Integrating an Automated Theorem Prover into Agda

Simon Foster and Georg Struth

University of Sheffield, UK

April 4, 2011

Premise

- Formal Methods is about building dependable software
- Approaches: post-hoc verification or systematic development

"...one should not first make the program and then prove its correctness, because then the requirement of providing the proof would only increase the poor programmer's burden. On the contrary: the programmer should let correctness proof and program grow hand in hand."
– ***Dijkstra, ACM Turing Lecture 1972***

- Need to close the formalisation gap between program and spec

Agda

- A functional specification/programming language **and** proof assistant
- Closes the formalisation gap
- Based on constructive type theory ("proofs-as-programs")
- Proofs used to programs with correctness guarantee
- Program derivation facilitated by meta-variable refinement

Proof in Agda

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ

data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n

n < m = suc n ≤ m
```

Proof in Agda

```
data _⩽_ : ℕ → ℕ → Set where
  z⩽n : ∀ {n} → zero ⩽ n
  s⩽s : ∀ {m n} (m⩽n : m ⩽ n) → suc m ⩽ suc n

n < m = suc n ⩽ m


greater : ∀ (n : ℕ) → ∃ (λ (m : ℕ) → n < m)
greater n = ?
```

Proof in Agda

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n

n < m = suc n ≤ m
```

```
greater : ∀ (n : ℕ) → ∃ (λ (m : ℕ) → n < m)
greater n = { } 0
```

$$\textbf{0)} \quad \frac{\textbf{Goal} : \exists(\lambda(m : \mathbb{N}) \to m < n)}{n : \mathbb{N}}$$

Proof in Agda

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n
n < m = suc n ≤ m
```

```
greater : ∀ (n : ℕ) → ∃ (λ (m : ℕ) → n < m)
greater zero = { } 0
greater (suc n) = { } 1
```

0) $$\frac{\textbf{Goal} : \exists\ (\lambda\ (m : \mathbb{N}) \to 0 < m)}{\_}$$

1) $$\frac{\textbf{Goal} : \exists\ (\lambda\ (m : \mathbb{N}) \to suc\ \ n < m)}{n : \mathbb{N}}$$

Proof in Agda

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n

n < m = suc n ≤ m
```

```
greater : ∀ (n : ℕ) → ∃ (λ (m : ℕ) → n < m)
greater zero = { } 0, { } 1
greater (suc n) = { } 2
```

$$0) \quad \frac{\textbf{Goal} : \mathbb{N}}{\_}$$

$$1) \quad \frac{\textbf{Goal} : 0 < ?0)}{n : \mathbb{N}}$$

Proof in Agda

```
data _≤_ : ℕ → ℕ → Set where
  z≤n : ∀ {n} → zero ≤ n
  s≤s : ∀ {m n} (m≤n : m ≤ n) → suc m ≤ suc n

n < m = suc n ≤ m
```

```
greater : ∀ (n : ℕ) → ∃ (λ (m : ℕ) → n < m)
greater zero = 1, s≤s z≤n
greater (suc n) = { } 2
```

$$\cfrac{\cfrac{\cfrac{-}{z \leqslant n : 0 \leqslant 0}}{s{\leqslant}s\ z{\leqslant}n : 1 \leqslant 1}}{s{\leqslant}s\ z{\leqslant}n : 0 < 1}$$

Proof in Agda

$$
\begin{array}{l}
\textbf{data}\ \_\leqslant\_\ :\ \mathbb{N} \to \mathbb{N} \to \mathsf{Set}\ \textbf{where} \\
\quad \mathsf{z{\leqslant}n}\ :\ \forall\ \{\mathsf{n}\} \to \mathsf{zero}\ \leqslant\ \mathsf{n} \\
\quad \mathsf{s{\leqslant}s}\ :\ \forall\ \{\mathsf{m\ n}\}\ (\mathsf{m{\leqslant}n}\ :\ \mathsf{m}\ \leqslant\ \mathsf{n}) \to \mathsf{suc\ m}\ \leqslant\ \mathsf{suc\ n} \\
\mathsf{n} < \mathsf{m}\ =\ \mathsf{suc\ n} \leqslant \mathsf{m}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{greater}\ :\ \forall\ (\mathsf{n}\ :\ \mathbb{N}) \to \exists\ (\lambda\ (\mathsf{m}\ :\ \mathbb{N}) \to \mathsf{n} < \mathsf{m}) \\
\mathsf{greater\ zero}\ =\ 1, \mathsf{s{\leqslant}s\ z{\leqslant}n} \\
\mathsf{greater\ (suc\ n)}\ =\ \mathsf{suc}\ (\mathsf{proj_1}\ (\mathsf{greater\ n})), \mathsf{s{\leqslant}s}\ (\mathsf{proj_2}\ (\mathsf{greater\ n}))
\end{array}
$$

- Agda's internal automated prover (Agsy) can solve this
- However it struggles with more complicated/larger proofs

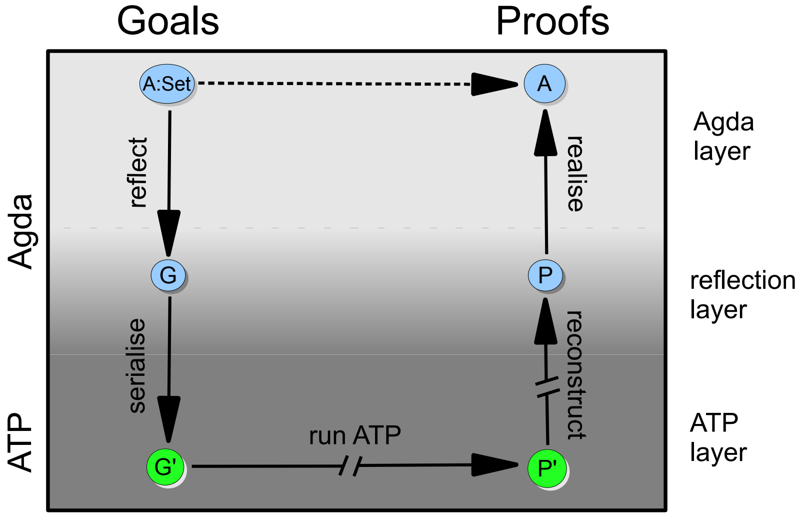
Objective

- Increase the level of automation for the programmer
- Combine the power of Agda with automated theorem provers
- ATPs like E, Vampire, SPASS, Waldmeister provide efficient solutions to large problem domains
- Fast SMT solvers exist for vectors, linear equations etc.
- Isabelle has a mature ATP/SMT integration (Sledgehammer)
- Can make development cleaner, faster and less error prone

Our contribution

- First step toward more powerful proof automation by
- a prototypical integration of Waldmeister
- the fastest equational reasoner in the world
- We provide
  - reflection layer data-types to represent problems/proofs
  - proof that a rewrite sequence respects (propositional) equality
  - a Haskell module to execute the ATP and reformat the output
  - a selection of toy examples
- Considerations:
  - classical vs. constructive logic (N/A for equational reasoning)
  - macro-step vs. micro-step proof reconstruction
  - internal vs. external proof representation

Integration Overview



$\mathsf{rev} \cdot \mathsf{rev} = \mathsf{id}$

Step 1: Reflection layer axioms

List signature Σ-List contains [], ::, ++ and rev

$$
\begin{aligned}
&\text{‘List} : \text{HypVec} \\
&\text{‘List} = \text{HyVec } \Sigma\text{-List axioms} \\
&\quad \textbf{where} \\
&\quad \text{++-nil} = \Gamma 1, \text{‘[] ‘}\!+\!\!+\ \alpha \approx \alpha \\
&\quad \text{++-cons} = \Gamma 3, (\alpha\ \text{‘::}\ \beta)\ \text{‘}\!+\!\!+\ \gamma \approx \alpha\ \text{‘::}\ (\beta\ \text{‘}\!+\!\!+\ \gamma) \\
&\quad \text{rev-nil} = \Gamma 0,\ \text{‘rev ‘[]} \approx \text{‘[]} \\
&\quad \text{rev-cons} = \Gamma 2,\ \text{‘rev}\ (\alpha\ \text{‘::}\ \beta) \approx \text{‘rev}\ \beta\ \text{‘}\!+\!\!+\ (\alpha\ \text{‘::}\ \text{‘[]}) \\
&\quad \text{axioms} = (\text{++-nil} :: \text{++-cons} :: \text{rev-nil} :: \text{rev-cons} :: [\,])
\end{aligned}
$$

$\mathsf{rev} \cdot \mathsf{rev} = \mathsf{id}$

Step 2: Creation of the goal(s)

- a rewrite proof has the form $E, L, \Gamma \vdash A \Longrightarrow s \approx t$
- E is the hypothesis vector (the basic axioms)
- L is a set of lemmas (rewrite proofs)
- Γ is the variable context
- A is a set of assumptions under Γ
- s and t are terms
- we have a correctness proof for rewrite proofs

$$E, L, A \vdash s = t \Longrightarrow [\![E]\!], [\![L]\!], [\![A]\!] \models [\![s]\!] \cong [\![t]\!]$$

$\mathsf{rev} \cdot \mathsf{rev} = \mathsf{id}$

Step 2: Creation of the goal(s)

$$
\begin{aligned}
&\text{rev-rev-nil} \;:\; \text{‘List}, [\,], \Gamma 0 \vdash [\,] \Rightarrow \;\text{‘rev}\;(\text{‘rev}\;\;\text{‘}[\,]) \approx \;\text{‘}[\,] \\
&\text{rev-rev-nil} \;=\; \{\,\}\; 0
\end{aligned}
$$

$$
\begin{aligned}
\text{rev-rev-cons} \;:\; & \text{‘List}, ((\Gamma 2,\; \text{‘rev}\;(\alpha\;\; \text{‘}{+\!\!+}\;\; \beta) \\
& \qquad\qquad\qquad \approx\; \text{‘rev}\;\beta\;\; \text{‘}{+\!\!+}\; \text{‘rev}\;\alpha) :: [\,]), \\
& \qquad \Gamma 2 \vdash ((\text{‘rev}\;(\text{‘rev}\;\beta) \approx \beta) :: [\,]) \Rightarrow \\
& \qquad\qquad (\text{‘rev}\;(\text{‘rev}\;(\alpha\;\; \text{‘}{::}\; \beta))) \approx (\alpha\;\; \text{‘}{::}\; \beta) \\
\text{rev-rev-cons} \;=\; & \{\,\}\; 1
\end{aligned}
$$

rev · rev = id

Step 3: Serialisation of Waldmeister input

```
EQUATIONS       app(nil,xs) = xs
                app(cons(x,xs),ys) = cons(x,app(xs,ys))
                rev(nil) = nil
                rev(cons(x,xs)) = app(rev(xs),cons(x,nil))
                app(rev(ys),rev(xs)) = rev(app(xs,ys))
                rev(rev(as)) = as
CONCLUSION      rev(rev(cons(a,as))) = cons(a,as)
```

rev · rev = id

Step 4: Execution of Waldmeister

```
 Lemma 1: app(rev(x1),nil) = rev(x1)

   app(rev(x1),nil)
=    by Axiom 3 RL
   app(rev(x1),rev(nil))
=    by Axiom 5 LR
   rev(app(nil,x1))
=    by Axiom 1 LR
   rev(x1)

 Lemma 2: app(cons(x1,as),nil) = cons(x1,as)

 Lemma 3: rev(app(rev(as),cons(x1,nil))) = app(app(cons(x1,nil),as),nil)

 Lemma 4: rev(rev(cons(x1,as))) = cons(x1,as)

 Theorem 1: rev(rev(cons(a,as))) = cons(a,as)

   rev(rev(cons(a,as)))
=    by Lemma 4 LR
   cons(a,as)
```

rev · rev = id

Step 5: Proof reconstruction ($\sim$ 15 steps)

$$
\begin{aligned}
\text{rev-rev-cons}\ &: \text{`List}, ((\Gamma 2,\ \text{`rev}\ (\alpha\ \text{`}{+\!\!+}\ \beta) \\
&\qquad\qquad\qquad \approx\ \text{`rev}\ \beta\ \ \text{`}{+\!\!+}\ \text{`rev}\ \alpha) :: []), \\
&\qquad\quad \Gamma 2 \vdash ((\text{`rev}\ (\text{`rev}\ \beta) \approx \beta) :: []) \Rightarrow \\
&\qquad\qquad (\text{`rev}\ (\text{`rev}\ (\alpha\ \ \text{`}{::}\ \beta))) \approx (\alpha\ \ \text{`}{::}\ \beta)
\end{aligned}
$$

$$
\begin{aligned}
&\text{rev-rev-cons}\ = \\
&\quad \text{fromJust}\ (\text{reconstruct}\ (\\
&\quad (\text{inj}_1\ (\#\ 3), \text{true}, \text{eq-step}\ (0 ::^{l} []^{l}) \\
&\qquad (\text{con}\ (\#\ 4)\ ([]^{x}) ::^{s} \text{con}\ (\#\ 5)\ ([]^{x}) ::^{s} []^{s})) ::^{l} \\
&\quad (\text{inj}_1\ (\#\ 0), \text{false}, \text{eq-step}\ (0 ::^{l} []^{l}) \\
&\qquad (\text{con}\ (\#\ 2)\ (\text{con}\ (\#\ 3)\ (\text{con}\ (\#\ 5)\ ([]^{x}) ::^{x} []^{x}) ::^{x} \\
&\qquad\quad \text{con}\ (\#\ 1)\ (\text{con}\ (\#\ 4)\ ([]^{x}) ::^{x} \text{con}\ (\#\ 0)\ ([]^{x}) ::^{x} \\
&\qquad\quad []^{x}) ::^{x} []^{x}) ::^{s} []^{s})) ::^{l} \\
&\quad (\text{inj}_2\ (\#\ 0), \text{true}, \text{eq-step}\ ([]^{l})\ (\text{con}\ (\#\ 0)\ ([]^{x}) \\
&\qquad \ldots
\end{aligned}
$$

Conclusion

- First step to semi-automated program construction in Agda
- Currently only applied to toy examples
- Could serve as a template for integrating more expressive ATPs
- Future work could focus on
  - optimisation
  - transparent proof automation like Sledgehammer
  - identify suitable fragment for full first-order logic
  - better exploration of the "external approach"
- Long-term goals
  - use with Eclipse as a behind-the-scenes proof engine
  - use in an MDA-style setting for forming model transformations

Website: http://simon-foster.staff.shef.ac.uk/agdaatp