Combining Interactive and Automatic Reasoning in First-Order Theories of Functional Programs

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

[1]Chalmers University of Technology, Sweden

[2]EAFIT University, Colombia

Foundations of Software Science and Computation Structures
(FoSSaCS - ETAPS)
Tallinn, Estonia
28 March 2012

Introduction

What if we have written a Haskell-like program and we want to verify it?

- What programming logic should we use?
- What proof assistant should we use?
- Can (part of) the job be automatic?

Introduction

What if we have written a Haskell-like program and we want to verify it?

- What programming logic should we use?
- What proof assistant should we use?
- Can (part of) the job be automatic?

Combining three strands of research:

1. Foundational frameworks and logics for functional programs (Aczel 1974, Dybjer 1985, Dybjer and Sander 1989, Bove, Dybjer and Sicard-Ramírez 2009)
2. Proving correctness of functional programs using automatic theorem provers for first-order logic (Claessen and Hamon 2003)
3. Connecting automatic theorem provers for first-order logic to type theory systems as Agda interactive proof assistant developed at Chalmers (Tammet and Smith 1996, Abel, Coquand and Norell 2005)

Our approach

First-order theory of combinators (FOTC)

- Logic for general recursive programs
- Inductive and co-inductive definitions
- Higher-order functions
- Martin-Löf type theory is a subsystem of FOTC

Our approach

First-order theory of combinators (FOTC)

- Logic for general recursive programs
- Inductive and co-inductive definitions
- Higher-order functions
- Martin-Löf type theory is a subsystem of FOTC

Agda as a logical framework for FOTC

- Using Agda's inductive notions
- Attractive user interface for interactive theorem proving

Our approach

First-order theory of combinators (FOTC)

- Logic for general recursive programs
- Inductive and co-inductive definitions
- Higher-order functions
- Martin-Löf type theory is a subsystem of FOTC

Agda as a logical framework for FOTC

- Using Agda's inductive notions
- Attractive user interface for interactive theorem proving

Automatic proofs

- An agda2atp program which translates first-order formulae in Agda into TPTP, and calls automatic theorem provers at them
- Combining automatic and interactive proofs

A First-Order Theory of Combinators I

Terms

$$t ::= x \mid t\,t \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if} \mid 0 \mid \mathsf{succ} \mid \mathsf{pred} \mid \mathsf{iszero} \mid \mathsf{f}$$

where f a new combinator defined by a (recursive) equation

$$\mathsf{f}\ t_1 \cdots t_n = e[\mathsf{f}, t_1, \ldots, t_n]$$

A First-Order Theory of Combinators I

Terms

$$t ::= x \mid t\,t \mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if} \mid \mathsf{0} \mid \mathsf{succ} \mid \mathsf{pred} \mid \mathsf{iszero} \mid \mathsf{f}$$

where f a new combinator defined by a (recursive) equation

$$\mathsf{f}\ t_1 \cdots t_n = e[\mathsf{f}, t_1, \ldots, t_n]$$

Formulae

$$\Phi ::= \top \mid \bot \mid \Phi \Rightarrow \Phi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \neg\Phi \mid \forall x.\Phi \mid \exists x.\Phi \mid t = t$$

$\mid N(t)$ (totality natural numbers inductive predicate)

$\mid Bool(t)$ (totality Booleans inductive predicates)

$\mid \ldots$ (additional inductive and co-inductive predicates)

A First-Order Theory of Combinators II

Conversion rules

$$
\begin{aligned}
\forall t\ t'.\mathsf{if}\ \mathsf{true}\ t\ t' &= t \\
\forall t\ t'.\mathsf{if}\ \mathsf{false}\ t\ t' &= t \\
\forall t.\mathsf{pred}\ (\mathsf{succ}\ t) &= t \\
\mathsf{iszero}\ 0 &= \mathsf{true} \\
\forall t.\mathsf{iszero}\ (\mathsf{succ}\ t) &= \mathsf{false}
\end{aligned}
$$

Discrimination rules

$$
\begin{aligned}
\neg(\mathsf{true} = \mathsf{false}) \\
\forall t.\neg(0 = \mathsf{succ}\ t)
\end{aligned}
$$

A First-Order Theory of Combinators II

Conversion rules

$$\forall t\ t'.\mathsf{if}\ \mathsf{true}\ t\ t' = t$$
$$\forall t\ t'.\mathsf{if}\ \mathsf{false}\ t\ t' = t$$
$$\forall t.\mathsf{pred}\ (\mathsf{succ}\ t) = t$$
$$\mathsf{iszero}\ 0 = \mathsf{true}$$
$$\forall t.\mathsf{iszero}\ (\mathsf{succ}\ t) = \mathsf{false}$$

Discrimination rules

$$\neg(\mathsf{true} = \mathsf{false})$$
$$\forall t.\neg(0 = \mathsf{succ}\ t)$$

Axioms for $N(t)$

$$\frac{}{N(0)} \qquad \frac{N(t)}{N(\mathsf{succ}\ t)}$$

$$\Phi(0)\ \wedge\ (\forall t.\Phi(t) \Rightarrow \Phi(\mathsf{succ}\ t)) \Rightarrow \forall t.N(t) \Rightarrow \Phi(t)$$

Agda as a Logical Framework for First-Order Logic

Features

- Postulating the logical constant and their axioms (Martin-Löf's LF 1986, Edinburgh Logical Framework 1987)
- First-order formulae type: Agda `Set` (or $\texttt{Set}_0$). Agda's first universe

Agda as a Logical Framework for First-Order Logic

Features

- Postulating the logical constant and their axioms (Martin-Löf's LF 1986, Edinburgh Logical Framework 1987)
- First-order formulae type: Agda `Set` (or `Set₀`). Agda's first universe

Example (Axiom schemata for disjunction)

```
postulate
  _∨_  : Set → Set → Set
  inj₁ : {A B : Set} → A → A ∨ B
  inj₂ : {A B : Set} → B → A ∨ B
  case : {A B C : Set} → (A → C) → (B → C) → A ∨ B → C
```

Agda as a Logical Framework for First-Order Logic

Features

- Postulating the logical constant and their axioms (Martin-Löf's LF 1986, Edinburgh Logical Framework 1987)
- First-order formulae type: Agda `Set` (or `Set₀`). Agda's first universe

Example (Axiom schemata for disjunction)

```
postulate
  _∨_  : Set → Set → Set
  inj₁ : {A B : Set} → A → A ∨ B
  inj₂ : {A B : Set} → B → A ∨ B
  case : {A B C : Set} → (A → C) → (B → C) → A ∨ B → C
```

Example (Interactive proof of commutativity of disjunction)

```
∨-comm : {A B : Set} → A ∨ B → B ∨ A
∨-comm h = case inj₂ inj₁ h
```

Proof by Pattern Matching

Example (Encoding disjunction)

```
data _∨_ (A B : Set) : Set where
  inj₁ : A → A ∨ B
  inj₂ : B → A ∨ B
```

Example (Proof of commutativity of disjunction by pattern matching)

```
∨-comm : {A B : Set} → A ∨ B → B ∨ A
∨-comm (inj₁ a) = inj₂ a
∨-comm (inj₂ b) = inj₁ b
```

Interacting with Automatic Theorem Provers

Example (Automatic proof)

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

Interacting with Automatic Theorem Provers

Example (Automatic proof)

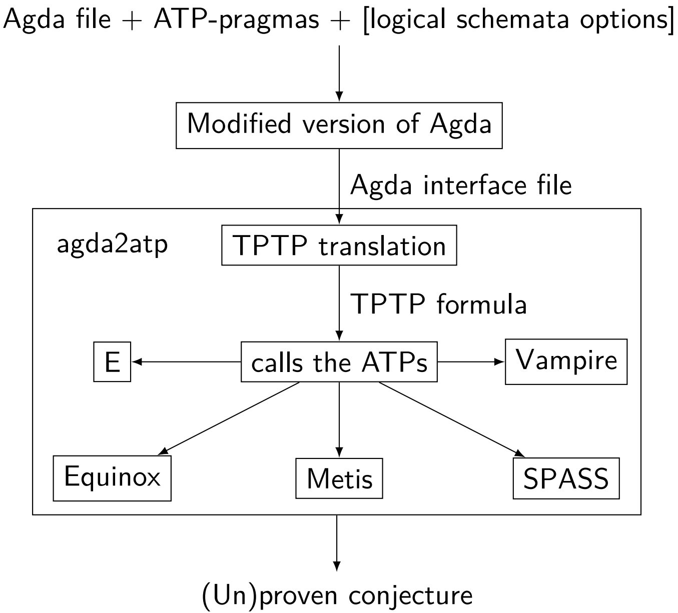
```
∨-comm : {A B : Set} → A ∨ B → B ∨ A
{-# ATP prove ∨-comm #-}
```

The automatic theorem provers use classical logic

We add as axiom the law of the excluded middle:

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

Combining Agda with Automatic Theorem Provers



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
```

Encoding Conversion Rules

Function symbols

```
postulate
  _·_             : D → D → D
  if_then_else_   : 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
```

ATPs axioms

```
{-# ATP axiom if-true if-false pred-S isZero-0 isZero-S #-}
```

Encoding Totality Inductive Predicates

Example (Totality natural numbers predicate)

Introduction rules:

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

ATP axioms:

```
{-# ATP axiom zN sN #-}
```

Induction principle:

```
N-ind : (P : D → Set) →
        P zero →
        (∀ {n} → P n → P (succ n)) →
        ∀ {n} → N n → P n
```

Remark: We will often write proof by induction using Agda's pattern matching.

The mirror Function I

Trees and forests constructors

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

Mutual totality predicates

```
data Forest : D → Set
data Tree   : D → Set

data Forest where
  nilF  : Forest []
  consF : ∀ {t ts} → Tree t → Forest ts → Forest (t ∷ ts)

data Tree where
  treeT : ∀ d {ts} → Forest ts → Tree (node d ts)
```

ATP axioms

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

The mirror Function II

Map axioms

```
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-∷ #-}
```

Mirror axioms

```
postulate
  mirror    : D
  mirror-eq : ∀ d ts →
              mirror · (node d ts) ≡ node d (reverse (map mirror ts))
{-# ATP axiom mirror-eq #-}
```

Property

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

The mirror Function III

Proof

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

Base case:

```
mirror-involutive (treeT d nilF) = prf
  where postulate prf : mirror · (mirror · node d []) ≡ node d []
        {-# ATP prove prf #-}
```

Inductive case:

```
mirror-involutive (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 #-}
```

Auxiliary lemma: (Proved similarly and given as a hint)

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

The map-iterate Property I

Map and iterate axioms

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

  iterate    : D → D → D
  iterate-eq : ∀ f x → iterate f x ≡ x :: iterate f (f · x)

{-# ATP axiom map-[] map-:: iterate-eq #-}
```

The property

Intuitively, `map f (iterate f x)` and `iterate f (f · x)` form the same infinite list: `f · x : f · (f · x) : f · (f · (f · x)) : ...`

How can the `map-iterate` property be proved?

The map-iterate Property II

Co-induction on infinite lists

- Bisimilarity: A co-inductive relation defined as a greatest fixed-point

```
_≈_ : D → D → Set
```

- Unfolding rule and co-induction principle

The map-iterate Property II

Co-induction on infinite lists

- Bisimilarity: A co-inductive relation defined as a greatest fixed-point

```
_≈_ : D → D → Set
```

- Unfolding rule and co-induction principle

The map-iterate property

Iterating a function and then mapping it gives the same result as applying the function and then iterating it:

```
∀ f x → map f (iterate f x) ≈ iterate f (f · x)
```

Proof

- The co-induction scheme must be instantiated manually on the relation (Giménez and Castéran, 2007):

```
R xs ys = ∃[ y ] xs ≡ map f (iterate f y)
                 ∧ ys ≡ iterate f (f · y)
```

- The rest was done automatically for the ATPs

Additional examples

From website www1.eafit.edu.co/asicard/code/fossacs-2012/:

- Modified version of Agda
- The agda2atp program
- First-order theory of combinators
  - The `mirror` function
  - The `map-iterate` property
  - The McCarthy 91 function
  - The alternating bit protocol written as a stream processing program
  - Additional examples of verification of programs
- Additional examples of first-order theories (Peano arithmetic, group theory, etc)

Conclusion

FOTC + Agda's inductive notions + external ATPs:

- Strong logic (Martin-Löf type theory is a subsystem of FOTC)
- General recursion
- Inductive and co-inductive definitions
- Higher-order functions
- Termination proofs
- Combined proofs using induction (pattern matching), co-induction, and ATPs
- Replacing the tedious equational reasoning by automatic proofs

Future work

- Proof reconstruction for the automatically proved theorems
- To merge FOTC-style for program verification with the dependently typed programming style (normalization and automatic type-checking)
- Integration with automatic inductive theorem provers
- A translator between Haskell programs and our Agda encoding of FOTC

Bonus slides

Termination Proofs

Addition axioms

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

Example (Totality of addition)

```
+-N : ∀ {m n} → N m → N n → N (m + n)
```

Base case:

```
+-N {n = n} zN Nn = prf
  where postulate prf : N (zero + n)
        {-# ATP prove prf #-}
```

Inductive case:

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

Replacing the Tedious Equational Reasoning

Example (Interactive proof)

```
+-comm : ∀ m n → N m → N n → m + n ≡ n + m
+-comm m n zN Nn = -- omitted
+-comm m n (sN m Nm) Nn =
  succ m + n   ≡⟨ +-Sx m n ⟩
  succ (m + n) ≡⟨ cong succ (+-comm Nm Nn) ⟩
  succ (n + m) ≡⟨ sym (x+Sy≡S[x+y] m Nn) ⟩
  n + succ m   ∎
```

Example (Combined proof)

```
+-comm : ∀ m n → N m → N n → m + n ≡ n + m
+-comm m n zN Nn = -- omitted
+-comm m n (sN m Nm) Nn = prf (+-comm Nm Nn)
  where
  postulate prf : m + n ≡ n + m → succ m + n ≡ n + succ m
  {-# ATP prove prf x+Sy≡S[x+y] #-}
```