Reasoning about Functional Programs by Combining Interactive and Automatic Proofs

Andrés Sicard-Ramírez

Supervisors: Ana Bove and Peter Dybjer
Director of studies: Alberto Pardo

Universidad de la República
Montevideo, Uruguay
25 July 2014

Introduction

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

Introduction

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

- How to deal with the possible use of general recursion (non-structural recursive, nested recursive, and higher-order recursive functions, and guarded and unguarded co-recursive functions)?

  Most of the proof assistants lack a direct treatment for general recursive functions (Bove, Krauss and Sozeau 2012).

Introduction

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

- How to deal with the possible use of general recursion (non-structural recursive, nested recursive, and higher-order recursive functions, and guarded and unguarded co-recursive functions)?

  Most of the proof assistants lack a direct treatment for general recursive functions (Bove, Krauss and Sozeau 2012).

- Other features of Haskell-Like programs

  Higher-order functions (in functional languages, functions can take functions as arguments and produce functions as results).

  Lazy (the arguments of a function are evaluated when it is strictly necessary).

  Inductive and co-inductive data types (finite and potentially infinite data).

Our Goal

To build a computer-assisted framework for reasoning about programs written in Haskell-like lazy functional languages.

Our Main Contributions

What programming logic should we use?

Our Main Contributions

What programming logic should we use?

We defined and formalised the First-Order Theory of Combinators

Our Main Contributions

What programming logic should we use?

We defined and formalised the First-Order Theory of Combinators

Target language	Type-free extended versions of Plotkin's PCF language
Basic data	Inductive and co-inductive data types
Specification language	First-order logic and predicates representing the property of being a finite or a potentially infinite value
The theory can deal with	General recursion, higher-order functions, (co-)inductive definitions of data types and proofs by (co-)induction
Consistency	Based on a translation into Dybjer's (1985) Logical Theory of Constructions

Our Main Contributions

What proof assistant should we use?

We formalise our programming logics and our examples of verification of functional programs in the Agda proof assistant:

Our Main Contributions

What proof assistant should we use?

We formalise our programming logics and our examples of verification of functional programs in the Agda proof assistant:

- we use Agda as a logical framework (meta-logical system for formalising other logics) and
- we use Agda's proof engine: (i) support for inductively defined types, including inductive families, and function definitions using pattern matching on such types, (ii) normalisation during type-checking, (iii) commands for refining proof terms, (iv) coverage checker and (v) termination checker.

Our Main Contributions

Can (part of) the job be automatic?

Yes! We can combine Agda interactive proofs and ATPs (automatic theorem provers for first-order logic) proofs:

Our Main Contributions

Can (part of) the job be automatic?

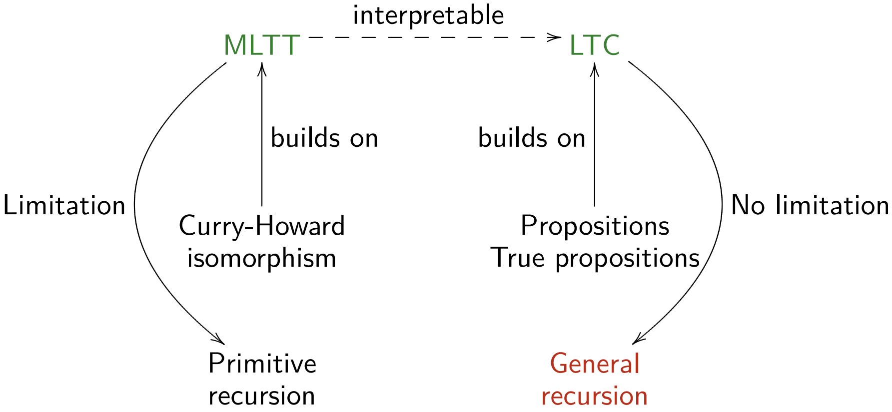
Yes! We can combine Agda interactive proofs and ATPs (automatic theorem provers for first-order logic) proofs:

- we provide a translation of our Agda representation of first-order formulae into TPTP (Sutcliffe 2009)—a language understood by many off-the-shelf ATPs—so we can use them when proving the properties of our programs,
- we extended Agda with an ATP-pragma, which instructs Agda to interact with the ATPs, and
- we wrote the Apia program, a Haskell program which uses Agda as a Haskell library, performs the above translation and calls the ATPs.

Combining Three Strands of Research

1. Foundational frameworks and logics for lazy functional programs

Why use LTC as a programming logics for lazy functional programs (Dybjer 1985, 1990; Dybjer and Sander 1989)



Combining Three Strands of Research

2. Proving correctness of functional programs using first-order automatic theorem provers

"The CoVer Translator" (Claessen and Hamon 2003)

Using ATPs for proving properties of functional programs by translating them into first-order logic.

Combining Three Strands of Research

2. Proving correctness of functional programs using first-order automatic theorem provers

"The CoVer Translator" (Claessen and Hamon 2003)

Using ATPs for proving properties of functional programs by translating them into first-order logic.

3. Connecting first-order automatic theorem provers to type theory systems

The implementation of the Apia program took some ideas from the connection of AgdaLight (an experimental version of Agda) to the Gandalf ATP (Abel, Coquand and Norell 2005).

First-Order Logic

Terms $\ni t ::= x$		variable
	$\mid c$	constant
	$\mid f(t, \dots, t)$	function

Formulae $\ni A ::= \top \mid \bot$		truth, falsehood
	$\mid A \supset A \mid A \wedge A \mid A \vee A$	binary logical connectives
	$\mid \forall x.A \mid \exists x.A$	quantifiers
	$\mid t = t$	equality
	$\mid P(t, \dots, t)$	predicate

Abbreviations

$$\neg A \stackrel{\text{def}}{=} A \supset \bot \qquad \text{negation}$$

$$t \neq t' \stackrel{\text{def}}{=} \neg(t = t') \qquad \text{inequality}$$

Formalising First-Order Logic

Using Agda as an logical framework

- Edinburgh Logical Framework (LF) approach

  We postulate each logical constant as a type former, and each axiom and inference rule as a constants of the corresponding type.

Formalising First-Order Logic

Using Agda as an logical framework

- Edinburgh Logical Framework (LF) approach

  We postulate each logical constant as a type former, and each axiom and inference rule as a constants of the corresponding type.

- Basic inductive approach

  The introduction rules of the logical constants are represented by inductive types, and their elimination rules are defined by pattern matching.

Formalising First-Order Logic

Using Agda as an logical framework

- Edinburgh Logical Framework (LF) approach

  We postulate each logical constant as a type former, and each axiom and inference rule as a constants of the corresponding type.

- Basic inductive approach

  The introduction rules of the logical constants are represented by inductive types, and their elimination rules are defined by pattern matching.

- Inductive approach

  To make full use of Agda's support for proof by pattern matching, we shall allow proofs by pattern matching in general (not only for the elimination rules), as long as they are accepted by Agda's coverage and termination checker.

Example (existential quantifier)

$$\frac{A(t)}{\exists x.A(x)}\ (\exists \mathrm{I}) \qquad \frac{\exists x.A(x) \qquad \begin{matrix}[A]\\ \vdots \\ B\end{matrix}}{B}\ (\exists \mathrm{E})$$

(side condition for the rule $\exists\mathrm{E}$: x is not free in B or in any of the assumptions of the proof of B other than $A(x)$)

Formalising First-Order Logic

Example (existential quantifier (cont.))

LF- and inductive approaches

Domain of quantification

```
postulate D : Set
```

Example (existential quantifier (cont.))

LF- and inductive approaches

Domain of quantification

```
postulate D : Set
```

LF-approach

```
postulate
  ∃      : (A : D → Set) → Set
  _,_    : {A : D → Set}(t : D) →
           A t → ∃ A
  ∃-elim : {A : D → Set}{B : Set} →
           ∃ A →
           (∀ {x} → A x → B) → B
```

Formalising First-Order Logic

Example (existential quantifier (cont.))

LF- and inductive approaches

Domain of quantification

```
postulate D : Set
```

LF-approach

```
postulate
  ∃       : (A : D → Set) → Set
  _,_     : {A : D → Set}(t : D) →
            A t → ∃ A
  ∃-elim : {A : D → Set}{B : Set} →
            ∃ A →
            (∀ {x} → A x → B) → B
```

Inductive approaches

```
data ∃ (A : D → Set) : Set where
  _,_ : (t : D) → A t → ∃ A

∃-elim : {A : D → Set}{B : Set} →
         ∃ A → (∀ {x} → A x → B) → B
∃-elim (_ , Ax) h = h Ax
```

Formalising First-Order Logic

Notation: It is possible to replace `∃ (λ x → e)` by `∃[ x ] e`.

Formalising First-Order Logic

Notation: It is possible to replace `∃ (λ x → e)` by `∃[ x ] e`.

Example

Let $A(x, y)$ be a propositional function. The proof of

$$\exists x.\forall y.A(x, y) \supset \forall y.\exists x.A(x, y),$$

is represented as follows.

Formalising First-Order Logic

Notation: It is possible to replace `∃ (λ x → e)` by `∃[ x ] e`.

Example

Let $A(x, y)$ be a propositional function. The proof of

$$\exists x.\forall y.A(x, y) \supset \forall y.\exists x.A(x, y),$$

is represented as follows.

The theorem:

```
∃∀ : {A : D → D → Set} → ∃[ x ](∀ y → A x y) → ∀ y → ∃[ x ] A x y
```

Formalising First-Order Logic

Notation: It is possible to replace `∃ (λ x → e)` by `∃[ x ] e`.

Example

Let $A(x, y)$ be a propositional function. The proof of

$$\exists x . \forall y . A(x, y) \supset \forall y . \exists x . A(x, y),$$

is represented as follows.

The theorem:

```
∃∀ : {A : D → D → Set} → ∃[ x ](∀ y → A x y) → ∀ y → ∃[ x ] A x y
```

LF- and basic inductive approach proof:

```
∃∀ h y = ∃-elim h (λ {x} ah → x , ah y)
```

Formalising First-Order Logic

Notation: It is possible to replace `∃ (λ x → e)` by `∃[ x ] e`.

Example

Let $A(x, y)$ be a propositional function. The proof of

$$\exists x.\forall y.A(x, y) \supset \forall y.\exists x.A(x, y),$$

is represented as follows.

The theorem:

```
∃∀ : {A : D → D → Set} → ∃[ x ](∀ y → A x y) → ∀ y → ∃[ x ] A x y
```

LF- and basic inductive approach proof:

```
∃∀ h y = ∃-elim h (λ {x} ah → x , ah y)
```

Inductive approach proof:

```
∃∀ (x , Ax) y = x , Ax y
```

Our Representation of First-Order Logic

Falsehood

```
data ⊥ : Set where

⊥-elim : {A : Set} → ⊥ → A
⊥-elim ()
```

Truth

```
data ⊤ : Set where tt : ⊤
```

Disjunction

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

case : ∀ {A B} → {C : Set} → (A → C) → (B → C) → A ∨ B → C
case f g (inj₁ a) = f a
case f g (inj₂ b) = g b
```

Conjunction

```
data _∧_ (A B : Set) : Set where
  _,_ : A → B → A ∧ B

∧-proj₁ : ∀ {A B} → A ∧ B → A
∧-proj₁ (a , _) = a

∧-proj₂ : ∀ {A B} → A ∧ B → B
∧-proj₂ (_ , b) = b
```

Our Representation of First-Order Logic

Conditional

`A → B` (non-dependent function type)

Negation

```
¬_ : Set → Set
¬ A = A → ⊥
```

Principle of the excluded middle

```
postulate pem : ∀ {A} → A ∨ ¬ A
```

Domain of discourse

```
postulate D : Set
```

Universal quantifier

`(x : D) → A` (dependent function type)

Existential quantifier

```
data ∃ (A : D → Set) : Set where
  _,_ : (t : D) → A t → ∃ A

∃-elim : {A : D → Set}{B : Set} →
         ∃ A → (∀ {x} → A x → B) → B
∃-elim (_ , Ax) h = h Ax
```

Equality

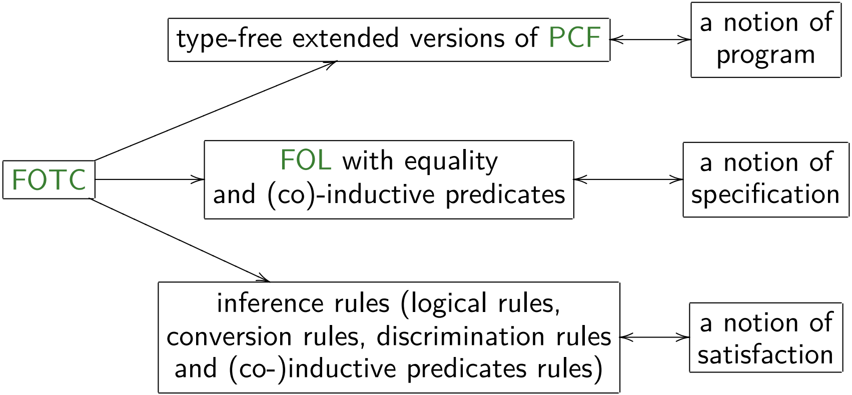
```
data _≡_ (x : D) : D → Set where refl : x ≡ x

subst : (A : D → Set) → ∀ {x y} → x ≡ y →
        A x → A y
subst A refl Ax = Ax
```

First-Order Theory of Combinators (FOTC)

The FOTC programming logic

We extended and formalised Dybjer's (1985) Logical Theory of Constructions for extended versions of PCF.



The Programming Language of FOTC

FOTC-terms

$t ::= x$	variable
$\mid t \cdot t$	application
$\mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}$	partial Boolean constants
$\mid 0 \mid \mathsf{succ} \mid \mathsf{pred} \mid \mathsf{iszero}$	partial natural number constants
$\mid \mathsf{f}$	function constant

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

$$\mathsf{f} \cdot x_1 \cdot \dots \cdot x_n = e[\mathsf{f}, x_1, \dots, x_n].$$

The Specification Language of FOTC

FOTC-formulae

$A ::= \top \mid \bot$	truth, falsehood
$\mid A \supset A \mid A \wedge A \mid A \vee A$	binary logical connectives
$\mid \forall x.A \mid \exists x.A$	quantifiers
$\mid t = t$	equality
$\mid P(t, \dots, t)$	predicate
$\mid \mathcal{B}ool(t)$	total Booleans predicate
$\mid \mathcal{N}(t)$	total natural numbers predicate

The Specification Language of FOTC

Inductive predicates

$\mathcal{B}ool$ and $\mathcal{N}$:	unary inductive predicate symbols
$\mathcal{B}ool(t)$:	t is a total and finite Boolean value (true or false)
$\mathcal{N}(t)$:	t is a total and finite natural number

The Specification Language of FOTC

Inductive predicates

$\mathcal{B}ool$ and $\mathcal{N}$:	unary inductive predicate symbols
$\mathcal{B}ool(t)$:	t is a total and finite Boolean value (true or false)
$\mathcal{N}(t)$:	t is a total and finite natural number

Example

We express that a function f terminates and it maps a total and finite natural number to a total and finite natural number by the formula

$$\forall t.\, \mathcal{N}(t) \supset \mathcal{N}(f \cdot t).$$

Conversion and Discrimination Rules of FOTC

Conversion rules

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

Discrimination rules for constructors

$$\mathsf{true} \neq \mathsf{false},$$
$$\text{and } \forall t.\ 0 \neq \mathsf{succ} \cdot t.$$

The Inductive Predicate Rules of FOTC

Introduction and elimination rules for the inductive predicates $\mathcal{B}ool$ and $\mathcal{N}$

$$\frac{}{\mathcal{B}ool(\mathsf{true})} \qquad \frac{}{\mathcal{B}ool(\mathsf{false})} \qquad \frac{\mathcal{B}ool(t) \qquad A(\mathsf{true}) \qquad A(\mathsf{false})}{A(t)}$$

$$\frac{}{\mathcal{N}(0)} \qquad \frac{\mathcal{N}(t)}{\mathcal{N}(\mathsf{succ} \cdot t)} \qquad \frac{\mathcal{N}(t) \qquad A(0) \qquad \begin{matrix}[A(t')] \\ \vdots \\ A(\mathsf{succ} \cdot t')\end{matrix}}{A(t)}$$

Inductive Representation of FOTC

FOTC-terms

The domain universe and the term constructors are formalised by the following postulates:

```
postulate
  D                    : Set
  _·_                  : D → D → D
  true false if        : D
  zero succ pred iszero : D
```

Inductive Representation of FOTC

Conversion rules

The conversion rules are formalised by the following postulates:

```
postulate
  if-true  : ∀ t {t'} → if · true · t · t' ≡ t
  if-false : ∀ {t} t' → if · false · t · t' ≡ t'
  pred-0   : pred · zero ≡ zero
  pred-S   : ∀ n → pred · (succ · n) ≡ n
  iszero-0 : iszero · zero ≡ true
  iszero-S : ∀ n → iszero · (succ · n) ≡ false
```

Inductive Representation of FOTC

Conversion rules

The conversion rules are formalised by the following postulates:

```
postulate
  if-true  : ∀ t {t'} → if · true · t · t' ≡ t
  if-false : ∀ {t} t' → if · false · t · t' ≡ t'
  pred-0   : pred · zero ≡ zero
  pred-S   : ∀ n → pred · (succ · n) ≡ n
  iszero-0 : iszero · zero ≡ true
  iszero-S : ∀ n → iszero · (succ · n) ≡ false
```

Discrimination rules

The discrimination rules are formalised by the following postulates:

```
postulate
  t≢f : true ≢ false
  0≢S : ∀ {n} → zero ≢ succ · n
```

Inductive Representation of FOTC

Classical predicate logic with equality

We use the inductive representation of FOL for representing the classical predicate logic of FOTC.

Inductive Representation of FOTC

Inference rules for the total and finite natural numbers predicate

The inductive predicate $\mathcal{N}$ is represented as an inductive family:

```
data N : D → Set where
  nzero : N zero
  nsucc : ∀ {n} → N n → N (succ · n)
```

Inductive Representation of FOTC

Inference rules for the total and finite natural numbers predicate

The inductive predicate $\mathcal{N}$ is represented as an inductive family:

```
data N : D → Set where
  nzero : N zero
  nsucc : ∀ {n} → N n → N (succ · n)
```

We define the elimination rule for $\mathcal{N}$ by pattern matching:

```
N-ind : (A : D → Set) →
        A zero →
        (∀ {n} → A n → A (succ · n)) →
        ∀ {n} → N n → A n
N-ind A A0 h nzero      = A0
N-ind A A0 h (nsucc Nn) = h (N-ind A A0 h Nn)
```

Convention

Instead of using the constants `if`, `succ`, `pred` and `iszero` of type `D`, we define more readable and writable function symbols of the appropriate types.

```
if_then_else_ : D → D → D → D
if b then t else t' = if · b · t · t'

succ₁ : D → D
succ₁ n = succ · n

pred₁ : D → D
pred₁ n = pred · n

iszero₁ : D → D
iszero₁ n = iszero · n
```

Example (addition is terminating)

The addition of total and finite natural numbers terminates.

The recursive equation:

```
postulate
  _+_  : D → D → D
  +-0x : ∀ n → zero + n ≡ n
  +-Sx : ∀ m n → succ₁ m + n ≡ succ₁ (m + n)
```

Proving Properties by Structural Recursion

Example (addition is terminating)

The property:

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

The proof is by pattern matching on the first explicit argument:

Base case:

```
+-N {n = n} nzero Nn = subst N (sym (+-leftIdentity n)) Nn
```

Inductive case:

```
+-N {n = n} (nsucc {m} Nm) Nn =
  subst N (sym (+-Sx m n)) (nsucc (+-N Nm Nn))
```

Representation of Higher-Order Functions in FOTC

Using FOTC binary application symbol

```
_·_ : D → D → D
```

we can represent higher-order functions.

Example

The higher-order function that applies a unary function twice is formalised by the axioms

```
twice : D → D → D
twice f x = f · (f · x)
```

Adding (Co-)Inductive Predicates to FOTC

FOTC is not one first-order theory, but a family of first-order theories

- We work with one FOTC for each verification problem
- The function symbols are determined by the program we want to verify
- The predicate symbols are determined by the (co-)inductively defined predicates we need in our proofs, which can be added to FOTC under certain conditions.

Adding Inductive Predicates to FOTC

The inductively defined predicates might not only be used for representing totality properties.

Example (even predicate)

$$\frac{}{\mathcal{E}ven(0)}\,, \qquad \frac{\mathcal{E}ven(t)}{\mathcal{E}ven(\mathsf{succ} \cdot (\mathsf{succ} \cdot t))}\,,$$

$$\frac{\mathcal{E}ven(t) \qquad A(0) \qquad \begin{array}{c}[A(t')] \\ \vdots \\ A(\mathsf{succ} \cdot (\mathsf{succ} \cdot t'))\end{array}}{A(t)}\,.$$

Adding Inductive Predicates to FOTC

Example (FOTC elements for working with lists)

To use lists we add the following elements:

FOTC-terms

$$\{[\,], \mathsf{cons}, \mathsf{null}, \mathsf{head}, \mathsf{tail}\}.$$

Conversion rules

$$\begin{aligned} \mathsf{null} \cdot [\,] &= \mathsf{true}, \\ \forall t\ ts.\ \mathsf{null} \cdot (\mathsf{cons} \cdot t \cdot ts) &= \mathsf{false}, \\ \forall t\ ts.\ \mathsf{head} \cdot (\mathsf{cons} \cdot t \cdot ts) &= t, \\ \forall t\ ts.\ \mathsf{tail} \cdot (\mathsf{cons} \cdot t \cdot ts) &= ts. \end{aligned}$$

Discrimination rule

$$\forall t\ ts.\ [\,] \neq \mathsf{cons} \cdot t \cdot ts.$$

Adding Inductive Predicates to FOTC

Example (representation of the $\mathcal{L}ist$ predicate)

The unary inductive predicate $\mathcal{L}ist(ts)$ representing that ts is a total and finite list of elements.

```
data List : D → Set where
  lnil  : List []
  lcons : ∀ x {xs} → List xs → List (x ∷ xs)
```

where

```
_∷_ : D → D → D
x ∷ xs = cons · x · xs
```

Remark: It is not necessary to implement the elimination rule of $\mathcal{L}ist$ because we shall use Agda's pattern matching instead.

Adding Co-Inductive Predicates

Example (co-natural numbers)

We implement a co-inductive predicate $\mathcal{C}onat(t)$ representing that t is potentially infinite natural number.

The unary predicate:

```
postulate Conat : D → Set
```

The unfolding rule:

```
postulate
  Conat-out : ∀ {n} → Conat n →
              n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ Conat n')
```

The co-induction rule:

```
postulate
  Conat-coind : (A : D → Set) →
                (∀ {n} → A n →
                  n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ A n')) →
                ∀ {n} → A n → Conat n
```

Adding Co-Inductive Predicates

Example (streams)

We implement a co-inductive predicate representing potentially infinite list.

The unary predicate:

```
postulate Stream : D → Set
```

The unfolding rule:

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

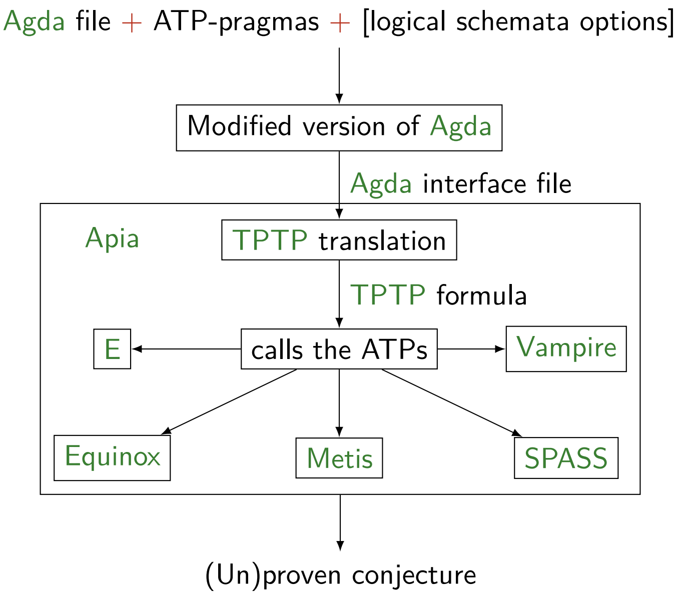
The co-induction rule:

```
postulate
  Stream-coind : (A : D → Set) →
                 (∀ {xs} → A xs →
                   ∃[ x' ] ∃[ xs' ] xs ≡ x' ∷ xs' ∧ A xs') →
                 ∀ {xs} → A xs → Stream xs
```

Combining Interactive and Automatic Proofs

- The verification of lazy functional programs requires the use of simple equational reasoning or simple first-order reasoning (low level reasoning)
- Much of this low-level reasoning can be done automatically with the help of, for example, automatic theorem provers for FOL
- By staying strictly within FOL, we shall be able to employ powerful ATPs for reasoning about functional programs

Extended Version of Agda, Apia and ATPs



The TPTP Language

In TPTP syntax, each problem contains one or more annotated formulae of the form

```
fof(name, role, formula)
```

where `name` identifies the formula within the problem, `formula` is a FOL-formula and `role` can be:

conjectures:	formulae to be proved
axioms:	formulae without proofs
hypotheses:	formulae assumed to be true
definitions:	formulae used to introduce symbols

Using the ATP-Pragma

ATP axioms

We tell the ATPs that the formulae `A`, `B` and `C` are axioms by

```
{-# ATP axiom A B C #-}
```

Using the ATP-Pragma

ATP axioms

We tell the ATPs that the formulae `A`, `B` and `C` are axioms by

```
{-# ATP axiom A B C #-}
```

ATP conjectures

To automatically prove a formula `A`, we shall postulate it and add the ATP-pragma

```
{-# ATP prove A #-}
```

that instructs the ATPs to prove the conjecture `A`.

Using the Apia Program

Example (commutativity of disjunction)

1. Postulating the property

```
postulate
  A B    : Set
  ∨-comm : A ∨ B → B ∨ A
```

Using the Apia Program

Example (commutativity of disjunction)

1. Postulating the property

```
postulate
  A B    : Set
  ∨-comm : A ∨ B → B ∨ A
```

2. Adding the ATP-pragma

```
{-# ATP prove ∨-comm #-}
```

Using the Apia Program

Example (commutativity of disjunction)

1. Postulating the property

```
postulate
  A B    : Set
  ∨-comm : A ∨ B → B ∨ A
```

2. Adding the ATP-pragma

```
{-# ATP prove ∨-comm #-}
```

3. Type-checking the program using Agda

```
$ agda CommDisjunction.agda
```

Using the Apia Program

Example (commutativity of disjunction)

1. Postulating the property

```
postulate
  A B    : Set
  ∨-comm : A ∨ B → B ∨ A
```

2. Adding the ATP-pragma

```
{-# ATP prove ∨-comm #-}
```

3. Type-checking the program using Agda

```
$ agda CommDisjunction.agda
```

4. Proving the conjecture using Apia

```
$ apia CommDisjunction.agda

Proving the conjecture in /tmp/CommDisjunction/10-8744-comm.tptp
Vampire 0.6 (revision 903) proved the conjecture
```

Using the Apia Program

Some command-line options

```
$ apia --help
Usage: apia [OPTIONS] FILE
          --atp=NAME     Set the ATP (e, equinox, ileancop, metis,
                         spass, vampire)
                         (default: e, equinox, and vampire).
          --dump-agdai   Dump the Agda interface file to stdout.
          --only-files   Do not call the ATPs, only to create the
                         TPTP files.
          --time=NUM     Set timeout for the ATPs in seconds
                         (default: 240).
```

Trust of our Approach

- We use the ATPs as oracles via the Apia program
- The user must:
  i) to add to the Agda program the required ATP-pragmas,
  ii) to run the Apia program on the corresponding Agda file and
  iii) to verify that some ATP could prove the formula.
- Implementation of the ATPs
- Implementation of Apia

Automatic Proofs in (Classical) First-Order Logic

Example (the principle of the exclude middle)

```
postulate pem : ∀ {A} → A ∨ ¬ A
{-# ATP prove pem #-}
```

Example (principle of the indirect proof)

```
postulate ¬-elim : ∀ {A} → (¬ A → ⊥) → A
{-# ATP prove ¬-elim #-}
```

General methodology

We informing the ATPs that:

1. The conversion and discrimination rules associated with the FOTC-terms are ATP axioms

General methodology

We informing the ATPs that:

1. The conversion and discrimination rules associated with the FOTC-terms are ATP axioms
2. Each new added recursive equation is an ATP axiom. For example,

```
postulate
  _+_  : D → D → D
  +-0x : ∀ n → zero + n ≡ n
  +-Sx : ∀ m n → succ₁ m + n ≡ succ₁ (m + n)
{-# ATP axiom +-0x +-Sx #-}
```

General methodology

We informing the ATPs that:

3. The inductive data type constructors of the inductive predicates are ATP axioms. For example,

```
data N : D → Set where
  nzero : N zero
  nsucc : ∀ {n} → N n → N (succ₁ n)
{-# ATP axiom nzero nsucc #-}
```

General methodology

We informing the ATPs that:

3. The inductive data type constructors of the inductive predicates are ATP axioms. For example,

```
data N : D → Set where
  nzero : N zero
  nsucc : ∀ {n} → N n → N (succ₁ n)
{-# ATP axiom nzero nsucc #-}
```

4. The unfolding rule of the co-inductive predicate is an ATP axiom. For example,

```
  Conat-out : ∀ {n} → Conat n →
              n ≡ zero ∨ (∃[ n' ] n ≡ succ₁ n' ∧ Conat n')
{-# ATP axiom Conat-out #-}
```

Combined Inductive Proofs in FOTC

Example (addition is terminating)

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

The proof is by pattern matching on the first explicit argument.

Base case:

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

Inductive case:

```
+-N {n = n} (nsucc {m} Nm) Nn = prf (+-N Nm Nn)
  where postulate prf : N (m + n) → N (succ₁ m + n)
        {-# ATP prove prf #-}
```

Combined Co-Inductive Proofs in FOTC

Example (The map-iterate property)

The `map-iterate` property is a common example to illustrate the use of co-induction.

Example (The map-iterate property)

The `map-iterate` property is a common example to illustrate the use of co-induction.

First-order versions of the `map` and `iterate` functions.

```
postulate
  map    : D → D → D
  map-[] : ∀ f → map f [] ≡ []
  map-∷  : ∀ f x xs → map f (x ∷ xs) ≡ f · x ∷ map f xs.
{-# ATP axiom map-[] map-∷ #-}

postulate
  iterate    : D → D → D
  iterate-eq : ∀ f x → iterate f x ≡ x ∷ iterate f (f · x)
{-# ATP axiom iterate-eq #-}
```

Combined Co-Inductive Proofs in FOTC

Example (The map-iterate property)

The bisimilarity relation (equality between potentially infinite terms).

```
postulate
  _≈_ : D → D → Set

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

  ≈-coind :
    (B : D → D → Set) →
    (∀ {xs ys} → B xs ys →
      ∃[ x' ] ∃[ xs' ] ∃[ ys' ]
        xs ≡ x' ∷ xs' ∧ ys ≡ x' ∷ ys' ∧ B xs' ys') →
    ∀ {xs ys} → B xs ys → xs ≈ ys
{-# ATP axiom ≈-out #-}
```

Combined Co-Inductive Proofs in FOTC

Example (The map-iterate property)

The `map-iterate` property asserts that the potentially infinite lists `map f (iterate f x)` and `iterate f (f · x)` are equals.

To prove the `map-iterate` property, we use the `≈-coind` rule on a particular bisimulation `B` (Giménez and Casterán 2007), and the hypotheses required by `≈-coind` are automatically proved by the ATPs.

Combined Co-Inductive Proofs in FOTC

Example (The map-iterate property)

```
≈-map-iterate : ∀ f x → map f (iterate f x) ≈ iterate f (f · x)
≈-map-iterate f x = ≈-coind B h₁ h₂
  where
  B : D → D → Set
  B xs ys =
    ∃[ y ] xs ≡ map f (iterate f y) ∧ ys ≡ iterate f (f · y)
  {-# ATP definition B #-}

  postulate
    h₁ : ∀ {xs ys} → B xs ys → ∃[ x' ] ∃[ xs' ] ∃[ ys' ]
           xs ≡ x' ∷ xs' ∧ ys ≡ x' ∷ ys' ∧ B xs' ys'
  {-# ATP prove h₁ #-}

  postulate h₂ : B (map f (iterate f x)) (iterate f (f · x))
  {-# ATP prove h₂ #-}.
```

Apia Implementation

- Using Agda as a Haskell library

  Working with a not stable API (Agda is a research system)

Apia Implementation

- Using Agda as a Haskell library

  Working with a not stable API (Agda is a research system)

- Agda η-contraction

  Agda performs η-contraction in the internal representation of their types. For example, the Agda internal representation of the following types are the same

```
t  : ∀ d → ∃[ e ] d ≡ e
t' : ∀ d → ∃ (_≡_ d).
```

  Since there is no notion of η-contraction in first-order theories, the Apia program performs an η-expansion on the Agda internal types.

Apia Implementation

- Erasing proof terms

  Since there is no notion of proof term in FOL, it is necessary to erase the proof terms when translating the Agda types into TPTP.
  In the translation of

  ```
  nsucc : ∀ {n} → (Nn : N n) → N (succ₁ n)
  ```

  the Apia programs erase the proof term `Nn`.

Apia Implementation

- Erasing proof terms

  Since there is no notion of proof term in FOL, it is necessary to erase the proof terms when translating the Agda types into TPTP.
  In the translation of

  ```
  nsucc : ∀ {n} → (Nn : N n) → N (succ₁ n)
  ```

  the Apia programs erase the proof term `Nn`.

- Parallel ATPs invocation

  From our experiments, we can conclude that the ATPs we use are complementary that is, where one ATP succeed, other ATPs fail, and the other way around.

The Automatic Theorem Provers

The overall performance of the ATPs in our formalisation of first-order theories is quite satisfactory.

ATP (total theorems: 855)	**Proven thms**	**Unproven thms**	**% Success**
E 1.8-001 Gopaldhara	828	27	97%
Vampire 0.6 (revision 903)	828	27	97%
Equinox 5.0 alpha (2010-06-29)	775	80	91%
SPASS 3.7	755	100	88%
Metis 2.3 (release 2012-09-27)	588	267	69%

Verification of Lazy Functional Programs

We illustrate our approach with some examples where we verify some general (co-)recursive programs and properties.

- Non-structural recursive functions
- Nested recursive functions
- Higher-order recursive functions
- Functions without a termination proof
- Unguarded co-recursive functions (e.g. verification of the alternating bit protocol)

Remark: None of the above examples can be directly formalised in Agda or Coq (they do not pass the termination checker).

Mirror: A Higher-Order Recursive Function

We prove that the `mirror` function for general trees (tree structures with an arbitrary branching) is an involution.

Mirror: A Higher-Order Recursive Function

We prove that the `mirror` function for general trees (tree structures with an arbitrary branching) is an involution.

We extend the FOTC-terms with a constructor for trees

```
postulate node : D → D → D
```

We mutually define predicates for total and finite trees and forests

```
data Forest where
  fnil  : Forest []
  fcons : ∀ {t ts} → Tree t → Forest ts → Forest (t ∷ ts)

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

ATP axioms

```
{-# ATP axiom fnil fcons tree #-}
```

Mirror: A Higher-Order Recursive Function

The mirror function

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

ATP axiom

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

The property

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

The proof is by pattern matching on the mutually defined totality predicates for trees and forests.

Mirror: A Higher-Order Recursive Function

The proof

Base case:

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

Inductive case:

```
mirror-involutive (tree d (fcons {t} {ts} Tt Fts)) = prf
  where
  postulate
    prf : mirror · (mirror · node d (t ∷ ts)) ≡ node d (t ∷ ts)
 {-# ATP prove prf helper #-}
```

The local hypothesis `helper` follows by induction on forests:

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

Conclusions

- We defined FOTC, a first-order programming logic suitable for reasoning about mainstream lazy functional programs including those that use general recursion
- We chose a mature system as our interactive proof assistant to formalise our programming logic. We use Agda's proof engine for writing our proofs and we use it as logical framework.
- To deal with low level reasoning (equational reasoning and first-order reasoning), we used off-the-shelf ATPs
  - We extended Agda with the ATP-pragma
  - We wrote the Apia program which translated our Agda representation of first-order formulae into the TPTP and it calls the ATPs to try to prove the translated conjectures

Future Work

- Proof term reconstruction

  We would like to modify our Apia program so that it can return witnesses for the automatically generated proofs so that they can be checked by Agda.

- Polymorphism

  We need to support polymorphism if we want to deal with a larger fragment of Haskell-like languages.

- Connection to Satisfiability Modulo Theories (SMT) solvers

  An interesting improvement to our Apia program would be to integrate SMT solvers into it.

Thanks!

References

Abel, Andreas, Coquand, Thierry and Norell, Ulf (2005). Connecting a Logical Framework to a First-Order Logic Prover. In: Frontiers of Combining Systems (FroCoS 2005). Ed. by Gramlich, B. Vol. 3717. Lecture Notes in Artifical Intellingence. Springer, pp. 285–301.

Bove, Ana, Krauss, Alexander and Sozeau, Mattieu (2012). Partiality and Recursion in Interactive Theorem Provers. An Overview. Accepted for publication at Mathematical Structures in Computer Science, special issue on DTP 2010.

Dybjer, Peter (1985). Program Verification in a Logical Theory of Constructions. In: Functional Programming Languages and Computer Architecture. Ed. by Jouannaud, Jean-Pierre. Vol. 201. Lecture Notes in Computer Science. Springer, pp. 334–349.

——— (1990). Comparing Integrated and External Logics of Functional Programs. Science of Computer Programming 14, pp. 59–79.

References

Dybjer, Peter and Sander, Herbert P. (1989). A Functional Programming Approach to the Specification and Verification of Concurrent Systems. Formal Aspects of Computing 1, pp. 303–319.

Giménez, Eduardo and Casterán, Pierre (2007). A Tutorial on [Co-]Inductive Types in Coq. URL: http://coq.inria.fr/documentation (visited on 29/07/2014).

Sutcliffe, Geoff (2009). The TPTP Problem Library and Associated Infrastructure. The FOT and CNF Parts, v3.5.0. Journal of Automated Reasoning 43.4, pp. 337–362.