Combining Automated and Interactive Theorem Proving in Agda

Anton Setzer
(Joint work with Karim Kanso)

May 21, 2010

Basics of Agda

- The core of Agda is a very simple language.
- Functional programming language based on dependent types.
- Mainly used as an interactive theorem prover.
- Compiled version exists, prototype of a dependently typed programming language.

Algebraic Data Types

- Agda has infinitely many type levels, called

$$\mathrm{Set} \subseteq \mathrm{Set1} \subseteq \mathrm{Set2} \subseteq \cdots$$

- Algebraic data types can be introduced by determining their strictly positive constructors, e.g.

$$\begin{array}{l} \text{data } \mathbb{N} : \text{Set where} \\ \quad \begin{array}{lcl} \text{zero} & : & \mathbb{N} \\ \text{suc} & : & \mathbb{N} \to \mathbb{N} \end{array} \end{array}$$

Pattern Matching

- Once a set is introduced in this way functions can be defined
  - using pattern matching
  - recursively, as long as termination is accepted by the termination checker.
- Example

$$
\begin{array}{lll}
\mathrm{double} : \mathbb{N} \to \mathbb{N} & & \\
\mathrm{double}\,\mathrm{zero} & = & \mathrm{zero} \\
\mathrm{double}\,(\mathrm{suc}\ n) & = & \mathrm{suc}\,(\mathrm{suc}\,(\mathrm{double}\ n))
\end{array}
$$

Mixfix Symbols

- Agda allows mixfix symbols, with positions denoted by _ e.g.

$$
\begin{array}{lllll}
\_+\_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
n & + & \mathrm{zero} & = & n \\
n & + & \mathrm{suc}\ m & = & \mathrm{suc}\ (n + m)
\end{array}
$$

- We replace suc by $\_+1$, use builtin $\mathbb{N}$ which allows 0 and obtain

$$
\begin{array}{lllll}
\_+\_ : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N} \\
n & + & 0 & = & n \\
n & + & (m + 1) & = & (n + m) + 1
\end{array}
$$

- It supports as well the use of Unicode symbols.
- This allows to write code which looks very close to mathematical code.

Dependent Types

Assume we have defined the type of matrices $\mathrm{Mat}\ n\ m$ depending on dimensions n and m:

$$\mathrm{Mat} : \mathbb{N} \to \mathbb{N} \to \mathrm{Set}$$

Then the type of matrix multiplication is

$$\begin{aligned}\mathrm{matmult} : {} & (n\ m\ k : \mathbb{N}) \\ & \to \mathrm{Mat}\ n\ m \\ & \to \mathrm{Mat}\ m\ k \\ & \to \mathrm{Mat}\ n\ k\end{aligned}$$

Dependent Algebraic Data Types

We can define the type of n-vectors (or n-tuples) based on a set X:
($\{n : \mathbb{N}\}$ denotes a **hidden argument**)

$$
\begin{array}{l}
\mathrm{data\ Vector}(X : \mathrm{Set}) : \mathbb{N} \rightarrow \mathrm{Set\ where} \\
\quad \begin{array}{lcl}
[] & : & \mathrm{Vector}\ X\ \mathrm{zero} \\
\_ :: \_ & : & X \rightarrow \{n : \mathbb{N}\} \rightarrow \mathrm{Vector}\ X\ n \rightarrow \mathrm{Vector}\ X\ (n + 1)
\end{array}
\end{array}
$$

e.g. (using the builtin natural numbers)

$$
\begin{array}{lcl}
a & : & \mathrm{Vector}\ \mathbb{N}\ 3 \\
a & = & 0 :: 1 :: 2 :: []
\end{array}
$$

Logic in Agda

Logic in Agda (which is intuitionistic) is based on the principle of propositions as types:

- Propositions are elements of Set.
- Elements of propositions are proofs of this proposition.
- A proposition holds iff it has a proof.

Examples:

- The **true proposition**:

$$
\begin{array}{l}
\mathrm{data}\ \top : \mathrm{Set}\ \mathrm{where} \\
\quad \mathrm{triv} : \top
\end{array}
$$

$\bot$, $\wedge$, $\vee$

- The **false proposition**:

$$\text{data } \bot : \text{Set where}$$

Pattern matching on an empty data type (ex falsum quodlibet) is denoted as follows:

$$\begin{array}{l} f : \bot \to \mathbb{N} \\ f \; () \end{array}$$

- **Conjunction**:

$$\begin{array}{l} \_\wedge\_ \; (A \; B : \text{Set}) : \text{Set where} \\ \quad \text{and} : A \to B \to A \wedge B \end{array}$$

- **Disjunction**:

$$\begin{array}{l} \_\vee\_ \; (A \; B : \text{Set}) : \text{Set where} \\ \quad \begin{array}{lcl} \text{inl} & : & A \to A \vee B \\ \text{inr} & : & B \to A \vee B \end{array} \end{array}$$

$\rightarrow$, $\neg$, $\forall$, $\exists$

- **Implication**: $A \rightarrow B$ is the function type $A \rightarrow B$.
- **Negation**: $\neg A = A \rightarrow \bot$.
- **Universal quantification**: $\forall x : A.\varphi$ is given as

$$(x : A) \rightarrow \varphi$$

- **Existential quantification**:

$$\begin{array}{l} \mathrm{data}\ \exists\ (A : \mathrm{Set})\ (\varphi : A \rightarrow \mathrm{Set}) : \mathrm{Set}\ \ \mathrm{where} \\ \quad \mathrm{exists} : (x : A) \rightarrow (\varphi\ x) \rightarrow \exists\ A\ \varphi \end{array}$$

- **Example**:
  $\forall \epsilon > 0.\exists \delta > 0.\varphi(\epsilon, \delta)$ is written as

$$(\epsilon : \mathbb{Q}) \rightarrow \epsilon > 0 \rightarrow \exists\ \mathbb{Q}\ (\lambda \delta.\delta > 0 \wedge \varphi\ \epsilon\ \delta)$$

Decidable Prime Formulas

Booleans:

$$
\begin{array}{l}
\mathrm{data}\ \mathbb{B} : \mathrm{Set}\ \ \mathrm{where} \\
\quad \mathrm{tt} \quad : \quad \mathbb{B} \\
\quad \mathrm{ff} \quad : \quad \mathbb{B}
\end{array}
$$

Atom converts Booleans into the corresponding formula:

$$
\begin{array}{l}
\mathrm{Atom} : \mathbb{B} \to \mathrm{Set} \\
\mathrm{Atom} \quad \mathrm{tt} \quad = \quad \top \\
\mathrm{Atom} \quad \mathrm{ff} \quad = \quad \bot
\end{array}
$$

Main Idea

- Define a data type of codes for formulas in Agda:

$$\begin{array}{l}\mathrm{data\ For} : \mathrm{Set\ \ where} \\ \quad \ldots\end{array}$$

- Define what is meant by an environment, which e.g. assigns values to free variables, determines the state etc. We get

$$\mathrm{Env} : \mathrm{Set}$$

- Define a function $[\![\_]\!]$ which assigns to codes for formulas and environments the corresponding Agda formula:

$$[\![\_]\!] : \mathrm{For} \to \mathrm{Env} \to \mathrm{Set}$$

Main Idea

Define a check function, which checks whether a formula is universally true:

$$\mathrm{check} : \mathrm{For} \longrightarrow \mathbb{B}$$

Prove that check is correct:

$$\mathrm{correctCheck} : (\varphi : \mathrm{For}) \longrightarrow \mathrm{Atom}\ (\mathrm{check}\ \varphi) \longrightarrow (\xi : \mathrm{Env}) \longrightarrow [\![\, \varphi \,]\!]\ \xi$$

Implement in Agda a builtin version of check which calls an automated theorem proving tool. Declare check as a builtin:

$$\{-\#\ \mathrm{BUILTIN\ CHECK\ check}\ \#-\}$$

Now when check is called for a closed element of For, instead of the (inefficient) Agda code the automated theorem prover is called.

Usage

Assume an Agda formula ψ, e.g.

$$
\begin{aligned}
&\psi : \mathbb{B} \to \mathbb{B} \to \mathrm{Set} \\
&\psi\ b\ b' = (\mathrm{Atom}\ b \wedge \mathrm{Atom}\ b') \vee \neg(\mathrm{Atom}\ b) \vee \neg(\mathrm{Atom}\ b')
\end{aligned}
$$

Assume that ψ has a code $\lceil \psi \rceil$ in For, i.e.

$$
\begin{aligned}
&\lceil \psi \rceil : \mathrm{For} \\
&\lceil \psi \rceil = \cdots
\end{aligned}
$$

s.t.

$$[\![\lceil \psi \rceil]\!][x \mapsto b, y \mapsto b'] = \psi\ b\ b'$$

Usage

$[\![\ulcorner\psi\urcorner]\!][x \mapsto b, y \mapsto b'] = \psi\ b\ b'$

Then we can prove this formula (which we could prove by hand) as follows:

$$\begin{array}{l} \mathrm{theorem} : (b\ b' : \mathbb{B}) \to \psi\ b\ b' \\ \mathrm{theorem}\ b\ b' = \mathrm{correctCheck}\ \lceil\psi\rceil\ \mathrm{triv}\ [x \mapsto b, y \mapsto b'] \end{array}$$

Type checking

$$\mathrm{triv} : \mathrm{Atom}\ (\mathrm{check}\ \lceil\psi\rceil)$$

will require that

$$\mathrm{check}\ \lceil\psi\rceil$$

evaluates to tt.

This evaluation will activate the automated theorem proving tool.

Note that in the example above we obtain

$$\mathrm{theorem} : (b\ b' : \mathbb{B}) \to (\mathrm{Atom}\ b \land \mathrm{Atom}\ b') \lor \neg(\mathrm{Atom}\ b) \lor \neg(\mathrm{Atom}\ b')$$

Interleaving Interactive and Automated Theorem Proving

This allows to combine both theorem proving techniques:

$$
\begin{array}{c}
\text{Interactive Theorem Proving} \\
\downarrow \\
\text{Automated Theorem Proving} \\
\downarrow \\
\text{Interactive Theorem Proving} \\
\downarrow \\
\text{Automated Theorem Proving} \\
\downarrow \\
\dots
\end{array}
$$

Simplicity of check

The function check will defined in such a way that

- The definition is simple.
  - When using a builtin function, we need to check that the function fulfils the equations.
  - So we need to implement in Agda the verification that when using check its Agda definition is correct.
- The correctness proof is simple, so that it can be given in Agda.
- Efficiency is not a concern since its usage will be replaced by a call to an efficient automated theorem prover.

Security Concerns

An initial idea was to define a flexible builtin in Agda, which automatically calls a user-defined Haskell function.

Problem:

- Then one could write Agda code, which during type checking calls an arbitrary Haskell function.
- Such a function might erase your hard disk.

Solution:

- To define a new builtin needs to require some modification of the Agda type checking program.
- Users should be aware that if programming is involved there might be a security problem.
- They won't expect this from a proof code to be type checked.

For

$$
\begin{array}{l}
\mathrm{data\ For : Set\ where} \\
\quad \begin{array}{lcl}
\mathrm{const} & : & \mathbb{B} \longrightarrow \mathrm{For} \\
\mathrm{x} & : & \mathbb{N} \longrightarrow \mathrm{For} \\
\_\wedge\mathrm{for}\_ & : & \mathrm{For} \longrightarrow \mathrm{For} \longrightarrow \mathrm{For} \\
\_\vee\mathrm{for}\_ & : & \mathrm{For} \longrightarrow \mathrm{For} \longrightarrow \mathrm{For} \\
\neg\mathrm{for} & : & \mathrm{For} \longrightarrow \mathrm{For}
\end{array}
\end{array}
$$

$\mathrm{check0}$ checks whether the formula holds if all variables are instantiated with tt:

$$
\begin{array}{llcl}
\multicolumn{4}{l}{\mathrm{check0} : \mathrm{For} \longrightarrow \mathbb{B}} \\
\mathrm{check0} & (\mathrm{const}\ b) & = & b \\
\mathrm{check0} & (\mathrm{x}\ n) & = & \mathrm{tt} \\
\mathrm{check0} & \left(\varphi \begin{array}{c} \wedge\mathrm{for} \\ \vee\mathrm{for} \end{array} \psi\right) & = & \mathrm{check0}\ \varphi \begin{array}{c} \wedge\mathbb{B} \\ \vee\mathbb{B} \end{array} \mathrm{check0}\ \psi \\
\mathrm{check0} & (\neg\mathrm{for}\ \varphi) & = & \neg\mathbb{B}\ (\mathrm{check0}\ \varphi)
\end{array}
$$

instantiate-

instantiate- φ b

- instantiates in φ variable $\mathrm{x}\ 0$ by b
- replaces $\mathrm{x}\ (n+1)$ by $\mathrm{x}\ n$

$$
\begin{array}{lllll}
\multicolumn{5}{l}{\text{instantiate-} : \text{For} \longrightarrow \mathbb{B} \longrightarrow \text{For}} \\
\text{instantiate-} & (\text{const } b) & b' & = & \text{const } b \\
\text{instantiate-} & (\mathrm{x}\ 0) & b' & = & \text{const } b' \\
\text{instantiate-} & (\mathrm{x}\ (n+1)) & b' & = & \mathrm{x}\ n \\
\text{instantiate-} & (\varphi \begin{array}{l} \wedge\text{for} \\ \vee\text{for} \end{array} \psi) & b' & = & \text{instantiate-}\ \varphi\ b' \\
 & & & & \begin{array}{l} \wedge\text{for} \\ \vee\text{for} \end{array} \ \text{instantiate-}\ \psi\ b' \\
\text{instantiate-} & (\neg\text{for}\ \varphi) & b' & = & \neg\text{for}\ (\text{instantiate-}\ \varphi\ b')
\end{array}
$$

check1

check1 φ n checks whether φ is universally true if

- variables $(\mathrm{x}\ 0) \cdots (\mathrm{x}\ (n-1))$ are arbitrary,
- other variables are instantiated by tt.

$$
\begin{array}{llll}
\multicolumn{4}{l}{\mathrm{check1} : \mathrm{For} \longrightarrow \mathbb{N} \longrightarrow \mathbb{B}} \\
\mathrm{check1}\ \ \varphi & 0 & = & \mathrm{check0}\ \varphi \\
\mathrm{check1}\ \ \varphi & (n+1) & = & \mathrm{check1}\ (\mathrm{instantiate}\text{-}\ \varphi\ \mathrm{tt})\ n \\
 & & & \wedge\mathbb{B}\ \mathrm{check1}\ (\mathrm{instantiate}\text{-}\ \varphi\ \mathrm{ff})\ n
\end{array}
$$

maxVar

maxVar returns

$$\max\{n+1 \mid (\mathrm{x}\ n) \text{ occurs in } \varphi\}$$

$$\begin{array}{lllcl}
\mathrm{maxVar} : \mathrm{For} \to \mathbb{N} & & & & \\
\mathrm{maxVar} & (\mathrm{const}\ b) & & = & 0 \\
\mathrm{maxVar} & (\mathrm{x}\ n) & & = & n+1 \\
\mathrm{maxVar} & (\varphi\ \begin{array}{l}\wedge\mathrm{for}\\ \vee\mathrm{for}\end{array}\ \psi) & & = & \max\ (\mathrm{maxVar}\ \varphi)\ (\mathrm{maxVar}\ \psi) \\
\mathrm{maxVar} & (\neg\mathrm{for}\ \varphi) & & = & \mathrm{maxVar}\ \varphi
\end{array}$$

Now we define check:

$$\begin{array}{l}
\mathrm{check} : \mathrm{For} \to \mathbb{B} \\
\mathrm{check}\ \varphi = \mathrm{check1}\ \varphi\ (\mathrm{maxVar}\ \varphi)
\end{array}$$

Nondependent Types

- Until now the code was kept minimal, and didn't require dependent types.
- check depends on all of this code.
- When defining the builtin function all this codes needs to be reflected into Haskell.
  - Possible because no dependent types were used.
- The code in the following needs not to be translated into Haskell code.
  - We will use dependent types, and will no longer be minimalistic.

$[\![\varphi]\!]$

Environments are given here as elements of *Vector* $\mathbb{B}$ n for some n.

- For $i < n$, variable $\mathrm{x}\ i$ is instantiated by the i element of this vector,
- For $i \geq n$, variable $\mathrm{x}\ i$ is instantiated by tt.

$$
\begin{array}{llllll}
\multicolumn{6}{l}{[\![\_]\!] : \mathrm{For} \to \{n : \mathbb{N}\} \to \mathrm{Vector}\ \mathbb{B}\ n \to \mathrm{Set}} \\
[\![& \mathrm{const}\ b &]\!] & \vec{b} & = & \mathrm{Atom}\ b \\
[\![& \mathrm{x}\ n &]\!] & [] & = & \mathrm{Atom}\ \mathrm{tt} \\
[\![& \mathrm{x}\ 0 &]\!] & (b :: \vec{b}) & = & \mathrm{Atom}\ b \\
[\![& \mathrm{x}\ (n+1) &]\!] & (b :: \vec{b}) & = & [\![\mathrm{x}\ n]\!]\ \vec{b} \\
[\![& \varphi \begin{array}{c} \wedge\mathrm{for} \\ \vee\mathrm{for} \end{array} \psi &]\!] & \vec{b} & = & [\![\varphi]\!]\ \vec{b} \begin{array}{c} \wedge \\ \vee \end{array} [\![\psi]\!]\ \vec{b} \\
[\![& \neg\mathrm{for}\ \varphi &]\!] & \vec{b} & = & \neg\ ([\![\varphi]\!]\ \vec{b})
\end{array}
$$

$[\![\, \varphi \,]\!]\mathrm{b}$

We have

$$[\![\, \mathrm{x}\ 0\ \ \wedge\mathrm{for}\ \ \mathrm{x}\ 1 \,]\!]\ (b :: b' :: [\,]) = \mathrm{Atom}\ b \wedge \mathrm{Atom} b'$$

We define as well $[\![\, \varphi \,]\!]\mathrm{b}$ s.t.

$$[\![\, \mathrm{x}\ 0\ \ \wedge\mathrm{for}\ \ \mathrm{x}\ 1 \,]\!]\mathrm{b}\ (b :: b' :: [\,]) = b\ \ \wedge\mathbb{B}\ \ b'$$

$$\begin{array}{lllll}
\multicolumn{5}{l}{[\![\, _ \,]\!]\mathrm{b} : \mathrm{For} \longrightarrow \{n : \mathbb{N}\} \longrightarrow \mathrm{Vector}\ \mathbb{B}\ n \longrightarrow \mathbb{B}} \\
[\![& \mathrm{const}\ b &]\!]\mathrm{b} & \vec{b} & = & b \\
[\![& \mathrm{x}\ n &]\!]\mathrm{b} & [\,] & = & \mathrm{tt} \\
[\![& \mathrm{x}\ 0 &]\!]\mathrm{b} & (b :: \vec{b}) & = & b \\
[\![& \mathrm{x}\ (n+1) &]\!]\mathrm{b} & (b :: \vec{b}) & = & [\![\, \mathrm{x}\ n \,]\!]\mathrm{b}\ \vec{b} \\
[\![& \varphi \begin{array}{c}\wedge\mathrm{for} \\ \vee\mathrm{for}\end{array} \psi &]\!]\mathrm{b} & \vec{b} & = & [\![\, \varphi \,]\!]\mathrm{b}\ \vec{b} \begin{array}{c}\wedge\mathbb{B} \\ \vee\mathbb{B}\end{array} [\![\, \psi \,]\!]\mathrm{b}\ \vec{b} \\
[\![& \neg\mathrm{for}\ \varphi &]\!]\mathrm{b} & \vec{b} & = & \neg\mathbb{B}\ \ ([\![\, \varphi \,]\!]\mathrm{b}\ \vec{b})
\end{array}$$

$[\![\,\varphi\,]\!]'$

We define $[\![\,\varphi\,]\!]'$ s.t.

$$[\![\,\mathrm{x}\;0\;\;\wedge\mathrm{for}\;\;\mathrm{x}\;1\,]\!]'\,(b :: b' :: [\,]) = \mathrm{Atom}\,(b\;\;\wedge\mathbb{B}\;\;b')$$

$$\begin{aligned}
&[\![\,_\,]\!]' : \mathrm{For} \longrightarrow \{n : \mathbb{N}\} \longrightarrow \mathrm{Vector}\;\mathbb{B}\;n \longrightarrow \mathrm{Set} \\
&[\![\,\varphi\,]\!]'\;\vec{b} = \mathrm{Atom}([\![\,\varphi\,]\!]\mathrm{b}\;\vec{b})
\end{aligned}$$

Correctness of check0 and Induction Step of check1

$$
\begin{array}{lll}
\text{lemma1} & : & (\varphi : \text{For}) \to (\text{Atom}\ (\text{check0}\ \varphi) \leftrightarrow [\![\, \varphi \,]\!]\ [\,]) \\[1em]
\text{lemma2} & : & (\varphi : \text{For}) \\
& & \to \{n : \mathbb{N}\} \to (\vec{b} : \text{Vector}\ \mathbb{B}\ (n + 1)) \\
& & \to ([\![\, \varphi \,]\!]\ \vec{b} \leftrightarrow [\![\, \text{instantiate-}\ \varphi\ (\text{head}\ \vec{b}) \,]\!]\ (\text{tail}\ \vec{b}))
\end{array}
$$

Correctness of check1

$$
\begin{aligned}
\mathrm{correctnessCheck1} \quad : \quad & (\varphi : \mathrm{For}) \\
& \rightarrow (n : \mathbb{N}) \\
& \rightarrow (\mathrm{Atom}\ (\mathrm{check1}\ \varphi\ n) \\
& \qquad \leftrightarrow ((\vec{b} : \mathrm{Vector}\ \mathbb{B}\ n) \rightarrow [\![\ \varphi\ ]\!]\ \vec{b}))
\end{aligned}
$$

Independence of $[\![\varphi]\!]\ \vec{b}$ of Variables out of Range

Let

$$\mathrm{truncateWithDefaultTt} : \{m : \mathbb{N}\} \to \mathrm{Vector}\ \mathit{Bool}\ m \to (n : \mathbb{N}) \to \mathrm{Vector}\ \mathbb{B}\ m$$

which

- truncates its argument to length n
- iff necessary fills it by tt.

$$\begin{aligned}\mathrm{lemma4} \quad : \quad & (\varphi : \mathrm{For}) \\ & \to (n : \mathbb{N}) \\ & \to (\mathrm{maxVar}\ \varphi \leq n) \\ & \to \{m : \mathbb{N}\} \to (\vec{b} : \mathrm{Vector}\ \mathbb{B}\ m) \\ & \to ([\![\varphi]\!]\ \vec{b} \leftrightarrow [\![\varphi]\!]\ (\mathrm{truncateWithDefaultTt}\ \vec{b}\ n))\end{aligned}$$

Equivalence of $[\![\varphi]\!]\ \vec{b}$ and $[\![\varphi]\!]'\ \vec{b}$

$$
\begin{aligned}
\mathrm{lemma3} \quad : \quad & (\varphi : \mathrm{For}) \\
& \rightarrow \{n : \mathbb{N}\} \rightarrow (\vec{b} : \mathrm{Vector}\ \mathbb{B}\ n) \\
& \rightarrow ([\![\varphi]\!] \vec{b} \leftrightarrow [\![\varphi]\!]'\ \vec{b}))
\end{aligned}
$$

Correctness of check

$$
\begin{aligned}
\text{corrrectnessCheck} \quad : \quad & (\varphi : \text{For}) \\
& \rightarrow \text{Atom}\ (\text{check}\ \varphi) \\
& \rightarrow \{m : \mathbb{N}\} \rightarrow (\vec{b} : \text{Vector}\ \mathbb{B}\ m) \\
& \rightarrow [\![\varphi]\!]\ \vec{b}
\end{aligned}
$$

$$
\begin{aligned}
\text{corrrectnessCheck}' \quad : \quad & (\varphi : \text{For}) \\
& \rightarrow \text{Atom}\ (\text{check}\ \varphi) \\
& \rightarrow \{m : \mathbb{N}\} \rightarrow (\vec{b} : \text{Vector}\ \mathbb{B}\ m) \\
& \rightarrow [\![\varphi]\!]'\ \vec{b}
\end{aligned}
$$

Example

x_0 : For
$x_0 = \mathrm{x}\ 0$

x_1 : For
$x_1 = \mathrm{x}\ 1$

example : For
$\mathrm{example} = ((x_0 \ \wedge\mathrm{for}\ \ x_1)\ \ \vee\mathrm{for}\ \ (\neg\mathrm{for}\ x_0)) \vee\mathrm{for}\ (\neg\mathrm{for}\ x_1)$

$\mathrm{proof} : (b\ b' : \mathbb{B}) \to ((\mathrm{Atom}\ b \wedge \mathrm{Atom}\ b') \vee (\neg(\mathrm{Atom}\ b)) \vee (\neg(\mathrm{Atom}\ b'))$
$\mathrm{proof}\ b\ b' = \mathrm{correctnessCheck\ example1\ triv}\ (b :: (b' :: []))$

$\mathrm{proof}' : (b\ b' : \mathbb{B}) \to \mathrm{Atom}(((b\ \ \wedge\mathbb{B}\ \ b')\ \ \vee\mathbb{B}\ \ (\neg\mathbb{B}\ b))\ \ \vee\mathbb{B}\ \ (\neg\mathbb{B}\ b'))$
$\mathrm{proof}'\ b\ b' = \mathrm{correctnessCheck}'\ \mathrm{example1\ triv}\ (b :: (b' :: []))$

Conclusion

- Proof in case of the SAT solver relatively short and quite readable.
- Builtin tool has been implemented by Karim Kanso; problem that it is not part of official Agda, therefore difficult to maintain with new versions.
  - Need for a more flexible builtin mechanism in Agda.
- Karim Kanso is carrying the same out for Model checking (CTL).

Future Work

- Combine with semidecision procedure.
- Combine with automated theorem provers which provide certificates.