₁ (nthState inp t))
                     (proj₁ (proj₁ (inp t)) ++ proj₁ (nthState inp (suc t)))
                     (subst (λ n → T (bound n φ-points13/fpl12))
                      (sym (proj₂ (nthState inp t))) tt)
                     (points13/fpl12 t inp)

Faceing-14 : ∀ t inp → ⟦ mkenv (proj₁ (nthState inp t)) ⊧ φ-points14/fpl15 ⟧pl
Faceing-14 t inp = lem-mkenv-++-pl' φ-points14/fpl15
                     (proj₁ (nthState inp t))
                     (proj₁ (proj₁ (inp t)) ++ proj₁ (nthState inp (suc t)))
                     (subst (λ n → T (bound n φ-points14/fpl15))
                      (sym (proj₂ (nthState inp t))) tt)
                     (points14/fpl15 t inp)
```

```
module Gwili.FacingPointLock where

open import RDM.fixedtrains
open import RDM.RailYard

open import Gwili.Layout
open import Gwili.State
open import Gwili.Ladder
open import Gwili.Abstract
open import Gwili.Ladder.Facing

open import Boolean.Formula renaming (¥ to var)

open import Data.Nat
open import Data.Sum
open import Data.Product
open import Data.List
open import Data.List.Inhabitence
open import Data.Fin using (Fin;zero;suc;#_)
open import Data.Bool

open import Relation.Binary.PropositionalEquality

open import PropIso

lem-aspmap : ∀ b → aspmap b ≡ Proceed → T b
lem-aspmap false ()
lem-aspmap true _ = tt

private
  ψ : ℕ → Fin 9 → Aspect → Locking → Set
  ψ v ts a l = ∀ inputs t rt → (⟦ mkenv (proj₁ (nthState inputs t)) ⊧ (var (unRoute rt)) ⟧pl
                                → ⟦  mkenv (proj₁ (nthState inputs t)) ⊧ (var v) ⟧pl)
                             → archSignalState (nthState inputs t) rt ≡ a
                             → archSegmentLock (nthState inputs t) ts ≡ l

-- Assemble opposing signal proof
#SegLock2 : ψ 12 (# 2) Proceed Locked
#SegLock2 inputs t < rt > p eq rewrite Tb (p (lem-aspmap _ eq)) = refl

#SegLock4 : ψ 15 (# 4) Proceed Locked
#SegLock4 inputs t < rt > p eq rewrite Tb (p (lem-aspmap _ eq)) = refl

#SegLock7 : ψ 15 (# 7) Proceed Locked
#SegLock7 inputs t < rt > p eq rewrite Tb (p (lem-aspmap _ eq)) = refl

private
  ψ' : ∀ inp t rt → Set
  ψ' inp t rt = archSignalState (nthState inp t) rt ≡ Proceed
        → (ts : Fin 9)
        → ts isin (ControlTableEntry.segments $ gwiliRoute rt)
        → ts isin (ControlTableEntry.facing $ gwiliRoute rt)
        → archSegmentLock (nthState inp t) ts ≡ Locked

SegLock : ∀ inputs t rt → ψ' inputs t rt
SegLock inputs t
  = elimGwiliRoute (ψ' inputs t)
      (record {
        f3  = λ _ _ _ → ⊥-elim; f19 = λ _ _ _ → ⊥-elim;
        f17 = λ _ _ _ → ⊥-elim; f16 = λ _ _ _ → ⊥-elim;
        f2  = λ { eq ._ p (inj₁ refl) →
                    #SegLock2 inputs t (< 2 >) (Faceing-13 t inputs ∘ inj₂ ∘ inj₁) eq ;
                  eq ts p (inj₂ ()) };
        f6  = λ { eq ._ p (inj₁ refl) →
                    #SegLock2 inputs t (< 6 >) (Faceing-13 t inputs ∘ inj₂ ∘ inj₂) eq ;
                  eq ts p (inj₂ ()) };
        f4  = λ { eq ._ p (inj₁ refl) →
                    #SegLock2 inputs t (< 4 >) (Faceing-13 t inputs ∘ inj₁) eq ;
                  eq ts p (inj₂ ()) };
        f7  = λ { eq ._ p (inj₁ refl) →
                    #SegLock4 inputs t (< 7 >) (Faceing-14 t inputs ∘ inj₂ ∘ inj₂) eq ;
                  eq ts p (inj₂ ()) };
        f5  = λ _ _ _ → ⊥-elim;
        f18 = λ { eq ._ p (inj₁ refl) →
                    #SegLock4 inputs t (< 18 >) (Faceing-14 t inputs ∘ inj₁) eq ;
                  eq ts p (inj₂ ()) };
        f20 = λ { eq ._ p (inj₁ refl) →
                    #SegLock4 inputs t (< 20 >) (Faceing-14 t inputs ∘ inj₂ ∘ inj₁) eq ;
                  eq ts p (inj₂ ()) } })
```

```
module Gwili.Interlocking where

open import RDM.fixedtrains
open import RDM.RailYard

open import Gwili.Layout
open import Gwili.State
open import Gwili.OpposingSignals
open import Gwili.FacingPointLock
open import Gwili.Abstract

open import Relation.Binary.PropositionalEquality
open import Relation.Decidable

open import Data.Nat
open import Data.Sum
open import Data.Product
open import Data.List
open import Data.List.Inhabitence

mutual
  CorrectTrainsAux : ∀ inp tinp t tr rt l
                   → (∀ x → x isin l → x isin gwiliRouteConnections (trainPosition inp tinp t tr))
                → trainPosition inp tinp t tr ≡ trainPosition-aux inp tinp t tr rt l
                  ⊎ (GwiliRouteConnections (trainPosition inp tinp t tr)
                                           (trainPosition-aux inp tinp t tr rt l)
                    × archSignalState (nthState inp t)
                                      (trainPosition-aux inp tinp t tr rt l) ≡ Proceed)
  CorrectTrainsAux inp tinp t tr rt [] q = inj₁ refl
  CorrectTrainsAux inp tinp t tr rt (rt' :: rts) q with decroute rt rt'
  CorrectTrainsAux inp tinp t tr rt (rt' :: rts) q | yes p
    with decasp (archSignalState (nthState inp t) rt) Proceed
  CorrectTrainsAux inp tinp t tr rt (rt' :: rts) q | yes p₁ | yes p = inj₂ (q rt (inj₁ p₁) , p)
  CorrectTrainsAux inp tinp t tr rt (rt' :: rts) q | yes p  | no ¬p = inj₁ refl
  CorrectTrainsAux inp tinp t tr rt (rt' :: rts) q | no ¬p
    = CorrectTrainsAux inp tinp t tr rt rts (λ x y → q x (inj₂ y))

  CorrectTrains : ∀ inp tinp t tr
                → trainPosition inp tinp t tr ≡ trainPosition inp tinp (suc t) tr
                  ⊎ (GwiliRouteConnections (trainPosition inp tinp t tr)
                                           (trainPosition inp tinp (suc t) tr)
                    × archSignalState (nthState inp t)
                                      (trainPosition inp tinp (suc t) tr) ≡ Proceed)
  CorrectTrains inp tinp t tr with tinp t tr
  CorrectTrains inp tinp t tr | moving x = CorrectTrainsAux inp tinp t tr x
                                           (gwiliRouteConnections (trainPosition inp tinp t tr))
                                           (λ _ x → x)
  CorrectTrains inp tinp t tr | stationary = inj₁ refl

CorrectInputs-SigGuard : Inputs → TrainInputs → Set
CorrectInputs-SigGuard inp tinp
  = ∀ t tr ts → ts isin (ControlTableEntry.segments (gwiliRoute (trainPosition inp tinp t tr)))
              → ∀ rt
              → ts isin (ControlTableEntry.segments (gwiliRoute rt))
              → archSignalState (nthState inp t) rt ≡ Stop

CorrectInputs-TrainLocked : Inputs → TrainInputs → Set
CorrectInputs-TrainLocked inp tin
  = ∀ t tr ts → ts isin (ControlTableEntry.segments (gwiliRoute (trainPosition inp tin (suc t) tr)))
              → archSegmentLock (nthState inp t) ts ≡ Locked
              → archSegmentLock (nthState inp (suc t)) ts ≡ Locked

-- the interlocking does not enforce that signals guard, i.e. does not clear when a train is
-- trailing a signal. thus, it is assumed that the signal man enforces this principle.
gwilirailway : (i : Inputs)
             → (ti : TrainInputs)
             → CorrectInputs-SigGuard i ti
             → CorrectInputs-TrainLocked i ti
             → Railway GwiliLayout
gwilirailway inputs tinputs sigguard trainlock
  = record {
      layoutState = λ n → toArchState n inputs tinputs;
      CorrectTrains-Route = CorrectTrains inputs tinputs;
      Principle-SignalsGuard = sigguard;
      Principle-OpposingSignals = OppSig inputs;
      Principle-ProceedLocked = SegLock inputs;
      Principle-TrainLocked = trainlock}
```

```
module Gwili.Safe where

open import RDM.fixedtrains

open import Gwili.State
open import Gwili.Interlocking
open import Gwili.Layout
open import Gwili.Abstract

open import Data.Fin
open import Data.Sum
open import Data.Product
open import Data.Empty

open import Relation.Binary.PropositionalEquality

open import PropIso

open import Ladder.Core

open Layout

GwiliNoCrashInit : (tr₁ tr₂ : Fin 2)
                 → tr₁ ≡ tr₂
                   ⊎ (∀ ts → SegInRoute GwiliLayout ts (initTrainPosition tr₁)
                           → ¬ SegInRoute GwiliLayout ts (initTrainPosition tr₂))
GwiliNoCrashInit zero zero = inj₁ refl
GwiliNoCrashInit zero (suc zero)
  = inj₂ λ { ._ (inj₁ refl) → λ { (inj₁ ()) ; (inj₂ ()) } ; ts (inj₂ ()) }
GwiliNoCrashInit zero (suc (suc ()))
GwiliNoCrashInit (suc zero) zero
  = inj₂ λ { ._ (inj₁ refl) → λ { (inj₁ ()) ; (inj₂ ()) } ; ts (inj₂ ()) }
GwiliNoCrashInit (suc zero) (suc zero) = inj₁ refl
GwiliNoCrashInit (suc zero) (suc (suc ()))
GwiliNoCrashInit (suc (suc ())) tr₂

GwiliLockInit : ∀ i tr ts
              → SegInRoute GwiliLayout ts (initTrainPosition tr)
              → FacingInRoute GwiliLayout ts (initTrainPosition tr)
              → archSegmentLock (nthState i 0) ts ≡ Locked
GwiliLockInit i zero ts q p = ⊥-elim p
GwiliLockInit i (suc zero) ts q p = ⊥-elim p
GwiliLockInit i (suc (suc ())) ts q p

Safe : (i : Inputs)
     → (ti : TrainInputs)
     → (q₁ : CorrectInputs-SigGuard i ti)
     → (q₂ : CorrectInputs-TrainLocked i ti)
     → TrainsDontCrash (gwilirailway i ti q₁ q₂)
       × FacingPointLock (gwilirailway i ti q₁ q₂)
Safe i ti q₁ q₂ = record { InitialSafe = GwiliNoCrashInit }
                , record { InitialSafe = GwiliLockInit i }
```

```
module Gwili.GwiliSimulator where

open import IO.Console
open import Foreign.Haskell
open import Data.Char
open import Data.Colist
open import Data.Bool
open import Data.String as String hiding (_==_) renaming (_++_ to _+++_)
open import Data.Product as Prod
open import Data.Conat hiding (_+_)
open import Data.Nat hiding (_<_)
open import Data.List as List
open import Data.Colist
open import Data.Maybe
open import Data.Sum

open import Coinduction

open import coparse as P hiding (choice)

open import Relation.Binary.PropositionalEquality
open import Relation.Decidable

open import PropIso

open import Gwili.Ladder

open import TransitionSystem.Decidable renaming (DecidableTransitionSystem to DTS)

open import Ladder.LockingTable

open import Boolean.Formula

private
  defhead : ∀ {A : Set} → A → List A → A
  defhead a [] = a
  defhead a (x ∷ _) = x

  deftail : ∀ {A : Set} → List A → List A → List A
  deftail a [] = a
  deftail a (_ ∷ x) = x

decList : ∀ {A : Set} → Decidable (_≡_ {A = A}) → Decidable (_≡_ {A = List A})
decList d [] [] = yes refl
decList d [] (x ∷ y) = no (λ ())
decList d (x ∷ x₁) [] = no (λ ())
decList d (x ∷ x₁) (x₂ ∷ y) with d x x₂
decList d (x ∷ x₁) (x₂ ∷ y) | yes p with decList d x₁ y
decList d (x ∷ x₁) (x₂ ∷ y) | yes p₁ | yes p = yes (cong₂ _∷_ p₁ p)
decList d (x ∷ x₁) (x₂ ∷ y) | yes p | no ¬p = no (λ x₃ → ¬p (cong (deftail []) x₃))
decList d (x ∷ x₁) (x₂ ∷ y) | no ¬p = no (λ x₃ → ¬p (cong (defhead x) x₃))

decState : Decidable (_≡_ {A = List Bool})
decState = decList Data.Bool._≟_

showBool : Bool → String
showBool true = "Reverse"
showBool false = "Normal"

showSig : Bool → String
showSig false = "Danger"
showSig true  = "Clear "

showLock : Bool → String
showLock true = "Locked"
showLock false = "Unlocked"

showState : Σ[ l ∶ List Bool ] (List.length l ≡ 22) → String
showState (_ ∷ _ ∷ sig2 ∷ sig3 ∷ sig4 ∷ sig5 ∷ sig6 ∷ sig7 ∷ _ ∷
           _ ∷ _ ∷ _ ∷ fpl12 ∷ point13 ∷ point14 ∷ fpl15 ∷ sig16 ∷
           sig17 ∷ sig18 ∷ sig19 ∷ sig20 ∷ _ ∷ [] , proj₂) = "\nSignals:\n" +++
  "2: "  +++ showSig sig2  +++ " | " +++  "16: " +++ showSig sig16 +++ "\n" +++
  "3: "  +++ showSig sig3  +++ " | " +++  "17: " +++ showSig sig17 +++ "\n" +++
  "4: "  +++ showSig sig4  +++ " | " +++  "18: " +++ showSig sig18 +++ "\n" +++
  "5: "  +++ showSig sig5  +++ " | " +++  "19: " +++ showSig sig19 +++ "\n" +++
  "6: "  +++ showSig sig6  +++ " | " +++  "20: " +++ showSig sig20 +++ "\n" +++
  "7: "  +++ showSig sig7  +++ "\n\n" +++
  "fpl 12 / point 13:\n  " +++ showLock fpl12 +++ " / " +++ showBool point13 +++ "\n" +++
  "fpl 15 / point 14:\n  " +++ showLock fpl15 +++ " / " +++ showBool point14 +++ "\n\n"
showState _ = "IMPOSSIBLE ERROR"

readLeaverNumber : IOConsole (Σ[ n ∶ ℕ ] (T (n < 22 ∧ 0 < n)))
```

```
readLeaverNumber = do (putStrLn (toCostring "enter leaver to move [1..21]")) \ _ →
                    ♯ do getLine \ l1 →
                    ♯ choice (parseSpaced parseNat l1)
                             (λ n → choice ([ just , const nothing ]' (ex-mid (proj₁ n < 22 ∧
                                                                               0 < proj₁ n )))
                                           (λ np → return (proj₁ n , np))
                                           readLeaverNumber)
                             readLeaverNumber

lowlist : (n : ℕ) → Σ[ l ∶ List Bool ] (List.length l ≡ n)
lowlist 0 = [] , refl
lowlist (suc n) = Prod.map (λ x → false ∷ x) (cong suc) (lowlist n)

mklist : (length high : ℕ) → List Bool
mklist zero h = []
mklist (suc l) zero = true ∷ proj₁ (lowlist l)
mklist (suc l) (suc h) = false ∷ mklist l h

lem-mklist : ∀ (length high : ℕ) → List.length (mklist length high) ≡ length
lem-mklist zero h = refl
lem-mklist (suc l) zero = cong suc (proj₂ (lowlist l))
lem-mklist (suc l) (suc h) = cong suc (lem-mklist l h)

--takes a number n in [1..21] and produces an input list l, such
--      that l[n] = true and all m!=n l[m]=false
mkinputs : ((Σ[ n ∶ ℕ ] (T (n < 22 ∧ 0 < n)))) → Σ[ l ∶ List Bool ] (List.length l ≡ 22)
mkinputs ((n , np))  = (mklist 22 n) , (lem-mklist 22 n)

mkDisj' : ℕ → PL-Formula
mkDisj' zero = ¥false
mkDisj' (suc n) = ¥ n || mkDisj' n

mkInvariant-aux' : ℕ → PL-Formula
mkInvariant-aux' zero = ¥true
mkInvariant-aux' (suc n) = (¥ (n) => ~ (mkDisj' n)) && mkInvariant-aux' n

lem-lowlist : ∀ a b → ¬ T (mkenv (proj₁ (lowlist a)) b)
lem-lowlist zero b = id
lem-lowlist (suc a) zero = id
lem-lowlist (suc a) (suc b) = lem-lowlist a b

lem-mklist' : ∀ a b c → T (mkenv (mklist a b) c) → b ≡ c
lem-mklist' zero b c = ⊥-elim
lem-mklist' (suc a) zero zero = const refl
lem-mklist' (suc a) zero (suc c) = ⊥-elim ∘ lem-lowlist a c
lem-mklist' (suc a) (suc b) zero = ⊥-elim
lem-mklist' (suc a) (suc b) (suc c) = cong suc ∘ lem-mklist' a b c

disj-bound : ∀ a → T (bound a (mkDisj' a) )
disj-bound zero = _
disj-bound (suc a) = ∧-intro _ _ (<-ord a)
                                 (injbound (mkDisj' a) a (suc a)
                                   (<-rsuc a (suc a) (<-ord a)) (disj-bound a))

lem-mkmk-bound : ∀ a b → T (b < a)
               → mkenv (mklist (suc (suc a)) (suc a)) b ≡ mkenv (mklist (suc a) a) b
lem-mkmk-bound zero b ()
lem-mkmk-bound (suc a) zero p = refl
lem-mkmk-bound (suc a) (suc b) p = lem-mkmk-bound a b p

lem-mkmk : ∀ a → ¬ ⟦ mkenv (mklist (suc a) a) ⊧ mkDisj' a ⟧pl
lem-mkmk zero p = p
lem-mkmk (suc zero) (inj₁ x) = x
lem-mkmk (suc (suc a)) (inj₁ x) = lem-mkmk (suc a) (inj₁ x)
lem-mkmk (suc a) (inj₂ y) = lem-mkmk a
                              (env-eq-bound-subst (mkenv (mklist (suc (suc a)) (suc a)))
                               (mkenv (mklist (suc a) a)) (mkDisj' a) a (disj-bound a)
                               (lem-mkmk-bound a) y)

↯ : ∀ {w} {Whatever : Set w} → ⊥ → Whatever
↯ = ⊥-elim

Empty : ℕ → Set
Empty 0 = ⊥
Empty (suc n) = ⊥ ⊎ Empty n

↯⊎ : ∀ {w} {Whatever : Set w} → (n : ℕ) → Empty n → Whatever
↯⊎ zero = ⊥-elim
↯⊎ (suc n) = [ ⊥-elim , ↯⊎ n ]'

inputsproof : (p : Σ[ n ∶ ℕ ] (T (n < 22 ∧ 0 < n)))
            → ⟦ mkenv (proj₁ (mkinputs p)) ⊧ mkInvariant-aux 0 22 ⟧pl
inputsproof (0 , p) = ⊥-elim p
```

```
inputsproof (1  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 1)  , ↯ , _
inputsproof (2  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 2)  , ↯ , ↯ , _
inputsproof (3  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 3)  , ↯ , ↯ , ↯ , _
inputsproof (4  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 4)  , ↯ , ↯ , ↯ , ↯ , _
inputsproof (5  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , const (↯ω 5)  , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (6  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , const (↯ω 6)  , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (7  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , const (↯ω 7)  , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (8  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       const (↯ω 8)  , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (9  , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       const (↯ω 9)  , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (10 , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 10) ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (11 , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 11) ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (12 , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 12) , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (13 , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 13) , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (14 , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 14) , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (15 , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 15) , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (16 , p) = ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 16) , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (17 , p) = ↯ , ↯ , ↯ , ↯ , ↯ , const (↯ω 17) , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (18 , p) = ↯ , ↯ , ↯ , ↯ , const (↯ω 18) , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (19 , p) = ↯ , ↯ , ↯ , const (↯ω 19) , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (20 , p) = ↯ , ↯ , const (↯ω 20) , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
                       ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
-- agda does not support builtin naturals > 20 in case distinctions
inputsproof (suc (suc (suc (suc (suc (suc (suc
             (suc (suc (suc (suc (suc (suc (suc (suc
             (suc (suc (suc (suc (suc zero))))))))))))))))))))) , p)
  = const (↯ω 21) , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ ,
    ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , ↯ , _
inputsproof (suc (suc (suc (suc (suc (suc (suc (suc
             (suc (suc (suc (suc (suc (suc (suc (suc
             (suc (suc (suc (suc (suc (suc n))))))))))))))))))))) , p) = ⊥-elim p

mkinput : Σ[ n ∶ ℕ ] (T (n < 22 ∧ 0 < n))
        → Σ[ m ∶ Σ[ l ∶ List Bool ] (List.length l ≡ 22) ]
             ⟦ mkenv (proj₁ m) ⊧ mkInvariant-aux 0 22 ⟧pl
mkinput p = mkinputs p  , inputsproof p

-- readExit : IOConsole Choice
-- readExit = do (putStrLn (toCostring "do you want to exit? [yes/no])) \ _ →
--          ♯ do getLine \ l1 →
--          ♯ choice (parseChoice l1) (return ∘ proj₁) readExit

mutual
  mainloop : Σ[ l ∶ List Bool ] (List.length l ≡ 22) → IOConsole Unit
  mainloop s = (do (putStrLn (toCostring (showState s))) \ _ → ♯ return unit) >>= const
               (♯ readLeaverNumber) >>= λ inp →
               ♯ eval (DecidableTransitionSystem.transitionFunction GwiliDecidableLadder
                                                    s (mkinput inp)) s

  eval : Σ[ l ∶ List Bool ] (List.length l ≡ 22)
       → Σ[ l ∶ List Bool ] (List.length l ≡ 22)
       → IOConsole Unit
  eval new old with decState (proj₁ new) (proj₁ old)
  eval new old | yes p = do (putStrLn (toCostring "## Request failed ##")) \ _ →
                         ♯ mainloop new
  eval new old | no ¬p = mainloop new

program : IOConsole Unit
program = (do (putStrLn (toCostring
                    "Gwili Rail simulator\nentering main loop...\n\ninitial state:")) \ _ →
        ♯ mainloop (DTS.initialState GwiliDecidableLadder))

main : PrimIO Unit
main = translateIOConsole program
```

Bibliography

[Abe98] A. Abel. Foetus - termination checker for simple functional programs. *Programming Lab Report*, 1998.

[ABH+10] J. Abrial, M. Butler, S. Hallerstede, T. Hoang, F. Mehta, and L. Voisin. Rodin: an open toolset for modelling and reasoning in Event-B. *International Journal on Software Tools for Technology Transfer (STTT)*, 12(6):447–466, 2010.

[ABK+02] E. Astesiano, M. Bidoit, H. Kirchner, B. Krieg-Brückner, P. Mosses, D. Sannella, and A. Tarlecki. CASL: The common algebraic specification language. *Theoretical Computer Science*, 286(2):153–196, 2002.

[Abr96] J. Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 1996.

[ACN90] L. Augustsson, T. Coquand, and B. Nordstrom. A short description of another logical framework. In *Proceedings of the First Workshop on Logical Frameworks*, volume 100, pages 39–42, Antibes, 1990.

[Agd12] The Agda Wiki. Web Page:`http://wiki.portal.chalmers.se/agda`, July 2012.

[AGST10] M. Armand, B. Grégoire, A. Spiwack, and L. Théry. Extending Coq with imperative features and its application to SAT verification. In M. Kaufmann and L. Paulson, editors, *Interactive Theorem Proving*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer Berlin / Heidelberg, 2010.

[AMM07] T. Altenkirch, C. McBride, and P. Morris. Generic programming with dependent types. In R. Backhouse, J. Gibbons, R. Hinze, and

J. Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 209–257. Springer Berlin / Heidelberg, 2007.

[BBFM99] P. Behm, P. Benoit, A. Faivre, and J.-M. Meynadier. Météor: A successful application of B in a large project. In J. Wing, J. Woodcock, and J. Davies, editors, *FM99 Formal Methods*, volume 1708 of *Lecture Notes in Computer Science*, pages 369–387. Springer Berlin / Heidelberg, 1999.

[BBM99] D. Bjørner, J. Braad, and K. Mogensen. Models of railway systems: Domain. In T. Lecomte and P. Larsen, editors, *Proceedings of FME Rail Workshop #5*, volume 5, Toulouse, France, 1999.

[BBV95] T. Basten, R. Bol, and M. Voorhoeve. Simulating and analyzing railway interlockings in ExSpect. *IEEE Parallel & Distrib. Technol.*, 3(3):50–62, September 1995.

[BCJ+04] D. Bjørner, P. Chiang, M. Jacobsen, J. Hansen, M. Madsen, and M. Penicka. Towards a formal model of CyberRail. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 657–664. Springer Boston, 2004.

[BDN09] A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda – a functional language with dependent types. In S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, pages 73–78. Springer Berlin / Heidelberg, 2009.

[BDSR12] A. Bove, P. Dybjer, and A. Sicard-Ramírez. Combining interactive and automatic reasoning in first order theories of functional programs. In L. Birkedal, editor, *15th International Conference on Foundations of Software Science and Computational Structures, FoSSaCS 2012*, volume 7213 of *Lecture Notes in Computer Science*, pages 104–118, March 2012.

[BFG+98] C. Bernardeschi, A. Fantechi, S. Gnesi, S. Larosa, G. Mongardi, and D. Romano. A formal verification environment for railway signaling system design. *Formal Methods in System Design*, 12(2):139–161, 1998.

[BG94] L. Bachmair and H. Ganzinger. Rewrite-based equational theorem proving with selection and simplification. *Journal of Logic and Computation*, 4(3):217, 1994.

[BG00] J. Boulanger and M. Gallardo. Validation and verification of METEOR safety software. In *International conference on computers in railways*, pages 189–200, 2000.

[BGHL10] G. Bierman, A. Gordon, C. Hriţcu, and D. Langworthy. Semantic subtyping with an SMT solver. In *Proceedings of the 15th ACM SIGPLAN international conference on Functional programming*, pages 105–116. ACM, 2010.

[BGP95] D. Bjørner, C. George, and S. Prehn. Domain analysis – a prerequisite for requirements capture. Technical Report 37, UNU/IIST Document, March 1995.

[BHPS64] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase structure grammars. *Language and Information: Selected Essays on their Theory and Application*, pages 116–150, 1964.

[Bir24] J. Birkinshaw. *Specification of John Birkinshaw's Patent for an Improvement in the Construction of Malleable Iron Rails: To be Used in Rail Roads, with Remarks on the Comparative Merits of Cast Metal and Malleable Iron Railways.* E. Walker, 1824.

[Bjø03] D. Bjørner. Dynamics of railway nets: On an interface between automatic control and software engineering. In S. Tsugawa and M. Aoki, editors, *CTS2003: 10th IFAC Symposium on Control in Transportation Systems*, Tokyo, Japan, 2003.

[Bjø04] D. Bjørner. TRain: The railway domain. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 607–611. Springer Boston, 2004.

[Bjø06] D. Bjørner. *Software Engineering: Domains, Requirements and Software Design*, volume 3 of *Texts in Theoretical Computer Science. An EATCS Series.* Springer, 2006.

[BM90] R. Boyer and J. S. Moore. A theorem prover for a computational logic. In M. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin / Heidelberg, 1990.

[BMS92] J. Berger, P. Middelraad, and A. Smith. EURIS, European railway interlocking specification. Technical Report 7A/16, UIC Commission, May 1992.

[BN10] S. Böhme and T. Nipkow. Sledgehammer: Judgement day. In J. Giesl and R. Hähnle, editors, *Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 107–121. Springer Berlin / Heidelberg, 2010.

[Bou97] S. Boutin. Using reflection to build efficient and certified decision procedures. In M. Abadi and T. Ito, editors, *Theoretical Aspects of Computer Software*, volume 1281 of *Lecture Notes in Computer Science*, pages 515–529. Springer Berlin / Heidelberg, 1997.

[Bra05] E. Brady. *Practical Implementation of a Dependently Typed Functional Programming Language*. PhD thesis, Durham University, 2005.

[BSST09] C. Barrett, R. Sebastiani, S. A. Seshia, and C. Tinelli. *Frontiers in Artificial Intelligence and Applications*, volume 185 of *Handbook of Satisfiability*, chapter 12: Satisfiability Modulo Theories, pages 737–797. IOS Press, 2009.

[Bun99] A. Bundy. The automation of proof by mathematical induction. In *Informatics Report Series*, number EDI-INF-RR-0002. 1999.

[Cal92] H. G. Calcraft. Requirements of the board of trade, in regard to the opening of railways. Board of Trade, Her Majesty's Stationary Office, 1892.

[CCG+02] A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV 2: An open-source tool for symbolic model checking. In E. Brinksma and K. Larsen, editors, *Computer Aided Verification*, volume 2404 of *Lecture Notes in Computer Science*, pages 241–268. Springer Berlin / Heidelberg, 2002.

[CFCC58] H. Curry, R. Feys, W. Craig, and W. Craig. *Combinatory Logic*, volume 1. North-Holland, 1958.

[CGP99] E. Clarke, O. Grumberg, and D. Peled. *Model Checking*. MIT Press, 1999.

[Cri87] A. Cribbens. Solid-state interlocking: an integrated electronic signalling system for mainline railways. *IEE Proceedings B. Electric Power Applications*, 134:148–58, 1987.

[CS03] K. Claessen and N. Sörensson. New techniques that improve MACE-style finite model finding. In *Proceedings of the CADE-19 Workshop: Model Computation-Principles, Algorithms, Applications*, pages 11–27, 2003.

[Cur34] H. Curry. Functionality in combinatory logic. *Proceedings of the National Academy of Sciences of the United States of America*, 20(11):584, 1934.

[dB70] N. de Bruijn. The mathematical language AUTOMATH, its usage, and some of its extensions. In M. Laudet, D. Lacombe, L. Nolin, and M. Schtzenberger, editors, *Symposium on Automatic Demonstration*, volume 125 of *Lecture Notes in Mathematics*, pages 29–61. Springer Berlin / Heidelberg, 1970.

[Ded63] R. Dedekind. *Vorlesungen über Zahlentheorie*. F. Vieweg und Sohn, 1863.

[dMB08] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In C. Ramakrishnan and J. Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer Berlin / Heidelberg, 2008.

[dMB09] L. de Moura and N. Bjørner. Satisfiability modulo theories: An appetizer. In M. Oliveira and J. Woodcock, editors, *Formal Methods: Foundations and Applications*, volume 5902 of *Lecture Notes in Computer Science*, pages 23–36. Springer Berlin / Heidelberg, 2009.

[DPR61] M. Davis, H. Putnam, and J. Robinson. The decision problem for exponential diophantine equations. *The Annals of Mathematics*, 74(3):425–436, 1961.

[DRS03] Y. Dong, C. Ramakrishnan, and S. Smolka. Model checking and evidence exploration. In *Proceedings, 10th IEEE International Conference and Workshop on the Engineering of Computer-Based Systems*, pages 214–223. IEEE Computer Society Press, April 2003.

[DS01] P. Dybjer and A. Setzer. Indexed induction-recursion. In R. Kahle, P. Schroeder-Heister, and R. Stärk, editors, *Proof Theory in Computer Science*, volume 2183 of *Lecture Notes in Computer Science*, pages 93–113. Springer Berlin / Heidelberg, 2001.

[DT84] J. Diller and A. S. Troelstra. Realizability and intuitionistic logic. *Synthese*, 60:253–282, 1984. 10.1007/BF00485463.

[Eis02] C. Eisner. Using symbolic CTL model checking to verify the railway stations of Hoorn-Kersenboogerd and Heerhugowaard. *International Journal on Software Tools for Technology Transfer (STTT)*, 4:107–124, 2002.

[Eri96] L. Eriksson. Specifying railway interlocking requirements for practical use. In E. Schoitsch, editor, *Proceedings of the 15th International Conference on Computer Safety, Reliability and Security (SAFECOMP'96)*. Springer-Verlag, 1996.

[Eri97a] L. Eriksson. Formalising railway interlocking requirements. Technical Report 1997:3, The Swedish National Rail Administration (Banverket), 1997. (Also available in Swedish as Formalisering av krav på signalställverk, Technical report 1997:1).

[Eri97b] L. Eriksson. Formal verification of railway interlockings. Technical Report 1997:4, The Swedish National Rail Administration (Banverket), 1997. (Also available in Swedish as Formell verifiering av signalställverk, Technical report 1997:2).

[ES03] N. Een and N. Sorensson. An extensible SAT-solver. *SAT*, 2003.

[FG11] O. Fredriksson and D. Gustafsson. A totaly Epic backend for Agda. Master's thesis, Chalmers University of Technology, Göteborg, Sweden, 2011.

[FGHvV98] W. Fokkink, J. Groote, M. Hollenberg, and B. van Vlijmen. *LARIS 1.0: LAnguage for Railway Interlocking Specification*. CWI, Amsterdam, 1998.

[FHG+98] W. Fokkink, P. Hollingshead, J. Groote, S. Luttik, and J. van Wamel. Verification of interlockings: from control tables to ladder logic diagrams. In *Proceedings of the 3rd Workshop on Formal Methods for Industrial Critical Systems (FMICS'98)*, volume 98, pages 171–185. Stichting Mathematisch Centrum, 1998.

[Fis12] S. Fischer. HackageDB: incremental-sat-solver-0.1.7. Web Page: `http://hackage.haskell.org/package/incremental-sat-solver`, July 2012.

[FMM+06] P. Fontaine, J.-Y. Marion, S. Merz, L. Nieto, and A. Tiu. Expressiveness + automation + soundness: Towards combining SMT solvers and interactive proof assistants. In H. Hermanns and J. Palsberg, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 3920 of *Lecture Notes in Computer Science*, pages 167–181. Springer Berlin / Heidelberg, 2006.

[Fok95] W. Fokkink. Safety criteria for Hoorn-Kersenboogerd railway station. *Logic Group Preprint Series*, 135, 1995.

[FS11] S. Foster and G. Struth. Integrating an automated theorem prover into Agda. In M. Bobaru, K. Havelund, G. Holzmann, and R. Joshi, editors, *NASA Formal Methods*, volume 6617 of *Lecture Notes in Computer Science*, pages 116–130. Springer Berlin / Heidelberg, 2011.

[GB84] J. Goguen and R. Burstall. Introducing institutions. In E. Clarke and D. Kozen, editors, *Logics of Programs*, volume 164 of *Lecture Notes in Computer Science*, pages 221–256. Springer Berlin / Heidelberg, 1984.

[Gen35] G. Gentzen. Untersuchungen über das logische schließen. i. *Mathematische Zeitschrift*, 39:176–210, 1935. Translated as: Investigations into logical deduction, *American Philosophical Quarterly*, Vol. 1, No. 4 (1964), pp. 288-306.

[Geo91] C. George. The RAISE specification language: A tutorial. In S. Prehn and H. Toetenel, editors, *VDM '91 Formal Software Development Methods*, volume 552 of *Lecture Notes in Computer Science*, pages 238–319. Springer Berlin / Heidelberg, 1991.

[GL04] T. Giras and Z. Lin. Stochastic train domain theory framework. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 643–649. Springer Boston, 2004.

[GvVK95] J. Groote, S. van Vlijmen, and J. Koorn. The safety guaranteeing system at station Hoorn-Kersenboogerd. In *Computer Assurance, 1995. COMPASS'95.'Systems Integrity, Software Safety and Process Security'. Proceedings of the Tenth Annual Conference on*, pages 57–68. IEEE, 1995.

[Han98] K. Hansen. Formalising railway interlocking systems. In *Nordic Seminar on Dependable Computing Systems*, pages 83–94, 1998.

[Har08] J. Harrison. Automated and interactive theorem proving. In O. Grumberg, T. Nipkow, and C. Pfaller, editors, *Formal Logical Methods for System Security and Correctness*, volume 14 of *NATO Science for Peace and Security Series*, pages 111–147. IOS Press, 2008.

[Hen02] D. Hendriks. Proof reflection in Coq. *Journal of Automated Reasoning*, 29:277–307, 2002.

[Hoa03] T. Hoare. The verifying compiler: a grand challenge for computing research. In *Proceedings of the 12th international conference on Compiler construction*, CC'03, pages 262–272, Berlin, Heidelberg, 2003. Springer-Verlag.

[How80] W. Howard. The formulae-as-types notion of construction. In J. Hindley and J. Seldin, editors, *To H.B. Curry: essays on combinatory logic, lambda calculus and formalism*, volume 44, pages 479–490. Academic Press, 1980.

[HP00] A. E. Haxthausen and J. Peleska. Formal development and verification of a distributed railway control system. *IEEE Trans. Softw. Eng.*, 26(8):687–701, August 2000.

[HP02] A. Haxthausen and J. Peleska. A domain specific language for railway control systems. In *Proceedings of the sixth biennial world conference on integrated design and process technology, (IDPT 2002)*, pages 23–28, 2002.

[HR04] M. Huth and M. Ryan. *Logic in Computer Science*. Cambridge University Press Cambridge, 2004.

[Hur05] J. Hurd. System description: The metis proof tactic. *Empirically Successful Automated Reasoning in Higher-Order Logic (ESHOL)*, pages 103–104, 2005.

[Hut89] C. S. Hutchinson. Accident Returns: Extract for the Accident at Armagh on 12 June 1889. Board of Trade, Her Majesty's Stationary Office, 1889.

[IRS01] IRSE. *Signalling Philosophy Review*. The Institution of Railway Signalling Engineers, Savoy Hill House, Savoy Hill, London, April 2001.

[Jam10] P. James. SAT-based model checking and its applications to train control software. Master's thesis, Dept. Computer Science, Swansea University, 2010.

[JGB10] C. Jones, G. Grov, and A. Bundy. Ideas for a high-level proof strategy language. Technical Report CS-TR-1210, Newcastle University, 2010.

[Jon90] C. Jones. *Systematic software development using VDM*, volume 2. Prentice Hall, 1990.

[JR10] P. James and M. Roggenbach. SAT-based model checking of train control systems. Technical Report 5-2010, Dipartimento di Matematica e Informatica, Universita di Udine, 2010.

[JR11] P. James and M. Roggenbach. Designing domain specific languages for verification: First steps. In G. S. Peter Hofner, Annabelle McIver, editor, *ATE-2011 – Proc. of the First Workshop on Automated Theory Engineering*, volume 760 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2011.

[Kan08] K. Kanso. Formal verification of ladder logic. Master's thesis, Dept. Computer Science, Swansea University, December 2008.

[Kin94] T. King. Formalising British rails signalling rules. *FME '94: Industrial Benefit of Formal Methods*, 873:45–54, 1994.

[Kir02] M. Kirby. *The Origins of Railway Enterprise: The Stockton and Darlington Railway 1821-1863*. Cambridge Univ Pr, 2002.

[KMS09] K. Kanso, F. Moller, and A. Setzer. Automated verification of signalling principles in railway interlocking systems. *Electronic Notes in Theoretical Computer Science*, 250(2):19–31, 2009.

[Kor08] K. Korovin. iProver – An instantiation-based theorem prover for first-order logic (system description). In A. Armando, P. Baumgartner, and G. Dowek, editors, *Automated Reasoning*, volume 5195 of *Lecture Notes in Computer Science*, pages 292–298. Springer Berlin / Heidelberg, 2008.

[KR01] D. Kerr and T. Rowbotham. *Introduction to Railway Signalling*. Institution of Railway Signal Engineers, Savoy Hill House, Savoy Hill, London, 2001.

[Law11] A. Lawrence. Verification of railway interlockings in Scade. Master's thesis, Dept. Computer Science, Swansea University, 2011.

[LB06] F. Lindblad and M. Benke. A tool for automated theorem proving in Agda. In J.-C. Filliâtre, C. Paulin-Mohring, and B. Werner, editors, *Types for Proofs and Programs*, volume 3839 of *Lecture Notes in Computer Science*, pages 154–169. Springer Berlin / Heidelberg, 2006.

[LC08] S. Lescuyer and S. Conchon. A reflexive formalization of a SAT solver in Coq. In *TPHOLs 2008: In Emerging Trends of the 21st International Conference on Theorem Proving in Higher Order Logics (TPHOLs)*, 2008.

[Lea03] M. Leach. *RAILWAY Control Systems*. Institution of Railway Signal Engineers, Savoy Hill House, Savoy Hill, London, 2^{nd} edition, 2003.

[Lew01] M. Lewis. Railways in the Greek and Roman world. In *Early Railways. A Selection of Papers from the First International Early Railways Conference*, pages 8–19, 2001.

[McB02] C. McBride. Faking it, simulating dependent types in haskell. *Journal of Functional Programming*, 12(4-5):375–392, 2002.

[Men43] L. Menabrea. Sketch of the analytical engine invented by Charles Babbage, by lf menabrea, officer of the military engineers, with notes upon the memoir by the translator. *Taylor's Scientific Memoirs*, 3:666–731, 1843. Translator is A. Lovelace. Also in B. V. Bowden (Eds), Faster Then Thought, Pitman, London, 1953, pp 341 – 408. Online available from `http://www.fourmilab.ch/babbage/sketch.html`.

[Mer96] J. Mertens. Verifying the safety guaranteeing system at railway station Heerhugowaard. Master's thesis, University of Utrecht, 1996.

[MLS84] P. Martin-Löf and G. Sambin. *Intuitionistic type theory*, volume 17. Bibliopolis Naples, Italy, 1984.

[MMZ+01] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[MN95] O. Müller and T. Nipkow. Combining model checking and deduction for I/O-automata. In E. Brinksma, W. Cleaveland, K. Larsen, T. Margaria, and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1019 of *Lecture Notes in Computer Science*, pages 1–16. Springer Berlin / Heidelberg, 1995.

[MNR+12a] F. Moller, H. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. Combining event-based and state-based modeling for railway verification. Technical Report CS-12-02, Department of Computing, University of Surrey, 2012.

[MNR+12b] F. Moller, H. Nguyen, M. Roggenbach, S. Schneider, and H. Treharne. CSP——B modelling for railway verification: the double junction case study. Technical Report CS-12-03, Department of Computing, University of Surrey, 2012.

[Mon92] M. Monigel. Formal representation of track topologies by double vertex graphs. *Proceedings of Railcomp '92 held in Washington DC*, 2:359–370, 1992.

[Mor96] M. Morley. *Safety Assurance in Interlocking Design.* PhD thesis, Edinburgh University, 1996. ECS-LFCS-96-348.

[MW04] A. McEwan and J. Woodcock. A refinement based approach to calculating a fault-tolerant railway signal device. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 621–627. Springer Boston, 2004.

[Net05] Network Rail, 40 Melton Street, London, NW1 2EE. *SSI Applications Manual – Interlocking*, October 2005. SSI8003-10.

[Noc02] O. Nock. *Railway Signalling.* Institution of Railway Signal Engineers, Savoy Hill House, Savoy Hill, London, 2^{nd} edition, 2002.

[Nor07] U. Norell. *Towards a practical programming language based on dependent type theory.* PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[Nor09] U. Norell. Dependently typed programming in Agda. In P. Koopman, R. Plasmeijer, and D. Swierstra, editors, *Advanced Functional Programming*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer Berlin / Heidelberg, 2009.

[NPS90] B. Nordström, K. Petersson, and J. Smith. *Programming in Martin-Löf's Type Theory*, volume 7 of *International Series of Monographs on Computer Science.* Oxford University Press, 1990.

[Ogi04] T. Ogino. CyberRail. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 651–655. Springer Boston, 2004.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *Automated Deduction CADE-11*, volume 607 of *Lecture Notes in Computer Science*, pages 748–752. Springer Berlin / Heidelberg, 1992.

[PB04] M. Penicka and D. Bjrner. From railway resource planning to train operation. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 629–636. Springer Boston, 2004.

[Pel99] F. J. Pelletier. A brief history of natural deduction. *History and Philosophy of Logic*, 20(1):1–31, 1999.

[PS07] L. Paulson and K. Susanto. Source-level proof reconstruction for interactive theorem proving. In K. Schneider and J. Brandt, editors, *Theorem Proving in Higher Order Logics*, volume 4732 of *Lecture Notes in Computer Science*, pages 232–245. Springer Berlin / Heidelberg, 2007.

[PSB03] M. Penicka, A. Strupchanska, and D. Bjørner. Train maintenance routing. In G. Tarnai and E. Schnider, editors, *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*, Hungry, 2003. Technical University of Budapest.

[P07] M. Pnika. Formal approach to railway applications. In C. Jones, Z. Liu, and J. Woodcock, editors, *Formal Methods and Hybrid Real-Time Systems*, volume 4700 of *Lecture Notes in Computer Science*, pages 504–520. Springer Berlin / Heidelberg, 2007.

[Ran94] A. Ranta. *Type Theoretical Grammar*. Oxford University Press, 1994.

[RL12] A. Russo and L. Ladenberger. A formal approach to safety verification of railway signaling systems. In *Reliability and Maintainability Symposium (RAMS), 2012 Proceedings - Annual*, pages 1–4, January 2012.

[RN03] N. J. Robinson and G. Nikandros. Railway signalling design tools – supporting control table designers. In J. Tillin, editor, *Signs of the Times for Train Control, ASPELT'03*, pages 79–86, Savoy Hill House, Savoy Hill, London, 2003. The Institution of Railway Signalling Engineers.

[ROTS04] W. Reif, F. Ortmeier, A. Thums, and G. Schellhorn. Integrated formal methods for safety analysis of train systems. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 637–642. Springer Boston, 2004.

[RV02] A. Riazanov and A. Voronkov. The design and implementation of VAMPIRE. *AI communications*, 15(2):91–110, 2002.

[Sab04] D. Sabatier. Reusing formal models. In R. Jacquart, editor, *Building the Information Society*, volume 156 of *IFIP International Federation for Information Processing*, pages 613–619. Springer Boston, 2004.

[SBRG12] D. Sabatier, L. Burdy, A. Requet, and J. Gury. Formal proofs for the NYCT line 7 (flushing) modernization project. In J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, and E. Riccobene, editors, *Abstract State Machines, Alloy, B, VDM, and*

Z, volume 7316 of *Lecture Notes in Computer Science*, pages 369–372. Springer Berlin / Heidelberg, 2012.

[Sch02] S. Schulz. E – A brainiac theorem prover. *AI Commun.*, 15:111–126, August 2002.

[Sch04] S. Schulz. System description: E 0.81. In D. Basin and M. Rusinowitch, editors, *Automated Reasoning*, volume 3097 of *Lecture Notes in Computer Science*, pages 223–228. Springer Berlin / Heidelberg, 2004.

[Set09] A. Setzer. Interactive programs in Agda. Presentation at Agda Implementers Meeting X, Göteborg, September 2009. `http://www.cs.swan.ac.uk/~csetzer/slides/goeteborg2009AgdaIntensiveMeeting.pdf`.

[Sha87] F. R. Shapiro. Etymology of the computer bug: History and folklore. *American Speech*, 62(4):376–378, 1987.

[Smi40] F. Smith. Accident Returns: Extract for Accident at Howden on 7th August 1840. Board of Trade, Her Majesty's Stationary Office, 1840.

[SPB03] A. Strupchanska, M. Penicka, and D. Bjørner. Railway staff rostering. In G. Tarnai and E. Schnider, editors, *FORMS'2003: Symposium on Formal Methods for Railway Operation and Control Systems*, Hungry, 2003. Technical University of Budapest.

[Sti01] C. Stirling. *Modal and Temporal Properties of Processes*. Springer-Verlag, New York, 2001.

[Stu09] A. Stump. Proof checking technology for satisfiability modulo theories. In *Proceedings of the International Workshop on Logical Frameworks and Metalanguages: Theory and Practice (LFMTP 2008)*, volume 228 of *Electronic Notes in Theoretical Computer Science*, pages 121–133. Elsevier, 2009.

[Sup99] P. Suppes. *Introduction to Logic*. Dover Pubns, 1999.

[Sut09] G. Sutcliffe. The TPTP problem library and associated infrastructure: The FOF and CNF parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, 2009.

[SWD97] A. Simpson, J. Woodcock, and J. Davies. The mechanical verification of solid state interlocking geographic data. *Proc. of Formal Methods Pacific (FMP97)*, 7:320, 1997.

[Tak09] M. Takeyama. Integrated? verification using Agda compiler. presentation at AIM X, 2009. Available from `http://wiki.portal.chalmers.se/agda/uploads/Main.AIMX/IV20090916.ppt`.

[TD88a] A. Troelstra and D. Dalen. *Constructivism in Mathematics: An introduction, Volume I*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1988.

[TD88b] A. Troelstra and D. Dalen. *Constructivism in Mathematics: An introduction, Volume II*, volume 123 of *Studies in Logic and the Foundations of Mathematics*. Elsevier Science, 1988.

[The04] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. Version 8.0.

[TRN02] D. Tombs, N. Robinson, and G. Nikandros. Signalling control table generation and verification. In *Proc. of Conference on Railway Engineering (CORE 2000), Railway Technical Society of Australasia*, 2002.

[Tur10] R. Turk. A modern back-end for a dependently typed language. Master's thesis, Institute for Logic, Language and Computation, Universiteit van Amsterdam, 2010.

[Ver00] K. N. Verma. Reflecting symbolic model checking in Coq. Master's thesis, Mémoire de DEA, DEA Programmation, Paris, September 2000.

[WBH+02] C. Weidenbach, U. Brahm, T. Hillenbrand, E. Keen, C. Theobald, and D. Topić. Spass version 2.0. In A. Voronkov, editor, *Automated Deduction (CADE-18)*, volume 2392 of *Lecture Notes in Computer Science*, pages 45–79. Springer Berlin / Heidelberg, 2002.

[Web06] T. Weber. Integrating a SAT solver with an LCF-style theorem prover. In *Proceedings of the Third Workshop on Pragmatics of Decision Procedures in Automated Reasoning (PDPAR 2005)*, volume 144 of *Electronic Notes in Theoretical Computer Science*, pages 67–78. Elsevier, 2006.

[Wik12] Wikipedia. Software bug. Available from `http://en.wikipedia.org/wiki/Software_bug`, 2012.

[Win02] K. Winter. Model checking railway interlocking systems. *Australian Computer Science Communications*, 24(1):303–310, 2002.

[WLBF09] J. Woodcock, P. G. Larsen, J. Bicarregui, and J. Fitzgerald. Formal methods: Practice and experience. *ACM Comput. Surv.*, 41(4):19:1–19:36, October 2009.

[Woo12] P. Woodbridge. Locking Frame Testing – Notes for the 21st Century S & T Engineer regarding testing 19th Century interlocking technology. Web Page: `http://www.signalbox.org/branches/pw/index.htm`, July 2012.

[WR03] K. Winter and N. Robinson. Modelling large railway interlockings and model checking small ones. In *Proceedings of the 26th Australasian computer science conference-Volume 16*, pages 309–316. Australian Computer Society, Inc., 2003.

[Wyn49] G. Wynne. Accident Returns: Extract for the Accident at Blue Pits on 17th March 1849. Board of Trade, Her Majesty's Stationary Office, 1849.

[YL97] S. Yu and Z. Luo. Implementing a model checker for LEGO. In J. Fitzgerald, C. B. Jones, and P. Lucas, editors, *FME '97: Industrial Applications and Strengthened Foundations of Formal Methods*, volume 1313 of *Lecture Notes in Computer Science*, pages 442–458. Springer Berlin / Heidelberg, 1997.

[Yol67] W. Yolland. Accident Returns: Extract for the Accident at Walton Junction on 29th June 1867. Board of Trade, Her Majesty's Stationary Office, 1867.

The Index