Combining Automated and Interactive Theorem Proving in Agda

Anton Setzer
(Joint work with Karim Kanso)

May 21, 2010
1. An Introduction to Agda

2. Integrating Automated Theorem Proving into Agda

3. Defining the Mini SAT Solver in Agda

4. Correctness Proof for the Mini SAT Solver
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

\[
\text{Set} \subseteq \text{Set1} \subseteq \text{Set2} \subseteq \cdots
\]

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

```agda
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```
Once a set is introduced in this way functions can be defined recursively, as long as termination is accepted by the termination checker.

Example

\[
\begin{align*}
\text{double} &: \mathbb{N} \to \mathbb{N} \\
\text{double } \text{zero} &= \text{zero} \\
\text{double } (\text{suc } n) &= \text{suc } (\text{suc } (\text{double } n))
\end{align*}
\]
Agda allows mixfix symbols, with positions denoted by `_` e.g.

\[ (\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} ) \]
\[ n + \text{zero} = n \]
\[ n + \text{suc } m = \text{suc } (n + m) \]

We replace \text{suc} by `_+1`, use builtin \( \mathbb{N} \) which allows 0 and obtain

\[ (\_+\_ : \mathbb{N} \to \mathbb{N} \to \mathbb{N} ) \]
\[ n + 0 = n \]
\[ n + (m +1) = (n + m) +1 \]

It supports as well the use of Unicode symbols.

This allows to write code which looks very close to mathematical code.
Assume we have defined the type of matrices $\text{Mat } n \ m$ depending on dimensions $n$ and $m$:

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

Then the type of matrix multiplication is

$$\text{matmult} : (n \ m \ k : \mathbb{N}) \to \text{Mat } n \ m \to \text{Mat } m \ k \to \text{Mat } n \ k$$
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*)

```agda
data Vector(X : Set) : \mathbb{N} \rightarrow Set where
  [] : Vector X zero
  _ :: _ : X \rightarrow \{n : \mathbb{N}\} \rightarrow Vector X n \rightarrow Vector X (n + 1)
```

e.g. (using the builtin natural numbers)

```agda
a : Vector \mathbb{N} 3
a = 0 :: 1 :: 2 :: []
```
Logic in Agda (which is intuitionistic) is based on the principle of propositions as types:

- Propositions are elements of \( \text{Set} \).
- Elements of propositions are proofs of this proposition.
- A proposition holds iff it has a proof.

Examples:

- The **true proposition**:

  ```agda
data \( \top \) : \text{Set} \ where
  \text{triv} : \top
  ```
1. An Introduction to Agda

⊥, ∧, ∨

The **false proposition**:

\[
\text{data } ⊥ : \text{Set where}
\]

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

\[
f : ⊥ \to \mathbb{N}
\]

\[
f ()
\]

**Conjunction**:

\[
_\wedge_ (A, B : \text{Set}) : \text{Set where}
\]

\[
\text{and} : A \to B \to A \wedge B
\]

**Disjunction**:

\[
_\vee_ (A, B : \text{Set}) : \text{Set where}
\]

\[
inl : A \to A \vee B
\]

\[
inr : B \to A \vee B
\]
1. An Introduction to Agda

→, ¬, ∀, ∃

- **Implication**: \( A \to B \) is the function type \( A \to B \).
- **Negation**: \( \neg A = A \to \bot \).
- **Universal quantification**: \( \forall x : A. \varphi \) is given as
  \[
  (x : A) \to \varphi
  \]
- **Existential quantification**:  
  \[
  \text{data } \exists (A : \text{Set}) \ (\varphi : A \to \text{Set}) : \text{Set where exists : } (x : A) \to (\varphi \ x) \to \exists A \varphi
  \]
- **Example**:
  \( \forall \epsilon > 0. \exists \delta > 0. \varphi(\epsilon, \delta) \) is written as
  \[
  (\epsilon : \mathbb{Q}) \to \epsilon > 0 \to \exists \mathbb{Q} (\lambda \delta. \delta > 0 \land \varphi \epsilon \delta)
  \]
Decidable Prime Formulas

Booleans:

\[
\begin{align*}
data \mathbb{B} & : \text{Set} \quad \text{where} \\
\text{tt} & : \mathbb{B} \\
\text{ff} & : \mathbb{B}
\end{align*}
\]

Atom converts Booleans into the corresponding formula:

\[
\begin{align*}
\text{Atom} : \mathbb{B} & \rightarrow \text{Set} \\
\text{Atom} \; \text{tt} & = \top \\
\text{Atom} \; \text{ff} & = \bot
\end{align*}
\]
2. Integrating Automated Theorem Proving into Agda

1. An Introduction to Agda

2. Integrating Automated Theorem Proving into Agda

3. Defining the Mini SAT Solver in Agda

4. Correctness Proof for the Mini SAT Solver
Main Idea

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

  \[
  \text{data For : Set where} \\
  \ldots \\
  \text{Env : Set}
  \]

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

- Define a function \( \text{⟦}_\_\text{⟧} \) which assigns to codes for formulas and environments the corresponding Agda formula:

  \[
  \text{⟦}_\_\text{⟧} : \text{For} \rightarrow \text{Env} \rightarrow \text{Set}
  \]
Main Idea

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

\[
\text{check} : \text{For} \rightarrow \mathbb{B}
\]

Prove that check is correct:

\[
\text{correctCheck} : (\varphi : \text{For}) \rightarrow \text{Atom} (\text{check} \varphi) \rightarrow (\xi : \text{Env}) \rightarrow [\varphi] \xi
\]

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

\[
\{-\# \text{BUILTIN CHECK check} \#-\}
\]

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 $\psi$, e.g.

$$\psi : \mathbb{B} \rightarrow \mathbb{B} \rightarrow \text{Set}$$

$$\psi \ b \ b' = (\text{Atom} \ b \land \text{Atom} \ b') \lor \lnot(\text{Atom} \ b) \lor \lnot(\text{Atom} \ b')$$

Assume that $\psi$ has a code $\llbracket \psi \rrbracket$ in For, i.e.

$$\llbracket \psi \rrbracket : \text{For}$$

$$\llbracket \psi \rrbracket = \cdots$$

s.t.

$$\llbracket \llbracket \psi \rrbracket \rrbracket [x \mapsto b, y \mapsto b'] = \psi \ b \ b'$$
Then we can prove this formula (which we could prove by hand) as follows:

\[
\begin{align*}
\text{theorem} &: (b \; b' : \mathbb{B}) \rightarrow \psi \; b \; b' \\
\text{theorem } b \; b' &= \text{correctCheck } \lceil \psi \rceil \; \text{triv } [x \mapsto b, \; y \mapsto b']
\end{align*}
\]

Type checking

\[\text{triv} : \text{Atom } (\text{check } \lceil \psi \rceil)\]

will require that

\[\text{check } \lceil \psi \rceil\]

evaluates to \(\texttt{tt}\).

This evaluation will activate the automated theorem proving tool.

Note that in the example above we obtain

\[
\begin{align*}
\text{theorem} &: (b \; b' : \mathbb{B}) \rightarrow (\text{Atom } b \land \text{Atom } b') \lor \neg(\text{Atom } b) \lor \neg(\text{Atom } b')
\end{align*}
\]
Interleaving Interactive and Automated Theorem Proving

This allows to combine both theorem proving techniques:

Interactive Theorem Proving
↓
Automated Theorem Proving
↓
Interactive Theorem Proving
↓
Automated Theorem Proving
↓

...
The function \texttt{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 \texttt{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.
3. Defining the Mini SAT Solver in Agda

1. An Introduction to Agda

2. Integrating Automated Theorem Proving into Agda

3. Defining the Mini SAT Solver in Agda

4. Correctness Proof for the Mini SAT Solver
data For : Set where
  const : B → For
  x : N → For
  ∧for_ : For → For → For
  ∨for_ : For → For → For
  ¬for : For → For

check0 checks whether the formula holds if all variables are instantiated with \( \mathsf{tt} \):

\[
\begin{align*}
\text{check0 : For → } &\ B \\
\text{check0 } (\text{const } b) &\ = \ b \\
\text{check0 } (x \ n) &\ = \ \mathsf{tt} \\
\text{check0 } (\varphi \ \land \forall \ \psi) &\ = \ \text{check0 } \varphi \ \land B \ \text{check0 } \psi \\
\text{check0 } (\neg \forall \varphi) &\ = \ \neg B \ (\text{check0 } \varphi)
\end{align*}
\]
3. Defining the Mini SAT Solver in Agda

**instantiate-**

- **Instantiates in φ variable x 0 by b**
- **Replaces x (n +1) by x n**

\[
\text{instantiate-: } \text{For} \to \mathbb{B} \to \text{For} \\
\text{instantiate- (const } b) \quad b' = \text{const } b \\
\text{instantiate- (x 0)} \quad b' = \text{const } b' \\
\text{instantiate- (x (n +1))} \quad b' = \text{x n} \\
\text{instantiate- (φ } \land \text{for } \psi) \quad b' = \text{instantiate- } \varphi \ b' \\
\quad \land \text{for } \text{instantiate- } \psi \ b' \\
\quad \lor \text{for } \text{instantiate- } \psi \ b' \\
\text{instantiate- (¬for } \varphi) \quad b' = \text{¬for } (\text{instantiate- } \varphi \ b')
\]
check1 \( \varphi \ n \) checks whether \( \varphi \) is universally true if

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

\[
\begin{align*}
\text{check1} & : \text{For} \to \mathbb{N} \to \mathbb{B} \\
\text{check1} \ \varphi \ 0 & = \ \text{check0} \ \varphi \\
\text{check1} \ \varphi \ (n + 1) & = \ \text{check1} \ (\text{instantiate-} \ \varphi \ \texttt{tt}) \ n \\
& \land \mathbb{B} \ \text{check1} \ (\text{instantiate-} \ \varphi \ \texttt{ff}) \ n
\end{align*}
\]
maxVar

maxVar returns

\[ \max\{ n +1 \mid (x\ n)\ \text{occurs in } \varphi \} \]

\[
\begin{align*}
\text{maxVar} &: \text{For} \to \mathbb{N} \\
\text{maxVar} \ (\text{const } b) &= 0 \\
\text{maxVar} \ (x\ n) &= n + 1 \\
\text{maxVar} \ (\varphi \land \text{for } \psi) &= \max (\text{maxVar } \varphi) (\text{maxVar } \psi) \\
\text{maxVar} \ (\neg \text{for } \varphi) &= \text{maxVar } \varphi
\end{align*}
\]

Now we define check:

\[
\begin{align*}
\text{check} &: \text{For} \to \mathbb{B} \\
\text{check } \varphi &= \text{check1 } \varphi (\text{maxVar } \varphi)
\end{align*}
\]
3. Defining the Mini SAT Solver in Agda

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

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

$$[[\varphi]] : \text{For} \to \{n : \mathbb{N}\} \to \text{Vector } \mathbb{B} \ n \to \text{Set}$$

- $[[\text{const } b]] \ b = \text{Atom } b$
- $[[x \ n]] [] = \text{Atom } \text{tt}$
- $[[x \ 0]] (b :: \ b) = \text{Atom } b$
- $[[x \ (n + 1)] \ (b :: \ b)] = [[x \ n]] \ b$
- $[[\varphi \ \wedge \text{for } \psi ]] \ b = [[\varphi]] \ b \ \wedge \ [[\psi]] \ b$
- $[[\neg \text{for } \varphi]] \ b = \neg ([[\varphi]] \ b)$
3. Defining the Mini SAT Solver in Agda

\[ \llbracket \varphi \rrbracket^b \]

We have

\[
\llbracket x \ 0 \ \land \ \forall x \ 1 \rrbracket (b :: b' :: []) = \text{Atom } b \land \text{Atom } b'
\]

We define as well \( \llbracket \varphi \rrbracket^b \) s.t.

\[
\llbracket x \ 0 \ \land \ \forall x \ 1 \rrbracket^b (b :: b' :: []) = b \land \mathbb{B} b'
\]

\[
\llbracket - \rrbracket^b : \text{For} \rightarrow \{ n : \mathbb{N} \} \rightarrow \text{Vector } \mathbb{B} \ n \rightarrow \mathbb{B}
\]

\[
\llbracket \text{const } b \rrbracket^b \vec{b} = b
\]

\[
\llbracket x \ n \rrbracket^b [] = \text{tt}
\]

\[
\llbracket x \ 0 \rrbracket^b (b :: \vec{b}) = b
\]

\[
\llbracket x \ (n + 1) \rrbracket^b (b :: \vec{b}) = \llbracket x \ n \rrbracket^b \vec{b}
\]

\[
\llbracket \varphi \land \forall \psi \rrbracket^b \vec{b} = \llbracket \varphi \rrbracket^b \vec{b} \land \mathbb{B} \llbracket \psi \rrbracket^b \vec{b}
\]

\[
\llbracket \neg \varphi \rrbracket^b \vec{b} = \neg \mathbb{B} (\llbracket \varphi \rrbracket^b \vec{b})
\]
We define $\llbracket \varphi \rrbracket'$ s.t.

\[
\llbracket x \ 0 \ \wedge \text{for } x \ 1 \rrbracket' (b :: b' :: []) = \text{Atom} (b \ \wedgeB b')
\]

\[
\llbracket \_ \rrbracket': \text{For} \to \{ n : \mathbb{N} \} \to \text{Vector} \ \mathbb{B} \ n \to \text{Set}
\]

\[
\llbracket \varphi \rrbracket' \ b = \text{Atom}(\llbracket \varphi \rrbracket b \ \vec{b})
\]
4. Correctness Proof for the Mini SAT Solver

1. An Introduction to Agda

2. Integrating Automated Theorem Proving into Agda

3. Defining the Mini SAT Solver in Agda

4. Correctness Proof for the Mini SAT Solver
Correctness of check0 and Induction Step of check1

lemma1 : \( (\varphi : \text{For}) \rightarrow (\text{Atom} \ (\text{check0} \ \varphi) \leftrightarrow \llbracket \varphi \rrbracket \emptyset) \)

lemma2 : \( (\varphi : \text{For}) \rightarrow \{ n : \mathbb{N} \} \rightarrow (\vec{b} : \text{Vector} \ \mathbb{B} \ (n + 1)) \rightarrow (\llbracket \varphi \rrbracket \vec{b} \leftrightarrow \llbracket \text{instantiate-} \ \varphi \ (\text{head} \ \vec{b}) \rrbracket \ (\text{tail} \ \vec{b})) \)
Correctness of check1

correctnessCheck1 : (φ : For)
   → (n : ℕ)
   → (Atom (check1 φ n)
       ↔ ((b : Vector ℝ n) → [[φ]] b))
Independence of \( \langle \varphi \rangle \vec{b} \) of Variables out of Range

Let

\[
\text{truncateWithDefaultTt} : \{m : \mathbb{N}\} \rightarrow \text{Vector } \mathbb{B} \ m \rightarrow (n : \mathbb{N}) \rightarrow \text{Vector } \mathbb{B} \ m
\]

which

- truncates its argument to length \( n \)
- iff necessary fills it by \( \text{tt} \).

\[
\text{lemma4} : (\varphi : \text{For}) \rightarrow (n : \mathbb{N}) \rightarrow (\maxVar \varphi \leq n) \rightarrow \{m : \mathbb{N}\} \rightarrow (\vec{b} : \text{Vector } \mathbb{B} \ m) \rightarrow (\langle \varphi \rangle \vec{b} \leftrightarrow \langle \varphi \rangle (\text{truncateWithDefaultTt} \vec{b} n))
\]
Equivalence of $\llbracket \varphi \rrbracket \vec{b}$ and $\llbracket \varphi \rrbracket' \vec{b}$

lemma3 : $(\varphi : \text{For}) \\
\quad \rightarrow \{n : \mathbb{N}\} \rightarrow (\vec{b} : \text{Vector } \mathbb{B} \ n) \\
\quad \rightarrow (\llbracket \varphi \rrbracket \vec{b} \leftrightarrow \llbracket \varphi \rrbracket' \vec{b}))$
Correctness of check

correctnessCheck : (φ : For)
→ Atom (check φ)
→ \{m : \mathbb{N}\} → (\vec{b} : \text{Vector } \mathbb{B} \ m)
→ \llbracket φ \rrbracket \vec{b}

correctnessCheck' : (φ : For)
→ Atom (check φ)
→ \{m : \mathbb{N}\} → (\vec{b} : \text{Vector } \mathbb{B} \ m)
→ \llbracket φ \rrbracket' \vec{b}
Example

$x_0 : \text{For}$
$x_0 = x \, 0$

$x_1 : \text{For}$
$x_1 = x \, 1$

elementary : \text{For}$
elementary = (((x_0 \land \text{for } x_1) \lor \text{for } (\neg \text{for } x_0)) \lor \text{for } (\neg \text{for } x_1)$

proof : $(b \, b' : B) \rightarrow (((\text{Atom } b \land \text{Atom } b') \lor (\neg (\text{Atom } b)) \lor (\neg (\text{Atom } b')))$
proof $b \, b' = \text{correctnessCheck} \text{ example1 } \text{triv } (b : (b' :: []))$

proof' : $(b \, b' : B) \rightarrow \text{Atom}(((b \land B \, b') \lor B \, (\neg B \, b)) \lor B \, (\neg B \, b'))$
proof' $b \, b' = \text{correctnessCheck}' \text{ example1 } \text{triv } (b : (b' :: []))$
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.