Defining Trace Semantics for CSP-Agda

Bashar Igried[1] and Anton Setzer[2]

1,2 Swansea University, Dept. of Computer Science
Swansea, Wales, UK
1 bashar.igried@yahoo.com
2 a.g.setzer@swansea.ac.uk

Abstract

This article is based on the library CSP-Agda, which represents the process algebra CSP coinductively in the interactive theorem prover Agda. The intended application area of CSP-Agda is the proof of properties of safety critical systems (especially the railway domain). In CSP-Agda, CSP processes have been extended to monadic form, allowing the design of processes in a more modular way. In this article we extend the trace semantics of CSP to this monadic setting. We implement this semantics, together with the corresponding refinement and equality relation, formally in CSP-Agda. In order to demonstrate the proof capabilities of CSP-Agda, we prove in CSP-Agda selected algebraic laws of CSP based on the trace semantics. The examples covered in this article are the laws of refinement, commutativity of interleaving and parallel, and the monad laws for the monadic extension of CSP. Further proofs of algebraic laws will be available in the repository of CSP-Agda.

1998 ACM Subject Classification D.2.4 Software/Program Verification—Formal methods, D.3.1 Formal Definitions and Theory—Semantics, F.3.2 Semantics of Programming Languages, F.4.3 Formal Languages—Operations on languages

Keywords and phrases Agda, CSP, Coalgebras, Coinductive Data Types, Dependent Type Theory, IO-Monad, Induction-Recursion, Interactive Program, Monad, Monadic Programming, Process Algebras, Sized Types, Universes, Trace Semantics

Digital Object Identifier 10.4230/LIPIcs.TYPES.2016.23

1 Introduction

Communicating Sequential Processes (CSP) [18, 25] is a formal specification language which was developed in order to modelling concurrent systems through their communications. It was developed by Hoare in 1978 [18]. It is a member of the family of process algebras. Process algebras are one of the most important concepts for describing concurrent behaviours of programs.

The starting point of this work was the modelling by the first author of processes of the European Railway Train Management System (ERTMS) in CSP. Having expertise in modelling railway interlocking systems in Agda (PhD project by Kanso [21, 22]), we thought that an interesting step forward would be to model CSP in Agda. A first step towards this project was the development of the library CSP-Agda [20, 19]. CSP-Agda represents CSP processes coinductively and in monadic form. The purpose of this article is to introduce CSP trace semantics in Agda, and carry out examples of proofs in CSP-Agda.

In CSP-Agda we developed a monadic extension of CSP, which is based on Moggi's IO monad [24]. This IO monad ($\mathsf{IO}\ A$) is the currently the main construct for representing interactive programs in pure functional programming. An element of ($\mathsf{IO}\ A$) is an interactive program, which may or may not terminate, and, if it terminates, returns an element of type

A. The monad provides the bind construct for combining elements of $(\mathsf{IO}\ A)$: It composes a $p : \mathsf{IO}\ A$ with a function $f : A \to \mathsf{IO}\ B$ to form an element of $\mathsf{IO}\ B$. The program is executed by first running p. If p terminates with result a, one continues running $(f\ a)$. This allows to write sequences of operations in a way which looks similar to sequences of assignments in imperative style programming languages.

Hancock and the second author [16, 15, 17] have developed a version of the IO monad in dependent type theory, which we call the HS-monad. The HS-monad reduces the IO monad to coinductively defined types. An element of $(\mathsf{IO}\ A)$ is either a terminated program, or it is node of a non-well-founded tree having as label a command to be executed, and as branching degree the set of responses the real world gives in response to this command. The HS-Monad has been extensively used for writing interactive programs in the paper [3] on object-based programming in Agda.

In [20], we modelled processes in a similar way as a monad and developed the library CSP-Agda. A CSP-Agda process can either terminate, returning a result. Or it can be a tree branching over external and internal choices, where for each such choice a continuing process is given. So instead of forming processes by using high level operators, as it is usually done in process algebras, our processes are given by these atomic one step operations. The high level operators are defined operations on these processes. CSP-Agda introduces a new concept to process algebra, namely that of a monadic processes. A monadic process may run or terminate. If it terminates, it returns a value. This facilitates the combination of processes in a modular way. Processes are defined coinductively, and therefore we can introduce processes directly corecursively without having to use the recursion combinator.

Abel, Pientka, Thibodeau and the second author have [4, 28] developed the notion of coinductive types as being defined by their elimination rules or observations. This notion has now been implemented in Agda. It turns out that classes and objects in object oriented programming are of similar nature: Classes are defined by their methods, and therefore given by their observations. The second author [27] has used this approach in order to develop the notion of objects in dependent type theory, which has been recently extended together with Abel and Adelsberger [3] to a library for objects in Agda with extensive examples including correctness proofs.

In CSP-Agda [20, 19] we made extensively use of the aforementioned representing of coinductive types by their elimination rules. Using a record type, we accessed directly for non-terminating processes the choice sets and corresponding subprocesses.

The goal of this paper is to extend CSP-Agda by adding (finite) trace semantics of CSP. This requires extending trace semantics to the monadic setting. We show how to prove selected algebraic laws of CSP in Agda using this semantics: the laws of refinement, commutativity of interleaving and parallel, and the monad laws for the monadic extension of CSP. Further proofs of algebraic laws will be available in the repository of CSP-Agda [19].

The **structure of this paper** is as follows: In Sect. 2, we review the process algebra CSP. In Sect. 3, we give a brief introduction into the type theoretic language of Agda. In Sect. 4, we review CSP-Agda, and introduce the CSP operators used in the examples of this paper (monadic bind, parallel and interleaving). In Sect. 5 we extend CSP-Agda by adding (finite) trace semantics of CSP. In Sect. 6 we prove selected algebraic laws of CSP processes. In Sect. 7, we will look at related work, give a short conclusion, and indicate future work to be done.

2 CSP

Process algebras were initiated in 1982 by Bergstra and Klop [7] in order to provide a formal semantics to concurrent systems. A "process" is a representation of the behaviour of a concurrent system. "Algebra" means that the system is dealt with in an algebraic and axiomatic way [6]. In this article we represent a process algebra in the interactive theorem prover Agda in order to prove properties of processes. The process algebra chosen is Communicating Sequential Processes (CSP). CSP [18, 25, 26] was developed by Hoare in 1978 [18].

Processes in CSP form a labelled transition system, where the one step transitions is written as

$$P \xrightarrow{\mu} Q \qquad \text{where } P, Q \text{ are processes and } \mu \text{ is an action,}$$

which means that process P can evolve to process Q by event μ. The event μ can be a label, the silent transition τ, or the termination event $\checkmark$. In case of the label $\checkmark$ Q will always be the specific process STOP. As an example, we give here the execution of the process $a \to b \to \mathsf{STOP}$:

$$(\ a \to b \to \mathsf{STOP}) \xrightarrow{a} (b \to \mathsf{STOP}) \xrightarrow{b} \mathsf{STOP}$$

The operational semantics of CSP defines processes as states, and defines the transition rules between the states using firing rule. In CSP-Agda [20, 19] we introduced the firing rules for CSP operators (taken from [26]), and modelled them in Agda. We followed the version of CSP used in [26, 25]. All rules (as well those in this paper) are taken from [26]. In the rules we follow the convention of [26] that a ranges over Label $\cup \{\checkmark\}$ and μ over Label $\cup \{\checkmark, \tau\}$.

In the following table, we list the constructs for forming CSP processes. Here Q represent CSP processes:

Q ::= STOP	STOP
| SKIP	SKIP
| prefix	$a \to Q$
| external choice	$Q \,\Box\, Q$
| internal choice	$Q \sqcap Q$
| hiding	$Q \setminus a$
| renaming	$Q[R]$
| parallel	$Q \,{}_X\!\|\!|_Y\, Q$
| interleaving	$Q \,\|\|\|\, Q$
| interrupt	$Q \,\triangle\, Q$
| composition	$Q \,⨾\, Q$

There are as well indexed versions of $\Box$, $\sqcap$, $\|$, $|||$. They are indexed over finite sets, and therefore can be reduced to the binary case.

3 Agda

In this chapter we introduce the main concepts of Agda [5, 8], a more extensive introduction can be found in [20].

Agda is based on dependent type theory. There are several levels of types in Agda, the lowest is for historic reasons called Set. Types in Agda are given as dependent function

types, and inductive types. In addition there exist record types (which are in the newer approach used for defining coinductive types) and a generalisation of inductive-recursive and inductive-inductive definitions. Inductive data type are dependent versions of algebraic data types as they occur in functional programming. Inductive data types are given as sets A together with constructors which are strictly positive in A. For instance the set of vectors of elements of A and of length n is given as

```
data Vec {A : Set} : ℕ → Set where
  []   : {n : ℕ} → Vec {A} zero
  _::_ : {n : ℕ} (a : A) (l : Vec {A} n) → Vec {A} (suc n)
```

Here $\{n : \mathbb{N}\}$ is an implicit argument. Implicit arguments are omitted, provided they can be uniquely determined by the type checker. We can make a hidden argument explicit by writing for instance $[]\ \{n\}$. $\_::\_$ is Agda's notation for mixfix symbols. the arguments of a mixfix operator are denoted by underscore ($\_$). $a :: l$ stands for $(\_::\_\ a\ l)$.

The above definition introduces a new type $\mathsf{Vec} : \{A : \mathsf{Set}\} \to \mathbb{N} \to \mathsf{Set}$, where $(\mathsf{Vec}\ \{A\}\ n)$ is a type of vectors of type A of length n. It has constructors $[]$ and $\_::\_$. The elements of $(\mathsf{Vec}\ \{A\}\ n)$ are those constructed from applying these constructors. Therefore we can define functions by case distinction on these constructors using pattern matching. The following defines the sum of elements of a vector of type $\mathbb{N}$:

```
sum : ∀ {n} → Vec {ℕ} n → ℕ
sum []       = 0
sum (n :: l) = n + sum l
```

Here we used the notation $\forall\ \{n\} \to \cdots$, which stands for $\{n : A\} \to \cdots$, where A (here $\mathbb{N}$) can be inferred by Agda. Nested patterns are allowed. The coverage checker checks completeness and the termination checker checks that the recursive calls follow a schema of extended primitive recursion.

In this paper we use the approach of defining coinductive types in Agda by their elimination rules as introduced in [4, 28]. The standard example is the set of streams:

```
record Stream (i : Size) : Set where
  coinductive
  field
    head : ℕ
    tail : {j : Size< i} → Stream j
```

If we first ignore the arguments Size, Size< we see that the type Stream is given as a record type in Agda. It is defined coinductively by its observations head, tail.

Elements of Stream are defined by copattern matching, i.e. by determining the result of applying head, tail to them. A simple (non-recursive) operation is the function cons for adding a new element in front of a stream:

```
cons : ∀ {i} → ℕ →  Stream i → Stream (↑ i)
head (cons n s) = n
tail (cons n s) = s
```

Without sizes, in recursive definitions only recursive calls to the function being defined are possible, with no restrictions on the arguments they are applied to. However, no functions can be applied to the recursive calls. This restriction on the recursive definitions is called the principle of guarded recursion [9] or primitive corecursion. As an example we give the pointwise addition of two streams:

```
_+s_ : ∀ {i} → Stream i → Stream i → Stream i
head (s +s s') = head s + head s'
tail (s +s s') = tail s +s tail s'
```

_+s_ makes a recursive call to tail s +s tail s'. Note that s, s' are arguments of _+s_, so we can apply tail to them freely.

Without the guarded recursion restriction, one could define non productive definitions, e.g. define tail $(f\ x)$ = tail $(f\ x)$. However, the guardedness restriction makes it difficult to define streams in a modular way, since we cannot in a corecursive call refer to other functions for forming streams at all, although many operations will not cause problems. Therefore Abel has introduced sized types [1, 2] in the context of coinductive types.

Sizes are essentially ordinals (without infinite branching one can think of them as natural numbers), however there is an additional infinite size ∞. We have as operations for forming sizes the the infinite size ∞, the successor operation on sizes $\uparrow$, and have the type of sizes less then i denoted by Size< i.

For ordinal sizes $i \neq \infty$, a stream s : Stream i allows up to i times of applications of tail. The true streams is the set Stream ∞ and s : Stream ∞ allows arbitrary many applications of tail. When defining an element f : $(i$: Size$)$ → $A\ i \rightarrow$ Stream i by corecursion, tail $(s\ (f\ i\ a))$ $\{j\}$ must be an element of size $\geq j$ which can refer to a recursive call $(f\ j\ a')$, and we can apply functions to it as long as the resulting size is $\geq j$. Elimination on the recursive call is prevented, since we don't have access to any size $< j$. However, we can apply size preserving and size increasing functions to the recursive call. This guarantees that streams are productive. We have ∞ : Size< ∞, so a recursive definition of elements of Stream ∞ can refer to itself.

Agda offers let and where expressions in order to declare a local definition. In comparison, where expressions allow a pattern matching or recursive function, whereas pattern matching and recursive functions are not allowed in let expressions. In Agda the let expressions can be represented as follows:

```
let
 a1 : A1
 a1 = s1
 a2 : A2
 a2 = s2
   ...
 an : An
 an = sn
in t
```

In the above definition, we use let expressions in order to introduce new local constants:

$$
\begin{array}{l}
a_1 \ : \ \mathsf{A}_1 \ s.t. \ a_1 \ = \ s_1, \\
a_2 \ : \ \mathsf{A}_2 \ s.t. \ a_2 \ = \ s_2, \\
\quad \ldots \\
an \ : \ \mathsf{An} \ s.t. \ an \ = \ s_n
\end{array}
$$

4 The Library CSP-Agda

In this section we repeat the main definition of processes in CSP-Agda from [20]. The reader might consult that paper for a more detailed motivation of the definitions in CSP-Agda.

4.1 Representing CSP Processes in Agda

As outlined before, we represent processes in Agda in a monadic way. A process P : Process A is either a terminating process (terminate a), which has return value a : A, or it is process (node P) which progresses. Here P : Process+ A, where (Process+ A) is the type of progressing processes. A progressing process can proceed at any time with labelled transitions (external choices), silent transitions (internal choices), or ✓-events (termination). After a ✓-event, the process becomes deadlocked, so there is no need to determine the process after that event. We will however add a return value a : A to ✓-events.

Elements of (Process+ A) are therefore determined by

(1) an index set E of external choices, and for each external choice e the Label (Lab e) and the next process (PE e);
(2) an index set of internal choices I, and for each internal choice i the next process (PI i); and
(3) an index set of termination choices T corresponding to ✓-events, and for each termination choice t the return value PT t : A.

In addition we add in CSP-Agda a type (Process∞ A). This makes it easy to define processes by guarded recursion, when the right hand side is defined directly and without having to define all 7 components of (Process+ A). Furthermore, in order to display processes, we add eliminators Str+ and Str∞ to (Process+ A) and (Process∞ A), respectively. They return a string representing the process. In case of (Process∞ A), this cannot be reduced to the string component of (Process+ A): in order to do this one would need a smaller size, which we don't have in general for arbitrary sizes.

We model the sets of external, internal, and termination choices as elements of an inductive-recursively defined universe Choice. Elements c of Choice are codes for finite sets, and (ChoiceSet c) is the set it denotes. In addition we define a string (choice2Str c) representing c, and a function choice2Enum which computes from c a list of all choices. This will be used to print a list of choices in the simulator for CSP processes.

We require as well that the set of return values are elements of Choice. This allows us to print the result returned when a process terminates. However, for the return types it is not needed that they are finite sets. So one use a different universe for the return values of processes, which would allow for instance the set of natural numbers as a return type.

The resulting code for processes in Agda is as follows:

```
mutual
  record  Process∞ (i : Size) (c : Choice) : Set where
    coinductive
```

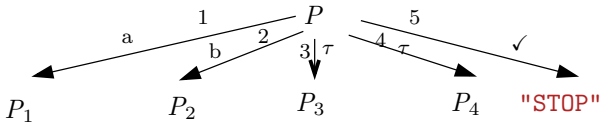
```
    field
      forcep : {j : Size< i} → Process j c
      Str∞   : String

data Process (i : Size) (c : Choice) : Set where
  terminate : ChoiceSet c   → Process i c
  node      : Process+ i c  → Process i c

record Process+ (i : Size) (c : Choice) : Set where
  constructor process+
  coinductive
  field
    E    : Choice
    Lab  : ChoiceSet E → Label
    PE   : ChoiceSet E → Process∞ i c
    I    : Choice
    PI   : ChoiceSet I → Process∞ i c
    T    : Choice
    PT   : ChoiceSet T → ChoiceSet c
    Str+ : String
```

An example of a process is as follows:

P = node Q : Process String where
E Q = code for $\{1,2\}$ I Q = code for $\{3,4\}$
T Q = code for $\{5\}$
Lab Q 1 = a Lab Q 2 = b PE Q 1 = P_1
PE Q 2 = P_2 PI Q 3 = P_3 PI Q 4 = P_4
PT Q 5 = "STOP"



The universe of choices is given by a set Choice of codes for choice sets, and a function ChoiceSet, which maps a code to the choice set it denotes. Universes were introduced by Martin-Löf (e.g. [23]) in order to formulate the notion of a type consisting of types. Universes are defined in Agda by an inductive-recursive definition [11, 10, 12, 13]: we define inductively the set of codes in the universe while recursively defining the decoding function.

We give here the code expressing that Choice is closed under fin, ⊎', ×', subset', Σ' and namedElements. (Fin n) is the finite set having n elements. The type (NamedElements l) is essentially (Fin (length l)). The function choice2Str will for elements of this set print the nth element of l, giving them more meaningful names. subset A f is the set of $a : A$ such that (f a) is true. The definition of ChoiceSet is as follows:

```
mutual
data Choice : Set where
  fin     : ℕ → Choice
  _⊎'_    : Choice → Choice → Choice
```

```
  _×'_ : Choice → Choice → Choice
  namedElements : List String → Choice
  subset' : (E : Choice) → (ChoiceSet E → Bool)
    → Choice
  Σ'      : (E : Choice) → (ChoiceSet E → Choice)
    → Choice

ChoiceSet : Choice → Set
ChoiceSet (fin n)     = Fin n
ChoiceSet (s ⊎' t)    = ChoiceSet s ⊎ ChoiceSet t
ChoiceSet (E ×' F)    = ChoiceSet E × ChoiceSet F
ChoiceSet (namedElements s) = NamedElements s
ChoiceSet (subset' E f) = subset (ChoiceSet E) f
ChoiceSet (Σ' A B)  =   Σ[ x ∈ ChoiceSet A ] ChoiceSet (B x)

choice2Str : {c : Choice} → ChoiceSet c → String
choice2Str {fin n} m = showℕ (toℕ m)
 ...

choice2Enum : (c : Choice) → List (ChoiceSet c)
choice2Enum (fin n) = fin2Option0 n
 ...
```

4.2 Monadic Bind, Parallel Operator and Interleaving

We introduce the three operators, for which we will prove algebraic properties in this paper: monadic bind, the parallel operator and interleaving.

4.2.1 Monadic Bind

In our article [20] we introduced the monadic bind. In Sect. 6.2 we will prove the monadic laws and therefore we briefly repeat the definition of the monadic bind. A more extensive motivation can be found in [20].

The monadic bind ($P \gg\!=\!+\ Q$) allows to compose two processes P and Q while allowing the second process depend on the return type c_0 of P. So Q has an an extra argument of the return type (ChoiceSet c_0). The transitions of ($P \gg\!=\!+\ Q$) are as follows: It follows first external and internal choices of P. If P is the terminated process with return type a, the process continues as process ($Q\ a$). A special case is a termination event in P with return value a. Following the operational semantics of CSP in this case ($P \gg\!=\!+\ Q$) has an internal choice (i.e. a τ-transition) to process $Q\ a$. Therefore ($P \gg\!=\!+\ Q$) has two possible internal choice events, namely internal choices of P and termination events of P. It has no termination events.

The full definition of monadic bind is as follows:

```
_≫=Str_ : {c₀ : Choice} → String
  → (ChoiceSet c₀ → String)          → String
s ≫=Str f = s  ++s ";" ++s choice2Str2Str f
```

```
mutual
  _≫=∞_ : {i : Size} → {c₀ c₁ : Choice}
        → Process∞ i c₀
        → (ChoiceSet c₀ → Process∞ i c₁)
        → Process∞ i c₁
  forcep (P ≫=∞ Q)    = forcep P ≫= Q
  Str∞ (P ≫=∞ Q)      = Str∞ P ≫=Str (Str∞ ∘ Q)

  _≫=_ : {i : Size} → {c₀ c₁ : Choice}
        → Process i c₀
        → (ChoiceSet c₀ → Process∞ (↑ i) c₁)
        → Process i c₁
  node      P ≫= Q = node (P ≫=+ Q)
  terminate x ≫= Q = forcep (Q x)

  _≫=+_ : {i : Size} → {c₀ c₁ : Choice}
        → Process+ i c₀
        → (ChoiceSet c₀ → Process∞ i c₁)
        → Process+ i c₁
  E   (P ≫=+ Q)          = E P
  Lab (P ≫=+ Q)          = Lab P
  PE  (P ≫=+ Q) c        = PE P c ≫=∞ Q
  I   (P ≫=+ Q)          = I P ⊎' T P
  PI  (P ≫=+ Q) (inj₁ c) = PI P c ≫=∞ Q
  PI  (P ≫=+ Q) (inj₂ c) = Q (PT P c)
  T   (P ≫=+ Q)          = ∅'
  PT  (P ≫=+ Q) ()
  Str+ (P ≫=+ Q) = Str+ P ≫=Str (Str∞ ∘ Q)
```

As in [20], when defining operators on Process we introduce simultaneously operators on the three categories of processes Process∞, Process, and Process+. We use qualifiers ∞, p, + attached to the operators for refer to the 3 categories of processes, respectively. We often omit p, and omitted it in case of _≫=_. We have as well a string forming operation indicated by Str, and sometimes a result type forming operation indicated by Res. For some binary operators we need versions where the arguments are from different categories of processes, in which case we add two qualifiers to the operators. We will only present the main cases of the operators. Especially, we will usually omit the functions involving Process∞, which in most cases follow the same pattern as the definition of _≫=∞_ above, and the string forming operation, which is easy to define. The full code can be found at [19].

4.2.2 The parallel operator

The parallel operator enforces two processes to work together and interact through synchronous events. For each of the two processes sets of labels A, B are given. For labels which are not in the intersection, both processes can execute independently. For processes in interaction, both processes need to synchronise on that event. The transition rules for the parallel operator are as follows (having two conclusion of a rule is an abbreviation for two rules having the same premises: one deriving the first and one deriving the second conclusion):

$$\frac{P \xrightarrow{a} \bar{P} \qquad Q \xrightarrow{a} \bar{Q}}{P \;{}_A\|_B\; Q \xrightarrow{a} \bar{P} \;{}_A\|_B\; \bar{Q}} [\; a \in A^{\checkmark} \cap B^{\checkmark} \;] \qquad \frac{P \xrightarrow{\mu} \bar{P}}{P \;{}_A\|_B\; Q \xrightarrow{\mu} \bar{P} \;{}_A\|_B\; Q} [\; \mu \in ((A \cup \tau) \backslash B) \;]$$

$$Q \;{}_B\|_A\; P \xrightarrow{\mu} Q \;{}_B\|_A\; \bar{P}$$

In CSP-Agda we define parallel operator as follow: We assume functions $A\ B : \mathsf{Label} \to \mathsf{Bool}$ which determines the labels for P, Q, for which these processes can act independently. Let $\neg\ \mathsf{b}$ be Boolean negation. The external choices of $P\ [\ A\ ]||+[\ B\ ]\ Q$ are:

- The external choices of $c : \mathsf{E}\ P$, for which the label in P is not in A, i.e. such that $\neg\mathsf{b}\ (A\ (\mathsf{Lab}\ P\ c)) = \mathsf{true}$. For such c the label for this external choice is the label of P for choice c, and the process obtained following this transition is the parallel construct applied to $\mathsf{PE}\ P\ c$ and Q.
- The external choices of $c : \mathsf{E}\ Q$, for which the label in Q is not in B, with similar definitions of the label and next process obtained.
- The combined external choices for P and Q, i.e. pairs $(e_1\ ,\ e_2)$ s.t. $e_1 : \mathsf{E}\ P$ and $e_2 : \mathsf{E}\ Q$, and s.t. their labels are equal, and the labels are in sets A and in B respectively, i.e.

$$\mathsf{Lab}\ P\ e_1 ==\mathsf{l}\ \mathsf{Lab}\ Q\ e_2 \wedge A\ (\mathsf{Lab}\ P\ e_1) \wedge B\ (\mathsf{Lab}\ Q\ e_2)\ =\ \mathsf{true}$$

Here $\_==\mathsf{l}\_$ is Boolean valued equality on Labels, and $\_\wedge\_$ is Boolean valued conjunction. The label of this external choice is the label of P (which is w.r.t. $\_==\mathsf{l}\_$ equal to the corresponding label of Q), and the process obtained when following this external choice is the parallel construct applied to the result of following the external choices of P and Q.

Furthermore

- The internal choices are the internal choices of P and Q, and the process obtained when following those transitions is obtained by following the corresponding transition in process P or Q, respectively.
- A termination event can happen only if both processes have a termination event. If they terminate with results a and b, then the parallel combination terminates with result $(a\ ,,\ b)$. Therefore the result type of the parallel construct is the product of the result type of the first and second process.

In order to define the above we use the subset' constructor of Choice which has equality rule

$$\mathsf{ChoiceSet}\ (\mathsf{subset'}\ E\ f) = \mathsf{subset}\ (\mathsf{ChoiceSet}\ E\ )\ f$$

Here subset $a\ f$ is the set of pairs sub $a\ b$ such that $a : A$ and $b : \mathsf{T}\ (f\ a)$, i.e. it is essentially the set $\{a : A \mid f\ a = \mathsf{true}\}$. We have $\mathsf{T} : \mathsf{Bool} \to \mathsf{Set}$, such that $\mathsf{T}\ \mathsf{true}$ is provable and $\mathsf{T}\ \mathsf{false}$ is empty, i.e. not provable.

The definition of the parallel operator in CSP-Agda for Process+ is as follows:

$$\begin{aligned}
&_[_]||+[_]_ : \{i : \mathsf{Size}\} \to \{c_0\ c_1 : \mathsf{Choice}\} \\
&\quad \to \mathsf{Process+}\ i\ c_0 \\
&\quad \to (A\ \ B : \mathsf{Label} \to \mathsf{Bool}) \\
&\quad \to \mathsf{Process+}\ i\ c_1
\end{aligned}$$

```
    → Process+ i (c₀ ×' c₁)
E   (P [ A ]||+[ B ] Q)  =  subset' (E P) ((¬b ∘ A) ∘ (Lab P)) ⊎'
                           subset' (E Q) ((¬b ∘ B) ∘ (Lab Q)) ⊎'
                           subset' (E P ×' E Q)
                              (λ {(e₁ „ e₂)
                                 → Lab P e₁ ==I Lab Q e₂ ∧ A (Lab P e₁) ∧ B (Lab Q e₂)})
Lab (P [ A ]||+[ B ] Q) (inj₁ (inj₁ (sub c p)))  =  Lab P c
Lab (P [ A ]||+[ B ] Q) (inj₁ (inj₂ (sub c p)))  =  Lab Q c
Lab (P [ A ]||+[ B ] Q) (inj₂ (sub (c₀ „ c₁) p))  =  Lab P c₀
PE  (P [ A ]||+[ B ] Q) (inj₁ (inj₁ (sub c p)))  =  PE P c  [ A ]||∞+[ B ] Q
PE  (P [ A ]||+[ B ] Q) (inj₁ (inj₂ (sub c p)))  =  P       [ A ]||+∞[ B ] PE Q c
PE  (P [ A ]||+[ B ] Q) (inj₂ (sub (c₀ „ c₁) p))  =  PE P c₀ [ A ]||∞[ B ] PE Q c₁
I   (P [ A ]||+[ B ] Q)  =  I P ⊎' I Q
PI  (P [ A ]||+[ B ] Q) (inj₁ c)  =  PI P c  [ A ]||∞+[ B ] Q
PI  (P [ A ]||+[ B ] Q) (inj₂ c)  =  P       [ A ]||+∞[ B ] PI Q c
T   (P [ A ]||+[ B ] Q)  =  T P ×' T Q
PT  (P [ A ]||+[ B ] Q) (c₀ „ c₁)  =  PT P c₀ „ PT Q c₁
Str+ (P [ A ]||+[ B ] Q)  =  Str+ P  [ A ]||Str[ B ] Str+ Q
```

When defining the parallel construct for elements of Process, we need to deal with the case one of the processes is the terminated process. In this case one continues as the other other process, until it has terminated. However, in case of P having terminated, only labels in the set $B \setminus A$ are allowed. We can therefore equate, if P has terminated, P [A]||+[B] Q with $Q \restriction (B \setminus A)$. However, we record the result obtained by P, and therefore apply fmap to Q in order to add the result of P to the result of the restriction of Q, when it terminates. Here fmap f P is the process obtained from P by applying f to any termination results.

The definition of the parallel operator for Process is therefore as follows:

```
_[_]||[_]_ : {i : Size} → {c₀ c₁ : Choice}
   → Process i c₀
   → (A  B : Label → Bool)
   → Process i c₁
   → Process i (c₀ ×' c₁)
node P  [ A ]||[ B ] node Q  =  node (P [ A ]||+[ B ] Q)
terminate a [ A ]||[ B ] Q  =  fmap (λ b → (a „ b))  (Q ↾ (B \ A))
P  [ A ]||[ B ] terminate b  =  fmap (λ a → (a „ b))  (P ↾ (A \ B))
```

4.2.3 The interleaving operator

The interleaving operator executes the external and internal choices of its arguments P and Q completely independently of each other. The CSP rules are as follows:

$$\frac{P \xrightarrow{\checkmark} \bar{P} \qquad Q \xrightarrow{\checkmark} \bar{Q}}{P \;|||\; Q \xrightarrow{\checkmark} \bar{P} \;|||\; \bar{Q}} \qquad \frac{P \xrightarrow{\mu} \bar{P}}{P \;|||\; Q \xrightarrow{\mu} \bar{P} \;|||\; Q} \; \mu \neq \checkmark$$

$$Q \;|||\; P \xrightarrow{\mu} Q \;|||\; \bar{P}$$

The definition of the two main cases in CSP-Agda is as follows:

```
mutual
  _|||_ : {i : Size} → {c₀ c₁ : Choice} → Process i c₀
        → Process i c₁ → Process i (c₀ ×' c₁)
  node P  ||| node Q  =  node  (P |||++ Q)
  terminate a |||  Q  =  fmap  (λ b → (a „ b)) Q
  P |||  terminate b  =  fmap  (λ a → (a „ b)) P

  _|||++_ : {i : Size} → {c₀ c₁ : Choice}
    → Process+ i c₀ → Process+ i c₁
    → Process+ i (c₀ ×' c₁)
  E   (P |||++ Q)          =  E P ⊎' E Q
  Lab (P |||++ Q) (inj₁ c) =  Lab P c
  Lab (P |||++ Q) (inj₂ c) =  Lab Q c
  PE  (P |||++ Q) (inj₁ c) =  PE P c |||∞+  Q
  PE  (P |||++ Q) (inj₂ c) =  P |||+∞ PE Q c
  I   (P |||++ Q)          =  I P ⊎' I Q
  PI  (P |||++ Q) (inj₁ c) =  PI P c |||∞+  Q
  PI  (P |||++ Q) (inj₂ c) =  P |||+∞ PI Q c
  T   (P |||++ Q)          =  T P ×' T Q
  PT  (P |||++ Q) (c „ c₁) =  PT P c „ PT Q c₁
  Str+ (P |||++ Q)         =  Str+ P |||Str Str+ Q
```

When processes P and Q haven't terminated, then P ||| Q will not terminate. The external choices are the external choices of P and Q. The labels are the labels from the processes P and Q, and we continue recursively with the interleaving combination. The internal choices are defined similarly. A termination event can happen only if both processes have a termination event.

If one process terminates but the other not, the rules of CSP express that one continues as the other other process, until it has terminated. We can therefore equate, if P has terminated, P ||| Q with Q. However, we record the result obtained by P, and therefore apply fmap to Q in order to add the result of P to the result of Q when it terminates. If both processes terminate with results a and b, then the interleaving combination terminates with result $(a \mathbin{„} b)$.

5 Defining Trace Semantics for CSP-Agda

In CSP traces of a process are the sequences of actions or labels of external choices, a process can perform. Since the process in CSP are non-deterministic, a process can follow different traces during its execution. The trace semantics of a process is the set of its traces.

Since in CSP-Agda processes are monadic, we need to record, in case after following a trace we obtain a terminated process, the result returned by the process following this trace. So we add a possible element of the result set to the trace. We can use for the set of possible elements the set (Maybe (ChoiceSet c)). Here the type (Maybe A) has elements (just a) for $a : A$, denoting defined elements, and an undefined element nothing. So (just a) denotes that the process has terminated with result a, whereas nothing means that it hasn't terminated (or more precisely been determined as terminated).

Taking this together, we obtain that traces are given by a list of labels and an element of (Maybe (ChoiceSet c)). We define the set of traces (Tr l m P) as a predicate which

determines for a process the lists of labels l and elements m : Maybe (ChoiceSet c), which form a trace. We define as well traces (Tr+ l m P) and (Tr∞ l m P) for processes in Process+ and Process∞, respectively.

In the trace semantics of CSP a process which has a termination event has two traces, the empty list, and the list consisting of a ✓-event. In order to be consistent with CSP, we will add therefore in case of a termination event or terminated process two traces: the empty list together with possible return value nothing, and with possible return value (just a) for the return value a.

For an element of Process+ ∞ c we obtain the following traces:

- The empty trace without termination is a trace of any process, and we denote the proof by empty.
- If a process P has external choice x, then from every trace for the result of following this choice, which consisting of a list of labels l and a possible result *tick*, we obtain a trace of P consisting of the result of adding in front of l the label of that external choice, and of the same possible result *tick*. The resulting proof will be denoted by (extc l *tick* x tr).
- Internal choices are ignored in traces. Therefore If a process P has an internal choice x, every trace of the result of following this process is a trace of P. The proof is denoted by (intc l *tick* x tr)
- If a process has a termination event x with return value t, then the empty trace with termination choice (just t) is a trace of process, having proof terc x.

The corresponding definition for the processes of Process+ is as follows:

```
data Tr+ {c : Choice} : (l : List Label) → Maybe (ChoiceSet c) → (P : Process+ ∞ c)
  → Set where
empty : {P : Process+ ∞ c} → Tr+ [] nothing P
extc  : {P : Process+ ∞ c} → (l : List Label) → (tick : Maybe (ChoiceSet c))
  → (x : ChoiceSet (E P)) → Tr∞ l tick (PE P x) → Tr+ (Lab P x :: l) tick P
intc  : {P : Process+ ∞ c} → (l : List Label) → (tick : Maybe (ChoiceSet c))
  → (x : ChoiceSet (I P)) → Tr∞ l tick (PI P x) → Tr+ l tick P
terc  : {P : Process+ ∞ c} → (x : ChoiceSet (T P)) → Tr+ [] (just (PT P x)) P
```

In case of Process we need to consider the termination events:

- The terminated process has two traces, namely the empty list of labels [] with termination event nothing, and the same list but with termination event (just x), where x is the return value.
- The traces of a non-terminated process are the traces of the corresponding element of Process+.

We obtain the following definition of the traces of Process:

```
data Tr {c : Choice} : (l : List Label) → Maybe (ChoiceSet c) → (P : Process ∞ c)
  → Set where
ter   : (x : ChoiceSet c) → Tr [] (just x) (terminate x)
empty : (x : ChoiceSet c) → Tr [] nothing (terminate x)
tnode : {l : List Label} → {x : Maybe (ChoiceSet c)} → {P : Process+ ∞ c}
  → Tr+ {c} l x P → Tr l x (node P)
```

Finally the traces for Process∞ are just the traces of the underlying Process:

```
record Tr∞ {c : Choice} (l : List Label) (tick : Maybe (ChoiceSet c))
    (P : Process∞ ∞ c) : Set  where
coinductive
field
    forcet : Tr  l tick (forcep P)
```

In CSP, a process P refines a process Q, written $P \sqsubseteq+ Q$ if and only if any observable behaviour of Q is an observable behaviour of P, i.e. if $traces(Q) \subseteq traces(P)$:

```
_⊑_ :  {c : Choice} (P : Process ∞ c) (Q : Process ∞ c) → Set
_⊑_  {c} P Q  = (l : List Label) → (m : Maybe (ChoiceSet c)) → Tr l m Q → Tr l m P
```

Two processes P, Q are equal w.r.t. trace semantics, written $P \equiv Q$, if they refine each other, i.e. if $traces(P) = traces(Q)$:

```
_≡_ : {c₀ : Choice} → (P Q : Process ∞ c₀) → Set
P ≡ Q =  P ⊑ Q ×  Q ⊑ P
```

6 Proof of the Algebraic Laws

Trace equivalence gives rise to algebraic laws for individual operators, and also concerning the relationships between different operators. Numerous laws are concerned with general algebraic properties such as commutativity and associativity of operators; these properties allow a process to be composed in any order, the identification of zeros and units for specific operators, where these properties allow process descriptions to be simplified, and idempotence. Many laws are concerned with the relationships between different operators, for example the expansion of a parallel into a prefix choice process. We will present examples of how to prove algebraic laws of CSP in Agda using this semantics. The examples covered in this article are commutativity of interleaving and parallel, and the monad laws for the monadic extension of CSP. Further examples will be available in the repository of CSP-Agda.

6.1 Proof of the Laws of Refinement

The refinement relation is reflexive, anti-symmetric and transitive, i.e. fulfils the following laws:

$P \sqsubseteq P$
$P_0 \sqsubseteq P_1 \wedge P_1 \sqsubseteq P_0 \Rightarrow P_0 = P_1$
$P_0 \sqsubseteq P_1 \wedge P_1 \sqsubseteq P_2 \Rightarrow P_0 \sqsubseteq P_2$

These laws are a direct consequence of the fact that $P \sqsubseteq Q$ means essentially $traces(Q) \subseteq traces(P)$ and $P \equiv Q$ means $traces(P) = traces(Q)$:

```
refl⊑ : {c : Choice} (P : Process ∞ c) → P ⊑ P
```

```
refl⊑ : {c} P l m x = x

antiSym⊑ : {c₀ : Choice} → (P Q : Process ∞ c₀) → P ⊑ Q → Q ⊑ P → P ≡ Q
antiSym⊑ P Q PQ QP = PQ , QP

trans⊑ : {c : Choice}(P : Process ∞ c)(Q : Process ∞ c)(R : Process ∞ c)
  → P ⊑ Q → Q ⊑ R → P ⊑ R
trans⊑ {c} P Q R PQ QR l m tr = PQ l m (QR l m tr)
```

6.2 Proof of the Monadic Laws

We defined processes in a monadic way, and will in this section prove the monad laws for processes.

In functional programming, a monad is given by a functor M together with morphisms $\gg\!= : \mathsf{M}\ A \to (A \to \mathsf{M}\ B) \to \mathsf{M}\ B$ and $\mathsf{return} : A \to \mathsf{M}\ A$ such that the following laws hold:

$$\begin{array}{lcl} \mathsf{return}\ a \gg\!= f & = & f\ a \\ p \gg\!= \mathsf{return} & = & p \\ (p \gg\!= f) \gg\!= g & = & p \gg\!= (\lambda\, x.f\ x \gg\!= g) \end{array}$$

For each monadic law we have to prove 2 directions, ("$\sqsubseteq$" and "$\sqsupseteq$"). Furthermore the laws need to be shown for Process+, Process and Process∞. We will present only one direction and one version of the processes for each law. Since proofs of $\_\equiv\_$ just follow from the left to right and right to left refinement, we will present this proof only for the first monadic law.

The proof of the first monadic law is trivial since terminate$a \gg\!= P$ is definitionally to P:

```
monadLaw₁ : {c₀ c₁ : Choice} (a : ChoiceSet c₀)(P : ChoiceSet c₀ → Process ∞ c₁)
  → (terminate a ≫= P) ⊑ P a
monadLaw₁ a P l m q = q

≡monadLaw₁ : {c₀ c₁ : Choice} (a : ChoiceSet c₀)(P : ChoiceSet c₀ → Process ∞ c₁)
  → (P a) ≡ (terminate a ≫= P)
≡monadLaw₁ {c₀} {c₁} a P = (monadLaw₁ a P) , (monadLaw₁r a P)
```

In case of the second monadic law the proof is by induction over the proofs of traces for $P \gg\!=+ \mathsf{terminate}$, which immediately turn into traces of P:

```
monadLaw₂+ : {c₀ : Choice} (P : Process+ ∞ c₀)→ (P ≫=+ terminate) ⊑+ P
monadLaw₂+ P .[] .nothing empty = empty
monadLaw₂+ P .(Lab P x :: l) m (extc l .m x x₁) = extc l m x (monadLaw₂∞ (PE P x) l m x₁)
monadLaw₂+ P l m (intc .l .m x x₁) = intc l m (inj₁ x) (monadLaw₂∞ (PI P x) l m x₁)
monadLaw₂+ P .[] .(just (PT P x))(terc x) = intc [] (just (PT P x)) (inj₂ x) (lemTrTerBind P x)
```

In third monadic law the proof is by induction over the proofs of traces for $(P \gg\!=+ (Q \gg\!=+ R))$. In most cases the proof of traces carry over after applying the induction hypothesis. One special case if the first process P has a termination event, which results in an internal choice to $Q\ x \gg\!= R$ on both sides. In this case the traces are essentially

the same, but only after applying forcet. We use here an operation

```
monadPT+ P Q R y l m tr
```

which is modulo an application of forcet equal to tr. There are no immediate termination events, and therefore no proofs of traces of the form (terc x). We use efq (ex falsum quodlibet), which constructs from an element of the empty set an element of any set, for dealing with this case. The resulting proof is as follows:

```
monadLaw3+ : {c0 c1 c2 : Choice} (P : Process+ ∞ c0)
  (Q : ChoiceSet c0 → Process ∞ c1)
  (R : ChoiceSet c1 → Process ∞ c2)
→ ((P >>=+ Q) >>=+ R) ⊑+ (P >>=+ (λ x → Q x >>= R))
monadLaw3+ P Q R .[] .nothing empty = empty
monadLaw3+ P Q R .(Lab P x :: l) m (extc l .m x x1) =
  extc l m x (monadLaw∞ P Q R l x m x1)
monadLaw3+ P Q R l m (intc .l .m (inj1 x) x1) =
  intc l m (inj1 (inj1 x))(monadLaw3∞ (PI P x) Q R l m x1)
monadLaw3+ P Q R l m (intc .l .m (inj2 y) x1) =
  intc l m (inj1 (inj2 y))(monadPT+ P Q R y l m x1)
monadLaw3+ P Q R .[] .(just (PT (P >>=+ (λ x → Q x >>= R)) x)) (terc x) = efq x
```

6.3 Proof Commutativity Laws for the Interleaving operator

The interleaving combination $(P \;|||\; Q)$ executes each component completely independent of the other, until termination. Traces of the interleaving combination $P \;|||\; Q$ will, therefore, appear as interleavings of traces of the two component, and therefore it is easy to see that $(P \;|||\; Q)$ and $(Q \;|||\; P)$ are trace equivalent.

However, because of the monadic setting, for most algebraic laws the return types of the left and right hand side of an equation are different. Assume the return types of P and Q are c_0 and c_1, respectively. Then for instance the return type of $(P \;|||\; Q)$ is $(c_0 \times' c_1)$ whereas the return type of $(Q \;|||\; P)$ is $(c_1 \times' c_0)$. Therefore the algebraic laws hold only modulo applying an adjustment of the return types using the operation fmap, which applies a function to the return types.

Once we have taken this into account, a proof of commutativity of _|||_ is obtained by exchanging the external/internal/termination choices, which means swapping inj_1 and inj_2. Here inj_1 refers to choices in the first and inj_2 to choices in the second process. Using this we can proof commutativity by induction as follows:

As an example we give the main case proof the commutative laws of interleaving (swap× swaps the two sides of a product):

```
mutual
S|||+ : {c0 c1 : Choice} (P : Process+ ∞ c0) (Q : Process+ ∞ c1)
  → (P |||++ Q) ⊑+ (fmap+ swap× (Q |||++ P))
S|||+ P Q .[] .nothing empty = empty
S|||+ P Q .(Lab Q x :: l) m (extc l .m (inj1 x) q)  = extc l m (inj2 x)  (S|||+∞  P (PE Q x) l m q)
S|||+ P Q .(Lab P x :: l) m (extc l .m (inj2 x) q)  = extc l m (inj1 x)  (S|||∞+  (PE P x) Q l m q)
```

```
S|||+ P Q l m (intc .l .m (inj₁ x) q)                = intc l m (inj₂ x)  (S|||+∞  P (PI Q x) l m q)
S|||+ P Q l m (intc .l .m (inj₂ x) q)                = intc l m (inj₁ x)  (S|||∞+  (PI P x) Q l m q)
S|||+ P Q .[] .(just (PT P x „ PT Q y)) (terc (y „ x)) = terc (x „ y)
```

```
≡S|||+ : {c₀ c₁ : Choice} (P : Process+ ∞ c₀) (Q : Process+ ∞ c₁)
  → (P |||++ Q) ≡+ (fmap+ swap× (Q |||++ P))
≡S|||+ P Q = (S|||+ P Q) , (S|||+R P Q)
```

6.4 Proof Commutativity Laws for the Parallel operator

Most cases in the proof of the commutativity of _[_]||+[_]_ are similar to the proof of commutativity _|||_ by swapping inj_1 and inj_2 and using induction. The only tricky case is when we have two processes synchronizing, resulting in both processes following choices having the same labels. This case involves a proof that two choices result for the two processes result in the same label and that both labels are in the synchronized sets. We obtain in this case from a proof that we have a trace a proof of the Boolean conjunct

```
Lab Q x ==I Lab P x₁ ∧ B (Lab X x) ∧ A (Lab P x₁)
```

which we need to transform into a Boolean conjunct

```
Lab P x₁ ==I Lab Q x ∧ A (Lab X x₁) ∧ B (Lab P x)
```

We use operations ∧BoolEliml, ∧BoolElimr which extract from a Boolean conjunct its two component, e.g.

```
∧BoolEliml : (a b : Bool) → T (a ∧ b) → T a
```

an operation introducing proofs of a Boolean conjunction

```
∧BoolIntro : (a b : Bool) → T a → T b → T (a ∧ b)
```

a proof sym of symmetry of the Boolean equality _==I_ on labels, and the transfer lemma

```
transf : (Q : Label → Set) → (l l' : Label ) → T (l ==I l' ) → Q l → Q l'
```

We now take the proof of the conjunction apart into its three components, apply the proof of symmetry to the equality proof and recombine them. Finally we need to carry out a transfer to replace the first label (Lab P x_1) in the trace by (Lab Q x), which are known to be equal. The resulting proof is as follows:

```
mutual
S[]+ : {c₀ c₁ : Choice} (P : Process+ ∞ c₀) (A  B : Label → Bool) (Q : Process+ ∞ c₁)
  → (P [ A ]||+[ B ] Q) ⊑+ fmap+ swap× (Q [ B ]||+[ A ] P)

S[]+    P A B Q .[] .nothing empty = empty
S[]+    P A B Q .(Lab Q a :: l) m (extc l .m (inj₁ (inj₁ (sub a x))) x₁) =
      extc l m (inj₁ (inj₂ (sub a x))) (S[]+∞ P A B (PE Q a) l m x₁)
S[]+    P A B Q .(Lab P a :: l) m (extc l .m (inj₁ (inj₂ (sub a x))) x₁) =
      extc l m (inj₁ (inj₁ (sub a x))) (S[]∞+ (PE P a) A B Q l m x₁)
```

```
S[]+  P A B Q .(Lab Q x :: l) m (extc l .m (inj₂ (sub (x „ x₁) x₂)) x₃) =
      let
          lxlx₁ : T (Lab Q x ==I Lab P x₁)
          lxlx₁  =  ∧BoolElimI  (Lab Q x ==I Lab P x₁)
                                (B (Lab Q x) ∧ A (Lab P x₁)) x₂

          BQx : T (B (Lab Q x))
          BQx  =  ∧BoolElimI  (B (Lab Q x)) (A (Lab P x₁))
                              (∧BoolElimr  (Lab Q x ==I Lab P x₁)
                                           (B (Lab Q x) ∧ A (Lab P x₁)) x₂)

          APx₁ : T (A (Lab P x₁))
          APx₁  =  ∧BoolElimr  (B (Lab Q x)) (A (Lab P x₁))
                               (∧BoolElimr  (Lab Q x ==I Lab P x₁)
                                            (B (Lab Q x) ∧ A (Lab P x₁)) x₂)

          lx₁lx : T (Lab P x₁ ==I Lab Q x)
          lx₁lx  =  sym (Lab Q x) (Lab P x₁) lxlx₁

          x₂' : T ((Lab P x₁ ==I Lab Q x) ∧ A (Lab P x₁) ∧ B (Lab Q x))
          x₂'  =  ∧BoolIntro  (Lab P x₁ ==I Lab Q x)
                              (A (Lab P x₁) ∧ B (Lab Q x))
                              lx₁lx
                              (∧BoolIntro (A (Lab P x₁)) (B (Lab Q x)) APx₁ BQx)

          auxpr : Tr+ (Lab P x₁ :: l) m (P [ A ]||+[ B ] Q)
          auxpr = extc  l m  (inj₂ (sub (x₁ „ x) x₂'))
                             (S[]∞∞ (PE P x₁) A B (PE Q x) l m x₃)

  in transf  (λ l' →  Tr+ (l' :: l) m (P [ A ]||+[ B ] Q))
             (Lab P x₁) (Lab Q x) lx₁lx auxpr

S[]+ P A B Q l m (intc .l .m (inj₁ x) x₁) = intc l m (inj₂ x) (S[]+∞ P A B (PI Q x) l m x₁)
S[]+ P A B Q l m (intc .l .m (inj₂ y) x₁) = intc l m (inj₁ y) (S[]∞+ (PI P y) A B Q l m x₁)
S[]+ P A B Q .[] .(just (PT P x₁ „ PT Q x)) (terc (x „ x₁)) = terc (x₁ „ x)

≡[||]+ : {c₀ c₁ : Choice} (P : Process+ ∞ c₀)(A  B : Label → Bool)(Q : Process+ ∞ c₁)
  → (P [ A ]||+[ B ] Q) ≡+  (fmap+ swap× ((Q [ B ]||+[ A ] P)))
≡[||]+  P A B Q = (S[]+ P A B Q) , (S[]+r P A B Q)
```

7 Related Work and Conclusion

Related Work. A detailed report on related work, which we don't want to repeat here, can be found in our previous paper [20].

Conclusion. The aims of this research is to give the type theoretic interactive theorem prover Agda the ability to model and verify concurrent programs by representing the process

algebra CSP in monadic form. We implement trace semantics of CSP in Agda, together with the corresponding refinement and equality relation, formally in CSP-Agda. In order to demonstrate the proof capabilities of CSP-Agda, we prove in CSP-Agda selected algebraic laws of CSP based on the trace semantics. In our approach we define processes coinductively and the trace semantic inductively.

Future Work. We are currently working on defining the other semantics of CSP in Agda. Since those semantics are rather long, proofs of algebraic properties are much more involved. The first author has developed elements of the European Rail Traffic Management System ERTMS [14] in CSP, and one goal is to implement those processes in CSP-Agda and prove safety properties. For larger case studies automated theorem proving techniques will be used. Here we can build on Kanso's PhD thesis [21] (see as well [22]), in which he verified real world railway interlocking systems in Agda. Verifying larger examples might require to upgrade the integration of SAT solvers into Agda2, which has been developed by Kanso [21], to the current version of Agda.

One goal is to integrate the CSP model checker FDR2 into Agda. One ambitious goal is to write prototypes of programs, e.g. of some elements of the ERTMS, in Agda and make them directly executable in Agda. This uses the unique feature of Agda of being both a theorem prover and a dependently typed programming language. So in Agda there is no distinction between proofs and programs, between data types and propositions, and therefore the prototype can be implemented and and verified in the same language, without the need to translate between two different languages.

Acknowledgements. This research was supported by the CORCON FP7 Marie Curie International Research Project, PIRSES-GA-2013-612638; COMPUTAL FP7 Marie Curie International Research Project, PIRSES-GA-2011-294962; and by CA COST Action CA15123 European research network on types for programming and verification (EUTYPES). The PhD project by B. Igried is supported by Hashemite University (FFNF150).

References

1 A. Abel. *A Polymorphic Lambda-Calculus with Sized Higher-Order Types*. PhD thesis, Ludwig-Maximilians-Universität München, 2006. Available from http://www2.tcs.ifi.lmu.de/∼abel/publications.html.

2 A. Abel. Compositional coinduction with sized types. In I. Hasuo, editor, *Coalgebraic Methods in Computer Science*, pages 5–10. Springer, 2016.

3 A. Abel, S. Adelsberger, and A. Setzer. Interactive Programming in Agda – Objects and Graphical User Interfaces. To appear in Jour. Functional Programming, preprint available at http://www.cs.swan.ac.uk/∼csetzer/articles/ooAgda.pdf, 2016.

4 A. Abel, B. Pientka, D. Thibodeau, and A. Setzer. Copatterns: Programming infinite structures by observations. In R. Giacobazzi and R. Cousot, editors, *Proceedings of POPL'13*, pages 27–38. ACM, 2013.

5 Agda Community. The Agda Wiki . http://wiki.portal.chalmers.se/agda/pmwiki.php, 2015.

6 J. Baeten, D. A. van Beek, and J. Rooda. Process algebra. *Handbook of Dynamic System Modeling*, pages 19–1, 2007.

7 J. A. Bergstra and J. W. Klop. Fixed point semantics in process algebras. CWI technical report, Stichting Mathematisch Centrum. Informatica-IW 206/82, 1982.

8 A. Bove, P. Dybjer, and U. Norell. A brief overview of Agda — a functional language with dependent types. In *Proceedings of TPHOLs '09*, pages 73–78. Springer, 2009.

9 T. Coquand. Infinite objects in type theory. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, volume 806 of *LNCS*, pages 62–78. Springer, 1994.

10 P. Dybjer. Inductive sets and families in Martin-Löf's type theory and their set-theoretic semantics. In G. Huet and G. Plotkin, editors, *Logical frameworks*, pages 280 – 306. Cambridge University Press, 1991.

11 P. Dybjer. Universes and a general notion of simultaneous inductive-recursive definition in type theory. In B. Nordström, K. Petersson, and G. Plotkin, editors, *Proceedings of the 1992 workshop on types for proofs and programs, Båstad*, June 1992. Available from http://www.lfcs.inf.ed.ac.uk/research/types-bra/proc/proc92.ps.gz.

12 P. Dybjer. A general formulation of simultaneous inductive-recursive definitions in type theory. *Journal of Symbolic Logic*, 65(2):525 – 549, June 2000.

13 P. Dybjer and A. Setzer. Induction-recursion and initial algebras. *Annals of Pure and Applied Logic*, 124:1 – 47, 2003.

14 ERTMS. The European Rail Traffic Mangement System. http://www.ertms.net/, 2013.

15 P. Hancock and A. Setzer. The IO monad in dependent type theory. In *Electronic proceedings of the workshop on dependent types in programming, Göteborg, 27-28 March 1999*, 1999. Available via http://www.md.chalmers.se/Cs/Research/Semantics /APPSEM/dtp99.html.

16 P. Hancock and A. Setzer. Interactive programs in dependent type theory. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic*, LNCS, Vol. 1862, pages 317 – 331, 2000.

17 P. Hancock and A. Setzer. Specifying interactions with dependent types. In *Workshop on subtyping and dependent types in programming, Portugal,7 July 2000*, 2000. Electronic proceedings, available via http://www-sop.inria.fr/oasis/DTP00/Proceedings/proceedings.html.

18 C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, Aug. 1978.

19 B. Igried and A. Setzer. CSP-Agda. Agda library. Available at http://www.cs.swan.ac.uk/∼csetzer/ software/agda2/cspagda/, 2016.

20 B. Igried and A. Setzer. Programming with monadic CSP-style processes in dependent type theory. In *Proceedings of the 1st International Workshop on Type-Driven Development*, TyDe 2016, pages 28–38, New York, NY, USA, 2016. ACM.

21 K. Kanso. *Agda as a Platform for the Development of Verified Railway Interlocking Systems*. PhD thesis, Dept. of Computer Science, Swansea University, Swansea, UK, August 2012. Available from http://www.swan.ac.uk/csetzer/articlesFromOthers /index.html and http://cs.swan.ac.uk/∼cskarim/files/.

22 K. Kanso and A. Setzer. A light-weight integration of automated and interactive theorem proving. *Mathematical Structures in Computer Science*, FirstView:1–25, 12 November 2014.

23 P. Martin-Löf. *Intuitionistic type theory.* Bibliopolis, Naples, 1984.

24 E. Moggi. Notions of computation and monads. *Information and Computation*, 93(1):55 – 92, 1991.

25 A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall, 1997.

26 S. Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley, 1st edition, 1999.

27 A. Setzer. Object-oriented programming in dependent type theory. In *Conference Proceedings of TFP 2006*, 2006. Available from http://www.cs.nott.ac.uk/∼nhn/TFP2006/TFP2006-Programme.html and http://www.cs.swan.ac.uk/∼csetzer/index.html.

28 A. Setzer, A. Abel, B. Pientka, and D. Thibodeau. Unnesting of copatterns. In G. Dowek, editor, *Rewriting and Typed Lambda Calculi*, volume 8560 of *LNCS*, pages 31–45. Springer, 2014.