Interactive and Automatic Theorem Proving in the First Order Theory of Combinators

Ana Bove[1], Peter Dybjer[1], Andrés Sicard-Ramírez[2]

1 Chalmers tekniska högskola, Göteborg, Sweden

2 EAFIT Medellin, Colombia

Göteborg, 30 November, 2011

Combining three strands of research

- Foundational frameworks based on partial functions and a separation of propositions and types (Feferman's "Explicit Mathematics" and Aczel's "Frege structures") and their use as *logics of functional programs*
- Proving correctness of functional programs using *automatic theorem provers for first order logic*
- *Connecting* automatic theorem provers for first order logic *to type theory systems*

Timeline

1974	First order formal combinatory arithmetic (Aczel)
1985	Logical theory of constructions as a logic for general recursive functional programs (Dybjer)
1989	Interactive proof using Isabelle (Dybjer-Sander)
1996	Gandalf: An automatic theorem prover for ALF (Tammet-Smith)
2003	Proving correctness of Haskell programs using automatic first order theorem provers (Claessen-Hamon)
2005	Connecting AgdaLight to a First-Order Logic Prover (Abel-Coquand-Norell)
current	Agda as a Logical Framework for combining the above

First order logic with equality

Terms and formulae:

$$
\begin{array}{rcl}
t & ::= & x \mid f(t, \dots, t) \\
\Phi & ::= & \bot \mid \top \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \supset \Phi \mid \neg\Phi \mid \forall x.\Phi \mid \exists x.\Phi \mid \\
 & & t = t \mid P(t, \dots, t)
\end{array}
$$

A *first order theory* is given by

- a list of function symbols f (with arities),
- a list of predicate symbols P (with arities),
- a set of proper axioms.

Agda as a logical framework for first order logic

- *Logical frameworks* based on dependent types (Martin-Löf's LF 1986, Edinburgh LF 1987, Twelf, etc): postulating the logical constants and the axioms using Curry-Howard.
- Gardner 1992 studied the *adequacy problem* for LF-representation of first order logic (and other logics), that is, whether the theorems provable in the LF-representation are the intended ones.

Example: syntax and axioms for disjunction

```
postulate _∨_ : Set → Set → Set
      inl : {A B : Set} → A → A ∨ B
      inr : {A B : Set} → B → A ∨ B
      case : {A B C : Set} → (A → C) → (B → C) → A ∨ B → C
```

Axiom *schemata* in first order logic.

Proof of commutativity of disjunction

```
commOr : {A B : Set} → A ∨ B → B ∨ A
commOr c = case inr inl c
```

Encoding classical logic

```
postulate lem : {A : Set} → A ∨ ¬ A
```

Interacting with Automatic Theorem Provers

Interactive proof:

```
commOr : {A B : Set} → A ∨ B → B ∨ A
commOr c = case inr inl c
```

Automatic proof:

```
postulate commOr : {A B : Set} → A ∨ B → B ∨ A
{-# ATP prove commOr #-}
```

Combining Agda with Automatic Theorem Provers

1. Type-check and generate interface file with axioms, definitions, conjectures (using ATP-pragmas)
2. Run `agda2atp` which
   1. translates axioms, definitions and conjectures in the interface file into the TPTP language and
   2. automatically tries to prove the conjectures using E, Equinox, SPASS, Metis, and Vampire.

In the terminal:

```
Proving the conjecture in /tmp/Examples.commOr_7.tptp ...
Vampire 0.6 (...) proved the conjecture in /tmp/Examples.com
```

Using data instead of postulates

To make use of Agda's pattern matching we define

```
data _∨_ (A B : Set) : Set where
  inl : A → A ∨ B
  inr : B → A ∨ B
```

Commutativity of disjunction with pattern matching

```
commOr : {A B : Set} → A ∨ B → B ∨ A
commOr (inl a) = inr a
commOr (inr b) = inl b
```

New adequacy problem. Only using pattern matching which can be compiled into elimination rules. Convenience vs rigour.

Encoding quantifiers

The domain of individuals of first order logic

```
postulate D : Set
```

Universal quantifier

```
∀ x → P = (x : D) → P
```

Existential quantifier

```
data ∃ (P : D → Set) : Set where
  _,_ : (x : D) → P x → ∃ P
```

```
syntax ∃ (λ x → P) = ∃[ x ] P
```

A First Order Theory of Combinators

Aczel, 1974: "The strength of Martin-Löf's intuitionistic type theory with one universe".

$$
\begin{aligned}
t &::= x \mid t\,t \mid \mathrm{K} \mid \mathrm{S} \\
\Phi &::= \bot \mid \top \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid \forall x.\Phi \mid \exists x.\Phi \mid t = t \mid \\
&\quad\;\; \mathcal{N}(t) \mid \mathcal{P}(t) \mid \mathcal{T}(t)
\end{aligned}
$$

Proper axioms:

- Conversion rules: $\mathrm{K}\,t\,t' = t$ and $\mathrm{S}\,t\,t'\,t'' = t\,t''\,(t'\,t'')$.
- Axioms for $\mathcal{N}, \mathcal{P}, \mathcal{T}$.

A Logic for PCF with totality predicates

$$
\begin{array}{rcl}
t & ::= & x \mid t\ t \mid \lambda x.t \mid \mathrm{true} \mid \mathrm{false} \mid \mathrm{if} \mid 0 \mid \mathrm{succ} \mid \mathrm{pred} \mid \mathrm{iszero} \mid \mathrm{fix} \\
\Phi & ::= & \bot \mid \top \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid \forall x.\Phi \mid \exists x.\Phi \mid t = t \mid \\
 & & \mathcal{B}ool(t) \mid \mathcal{N}(t)
\end{array}
$$

Proper axioms:

- Conversion rules: $\mathrm{if}\,\mathrm{true}\,t\,t' = t$, etc.
- Discrimination rules: $\neg\ \mathrm{true} = \mathrm{false}$. etc.
- Axioms for $\mathcal{N}, \mathcal{B}ool$.

A first order theory of combinators (FOTC) for PCF

$$
\begin{array}{rcl}
t & ::= & x \mid t\ t \mid \mathrm{true} \mid \mathrm{false} \mid \mathrm{if} \mid 0 \mid \mathrm{succ} \mid \mathrm{pred} \mid \mathrm{iszero} \mid f \\
\Phi & ::= & \bot \mid \top \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid \forall x.\Phi \mid \exists x.\Phi \mid t = t \mid \\
 & & \mathcal{B}ool(t) \mid \mathcal{N}(t)
\end{array}
$$

where x is a variable, and f a new combinator defined by a (recursive) equation

$$
f\ x_1\ \cdots\ x_n = e[f, x_1\ \cdots\ x_n]
$$

Encoding in Agda: function symbols

```
postulate if_then_else_ : D → D → D → D
          _·_ : D → D → D
          succ pred isZero : D → D
          zero true false : D
```

Conversion rules

```
postulate if-true  : ∀ d₁ {d₂} → if true then d₁ else d₂ ≡ d₁
          if-false : ∀ {d₁} d₂ → if false then d₁ else d₂ ≡ d₂
          pred-S : ∀ d → pred (succ d) ≡ d
          isZero-0 : isZero zero ≡ true
          isZero-S : ∀ d → isZero (succ d) ≡ false
{-# ATP axiom if-true if-false pred-S isZero-0 isZero-S #-}
```

Axioms for natural numbers

```
data N : D → Set where
  zN : N zero
  sN : ∀ {n} → N n → N (succ n)
{-# ATP axiom zN sN #-}

indN : (P : D → Set) → P zero →
       (∀ {n} → P n → P (succ n)) → ∀ {n} → N n → P n
indN P P0 h zN = P0
indN P P0 h (sN Nn) = h (indN P P0 h Nn)
```

Induction is an axiom *schema*! TPTP only understands *axioms*.

Totality of addition - version 1

```
postulate _+_ : D → D → D
          +-0x : ∀ d → zero + e ≡ e
          +-Sx : ∀ d e → succ d + e ≡ succ (d + e)
{-# ATP axiom +-0x +-Sx #-}

indN-instance : ∀ x → N (zero + x) →
                (∀ {n} → N (n + x) → N (succ n + x)) →
                ∀ {n} → N (n + x)
indN-instance x = indN (λ i → N (i + x))

postulate +-N₁ : ∀ {m n} → N m → N n → N (m + n)
{-# ATP prove +-N₁ indN-instance #-}
```

Totality of addition - version 2

```
+-N : ∀ {m n} → N m → N n → N (m + n)
+-N {n = n} zN Nn = prf
  where postulate prf : N (zero + n)
        {-# ATP prove prf #-}
+-N {n = n} (sN {m} Nm) Nn = prf (+-N Nm Nn)
  where postulate prf : N (m + n) → N (succ m + n)
        {-# ATP prove prf #-}
```

An inductive predicate

We can add inductive predicates other than totality predicates:

```
data Even : D → Set where
  zeroeven : Even zero
  nexteven : ∀ {d} → Even d → Even (succ (succ d))
```

Induction principle:

```
indEven : (P : D → Set) →
          P zero →
          (∀ {d} → P d → P (succ (succ d))) →
          ∀ {d} → Even d → P d
indEven P P0 h zeroeven      = P0
indEven P P0 h (nexteven Ed) = h (indEven P P0 h Ed)
```

Trees and forests

Constructors:

```
postulate [] : D
          _::_  node : D → D → D
```

Totality predicates:

```
mutual
 data Forest : D → Set where
    nilF : Forest []
    consF : ∀ {t ts} → Tree t → Forest ts → Forest (t :: ts)
 data Tree : D → Set where
    treeT : ∀ d {ts} → Forest ts → Tree (node d ts)

 {-# ATP axiom nilF consF treeT #-}
```

Map and mirror

```
postulate map : D → D → D
    map-[] : ∀ f → map f [] ≡ []
    map-:: : ∀ f d ds → map f (d :: ds) ≡ f · d :: map f ds

{-# ATP axiom map-[] map-:: #-}


postulate mirror : D
    mirror-eq : ∀ d ts → mirror · (node d ts) ≡
                         node d (reverse (map mirror ts))

{-# ATP axiom mirror-eq #-}
```

A property of mirror

```
mirror² : ∀ {t} → Tree t → mirror · (mirror · t) ≡ t
```

The proof is by induction on the mutually defined totality predicates for trees and forests:

```
mirror² (treeT d nilF) = prf
 where postulate prf : mirror · (mirror · node d []) ≡ node d []
       {-# ATP prove prf #-}
mirror² (treeT d (consF {t} {ts} Tt Fts)) = prf
 where postulate prf : mirror · (mirror · node d (t ∷ ts)) ≡
                       node d (t ∷ ts)
       {-# ATP prove prf helper #-}
```

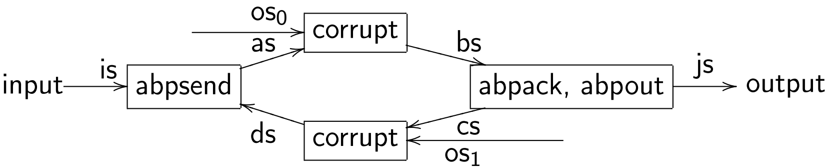
where the proof `helper` of a lemma is given as a hint.

The lemma

```
helper : ∀ {ts} → Forest ts →
         reverse (map mirror (reverse (map mirror ts))) ≡ ts
```

is proved by induction on forest and trees where the cases are proved automatically.

The alternating bit protocol as a Kahn network



$$\texttt{ax-1} : \texttt{corrupt} \cdot (1 :: \texttt{os}) \cdot (\texttt{x} :: \texttt{xs}) \equiv \texttt{ok x} :: \texttt{corrupt} \cdot \texttt{os} \cdot \texttt{xs}$$
$$\texttt{ax-0} : \texttt{corrupt} \cdot (0 :: \texttt{os}) \cdot (\texttt{x} :: \texttt{xs}) \equiv \texttt{error} :: \texttt{corrupt} \cdot \texttt{os} \cdot \texttt{xs}$$

Specification of the protocol

The protocol should implement the identity stream transformers if the unreliable channel is "fair". The output should be *bisimilar* to the input under this condition:

$$\mathtt{spec} : \mathtt{Bit}\ \mathtt{b} \rightarrow \mathtt{Stream}\ \mathtt{is} \rightarrow \mathtt{Fair}\ \mathtt{os}_0 \rightarrow \mathtt{Fair}\ \mathtt{os}_1 \rightarrow \\ \mathtt{is} \approx \mathtt{abptransfer}\ \mathtt{b}\ \mathtt{os}_0\ \mathtt{os}_1\ \mathtt{is}$$

Totality of streams

To be a total possibly infinite stream is defined coinductively, as a greatest fixed point. The axioms state that `Stream` is a postfixed point

```
Stream-gfp₁ : ∀ {xs} → Stream xs →
              ∃[ x' ] → ∃[ xs' ]→ Stream xs' ∧ xs ≡ x' :: xs'
```

and the greatest postfixed point

```
Stream-gfp₂ : (P : D → Set) →
              (∀ {xs} → P xs → ∃ [ x' ] → ∃[ xs' ] →
                          P xs' ∧ xs ≡ x' :: xs') →
              ∀ {xs} → P xs → Stream xs
```

Bisimilarity

Bisimilarity is also a postfixed point

```
≈-gfp₁ : ∀ {xs ys} → xs ≈ ys →
         ∃[ x' ]→ ∃[ xs' ] → ∃[ ys' ] →
         xs' ≈ ys' ∧ xs ≡ x' ∷ xs' ∧ ys ≡ x' ∷ ys'
```

and the greatest postfixed point

```
≈-gfp₂ : (_R_ : D → D → Set) → (∀ {xs ys} → xs R ys →
         ∃[ x' ] → ∃[ xs' ] → ∃[ ys' ] →
         xs' R ys' ∧ xs ≡ x' ∷ xs' ∧ ys ≡ x' ∷ ys') →
         ∀ {xs ys} → xs R ys → xs ≈ ys
```

Fairness

Fairness is also a postfixed point

```
Fair-gfp₁ : ∀ {os} → Fair os →
            ∃[ ol ] → ∃[ os' ] →
            O*1 ol ∧ Fair os' ∧ os ≡ ol ++ os'
```

and the greatest postfixed point

```
Fair-gfp₂ : (P : D → Set) → (∀ {os} → P os →
            ∃[ ol ] → ∃[ os' ] →
            O*1 ol ∧ P os' ∧ os ≡ ol ++ os') →
            ∀ {os} → P os → Fair os
```

The sender

$$\text{ax0} : \text{abpsend} \cdot b \cdot (i :: is) \cdot ds \equiv \langle i , b \rangle :: \text{await } b\ i\ is\ ds$$

$$\begin{aligned}\text{ax1} : {} & b \equiv b_0 \to \\ & \text{await } b\ i\ is\ (\text{ok } b_0 :: ds) \equiv \text{abpsend} \cdot (\text{not } b) \cdot is \cdot ds\end{aligned}$$

$$\begin{aligned}\text{ax2} : {} & \neg\ (b \equiv b_0) \to \\ & \text{await } b\ i\ is\ (\text{ok } b_0 :: ds) \equiv \langle i , b \rangle :: \text{await } b\ i\ is\ ds\end{aligned}$$

$$\text{ax3} : \text{await } b\ i\ is\ (\text{error} :: ds) \equiv \langle i , b \rangle :: \text{await } b\ i\ is\ ds$$

The receiver

```
ax4 : b ≡ b₀ →
  abpack · b · (ok < i , b₀ > :: bs) ≡ b :: abpack · (not b) · bs

ax5 : ¬ (b ≡ b₀) →
  abpack · b · (ok < i , b₀ > :: bs) ≡ not b :: abpack · b · bs

ax6 : abpack · b · (error :: bs) ≡ not b :: abpack · b · bs

ax7 : b ≡ b₀ →
  abpout · b · (ok < i , b₀ > :: bs) ≡ i :: abpout · (not b) · bs

ax8 : ¬ (b ≡ b₀) →
  abpout · b · (ok < i , b₀ > :: bs) ≡ abpout · b · bs

ax9 : ∀ b bs →
  abpout · b · (error :: bs) ≡ abpout · b · bs
```

The network transfer function

A higher order function that computes the output from the input and the stream tranformers associated with the edges of the network

$$
\begin{array}{lll}
\mathtt{ax10} & : & \mathtt{transfer}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is} \equiv f_3 \cdot (\mathtt{hbs}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is}) \\
\mathtt{ax11} & : & \mathtt{has}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is} \equiv f_1 \cdot \mathtt{is} \cdot (\mathtt{hds}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is}) \\
\mathtt{ax12} & : & \mathtt{hbs}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is} \equiv g_1 \cdot (\mathtt{has}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is}) \\
\mathtt{ax13} & : & \mathtt{hcs}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is} \equiv f_2 \cdot (\mathtt{hbs}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is}) \\
\mathtt{ax14} & : & \mathtt{hds}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is} \equiv g_2 \cdot (\mathtt{hcs}\ f_1\ f_2\ f_3\ g_1\ g_2\ \mathtt{is})
\end{array}
$$

The alternating bit protocol as a stream transformer

```
abptransfer-eq : abptransfer b os₀ os₁ is ≡
                 transfer (abpsend · b) (abpack · b) (abpout · b)
                          (corrupt · os₀) (corrupt · os₁) is
```

Combined automatic and interactive proof of ABP

- Proof by coinduction and induction.
- The induction and coinduction schemata must be instantiated manually.
- A large part, but far from all, of the induction-coinduction free part is done automatically by the FOL-provers. The provers are not good enough at rewriting based proofs.

The future of verified functional programming?

Pros of FOTC approach:

- Program as usual in Haskell
- General recursion
- Separate programs and proofs
- Automatic theorem proving for classical first order logic

Pros of DTP appraoch:

- Normalization and automatic type-checking
- Dependent types
- Programs as proofs

Note that the "standard" model of MLTT is an interpretation in Aczel's FOTC! Everything we do in MLTT can be translated (without much coding) into FOTC.

Related work

Lots!

- LCF, McCarthy's first order programming logic
- Boyer-Moore
- NuPRL
- MinLog
- Function package in Isabelle, Sledgehammer
- Sparkle, Plover (Programatica)
- Chargueraud (Coq)
- Bove-Capretta (MLTT)
- Etc