Using automated and interactive theorem proving in Agda

Andrés Sicard-Ramírez[1]

(joint work with Ana Bove[2] **and Peter Dybjer**[2]**)**

[1]EAFIT University
[2]Chalmers University of Technology

Logic and computation seminar
EAFIT University
August 20, 2010

Abstract

Interactive proof assistants based on higher-order logics (e.g. Agda, Coq) usually lack a good support of proof automation (even in the first-order world). We have been developing a tool in which Agda users obtain support from first-order automatic theorem provers (ATPs) such as Equinox and Eprover. In our current approach, the ATPs are called by our tool on users' marked conjectures after the Agda type-checking is finished.

Motivation

Proof assistants (most)

- Higher order-logic
- Interactive (more user effort)
- Expressive types systems
- Complex developments
- Response from seconds to minutes

Automatic theorem provers (ATPs) (most)

- First-order logic
- Automatic
- Untyped
- One-shot problems
- Response from seconds to hours

Motivation

Proof assistants (most)

- Higher order-logic
- Interactive (more user effort)
- Expressive types systems
- Complex developments
- Response from seconds to minutes

Automatic theorem provers (ATPs) (most)

- First-order logic
- Automatic
- Untyped
- One-shot problems
- Response from seconds to hours

$\Rightarrow$ Combination of automatic and interactive theorem proving

Overview: the agda2atp tool

Goal: Agda users obtain support from first-order ATPs

Features:

- Agda: The high level proofs steps (introduction of hypothesis, case analysis, induction steps, etc.)
- ATPs: The "trivial" proofs steps
- The ATPs are called on users' marked conjectures
- The ATPs are called after the Agda type-checking is finished

Overview: the agda2atp tool (cont.)

What we did?

1. To modify Agda to accept the users' marked conjectures
2. To translate the required Agda internal types to FOL formulas
3. To translate the FOL formulas to ATPs' inputs

Proofs examples

Example (Agda and ATPs proofs). We will see proofs by induction, pattern matching and using equational reasoning.

Users' marked conjectures

We added a new built-in pragma to Agda:

```
{-# ATP axiom myAxiom #-}
{-# ATP definition myDefinition #-}
{-# ATP hint myHypothesis #-}
{-# ATP prove myPostulate h1 h2 ... hn #-}
```

Users' marked conjectures

We added a new built-in pragma to Agda:

```
{-# ATP axiom myAxiom #-}
{-# ATP definition myDefinition #-}
{-# ATP hint myHypothesis #-}
{-# ATP prove myPostulate h1 h2 ... hn #-}
```

Example (Agda and ATPs proofs). We will see the previous proofs using the ATP pragma.

The translation algorithm

Source: Agda internal types (simplified)

$$
\begin{aligned}
\text{Types} \ni T\ U &::= S\ t \\
\text{Sorts} \ni S &::= \text{Set}_0 \mid \text{Set}_1 \mid \ldots \\
\text{Terms} \ni t &::= \text{Var}\ x \mid \text{Lam}\ \lambda x.t \mid \text{Pi}\ T\ (\lambda x.U) \mid \text{Fun}\ T\ U \\
&\quad \mid \text{Def}\ d\ t^* \mid \text{Con}\ c\ t^* \mid \text{Sort}\ S \mid \ldots
\end{aligned}
$$

The translation algorithm

Source: Agda internal types (simplified)

$$
\begin{aligned}
\text{Types} \ni T\ U &::= S\ t \\
\text{Sorts} \ni S &::= \text{Set}_0 \mid \text{Set}_1 \mid \ldots \\
\text{Terms} \ni t &::= \text{Var}\ x \mid \text{Lam}\ \lambda x.t \mid \text{Pi}\ T\ (\lambda x.U) \mid \text{Fun}\ T\ U \\
&\quad \mid \text{Def}\ d\ t^* \mid \text{Con}\ c\ t^* \mid \text{Sort}\ S \mid \ldots
\end{aligned}
$$

Target: First-order predicate logic with equality

$$
\begin{aligned}
\text{Terms}\quad t &::= \text{FOLVar}\ x \mid \text{FOLFun}\ f\ t^* \\
\text{Formulas}\ F &::= \top \mid \bot \mid \neg F \mid F \wedge F \mid F \vee F \mid F \Rightarrow F \mid F \Leftrightarrow F \\
&\quad \mid \forall x.F \mid \exists x.F \mid \text{Predicate}\ p\ t^* \mid t \equiv t
\end{aligned}
$$

The translation algorithm (cont.)

Algorithm 0.1: typeToFormula(Γ :: Env, T :: Type)

case T

of
$$\begin{cases} (\mathrm{Set}_0, t) \rightarrow \mathrm{termToFormula}(\Gamma, t) \\ (\mathrm{Set}_1, t) \rightarrow \mathrm{termToFormula}(\Gamma, t) \\ \textbf{others} \rightarrow \textbf{fail} \end{cases}$$

The translation algorithm (cont.)

Algorithm 0.2: termToFormula(Γ :: Env, t :: Term)

case t **of**

- Var $x \rightarrow$
  - **if** $x \in \Gamma$
    then return (Predicate x [ ])
    else fail
- Lam $\lambda x.t \rightarrow$
  - $x' \leftarrow$ freshVar(Γ)
  - $f \leftarrow$ termToFormula($\Gamma \cup \{x'\}, t$)
  - **return** (f)

Algorithm 0.3: termToFormula(Γ :: Env, t :: Term)

case t **of**

- Pi T $(\lambda x.U) \rightarrow$
  - $x' \leftarrow$ freshVar(Γ)
  - $f_2 \leftarrow$ typeToFormula($\Gamma \cup \{x'\}, U$)
  - **case** T **of**
    - $(\text{Set}_0, \text{Def } d) \rightarrow$
      - – $x :: \text{Set}_0$
      - **return** $(\forall x'.f_2)$
    - $(\text{Set}_0, \text{Def } d\ t_1, \ldots, t_n) \rightarrow$
      - – The variable x is a pr
      - $f_1 \leftarrow$ typeToFormula(Γ
      - **return** $(f_1 \Rightarrow f_2)$
    - $(\text{Set}_1, \text{Sort } s) \rightarrow$
      - – $x :: \text{Set}_1$
      - **return** (f_2)
    - **others** $\rightarrow$ **fail**

Algorithm 0.4: termToFormula(Γ :: Env, t :: Term)

```
case t of
  Fun T U →
    f1 ← typeToFormula(Γ, T)
    f2 ← typeToFormula(Γ, U)
    return (f1 ⇒ f2)
  Def d [ ] →
    if d ∈ {⊤, ⊥}
      then return (d)
      else return (Predicate d [ ])
  Def d [t] →
    if (d == ¬)
      then f ← termToFormula(Γ, t); return (¬ f)
      else if d ∈ {∀D, ∃D}
             then f ← termToFormula(Γ, t)
                  x ← freshVar(Γ);
                  return ((∀/∃) x.f)
             else a ← termToFOLTerm(Γ, t)
                  return (Predicate d [a])
```

Algorithm 0.5: termToFormula(Γ :: Env, t :: Term)

case t **of**

- Def d $[t_1, t_2]$ $\rightarrow$
  - **if** $d \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$
    - **then**
      - $f1 \leftarrow$ termToFormula(Γ, t_1)
      - $f2 \leftarrow$ termToFormula(Γ, t_2)
      - **return** (f_1 d f_2)
    - **else**
      - $a1 \leftarrow$ termToFOLTerm(Γ, t_1)
      - $a2 \leftarrow$ termToFOLTerm(Γ, t_2)
      - **if** ($d == \equiv$)
        - **then return** ($a_1 \equiv a_2$)
        - **else return** (Predicate d $[a_1, a_2]$)
- Def d $[t_1, \ldots, t_n]$ $\rightarrow$
  - $a_i \leftarrow$ termToFormula(Γ, t_i)
  - **return** (Predicate d $[a_1, \ldots, a_n]$)
- **others** $\rightarrow$ **fail**

The translation algorithm (cont.)

Algorithm 0.6: termToFOLTerm(Γ :: Env, t :: Term)

case t **of**

- Var $x \rightarrow$ **if** $x \in \Gamma$ **then return** (FOLVar x) **else fail**
- Con c $[t_1, \ldots, t_n]$ or Def d $[t_1, \ldots, t_n] \rightarrow$ appArgs(Γ, c/d, $[t_1, \ldots, t_n]$)
- **others** $\rightarrow$ **fail**

where

appArgs :: Env $\rightarrow$ Name $\rightarrow$ [Term] $\rightarrow$ FOLTerm

Implementation

Modification of the development version of Agda:

- Obvious modifications (lexer, parser, errors, etc.)
- To change the Agda internal signature

Implementation

Modification of the development version of Agda:

- Obvious modifications (lexer, parser, errors, etc.)
- To change the Agda internal signature

The external tool **agda2atp**:

- Agda has a lot features (implicit arguments, η-conversion rules, where clauses, etc.)
- Using Agda as an Haskell library (Agda has not a stable API)
- Source: Agda interface files (*.agdai)
- Target: TPTP
- ATPs supported: Equinox, Eprover and Metis

Related work

External, internal or mix approach

Related work

External, internal or mix approach

- Andreas Abel, Thierry Coquand and Ulf Norell (2005)

  FOL plug-in to the Gandalf system for a previous and experimental version of Agda called AgdaLight (external approach)

- Makoto Takeyama (2009)

  Integration of Agda with external tools using Agda capability to generate an executable Haskell program (mix approach)

- Anton Setzer and Karim Kanso (2010)

  Combination of automated and interactive theorem proving using a built-in pragma (mix approach)