Reasoning about Functional Programs by Combining Interactive and Automatic Proofs

Andrés Sicard-Ramírez[1]
(joint work with Ana Bove[2] and Peter Dybjer[2])

[1]EAFIT University, Colombia

[2]Chalmers University of Technology, Sweden

Seminar of the PhD in Mathematical Engineering
EAFIT University
8 September 2014

Our Goal

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

Some Paradigms of Programming

Imperative: Describe computation in terms of state-transforming operations such as assignment. Programming is done with statements.

Logic: Predicate calculus as a programming language. Programming is done with sentences.

Functional: Describe computation in terms of (mathematical) functions. Programming is done with expressions.

Some Paradigms of Programming

Imperative: Describe computation in terms of state-transforming operations such as assignment. Programming is done with statements.

Logic: Predicate calculus as a programming language. Programming is done with sentences.

Functional: Describe computation in terms of (mathematical) functions. Programming is done with expressions.

Examples

$$\text{Imperative}\begin{cases} \mathrm{C} \\ \mathrm{C}++ \\ \mathrm{Java} \end{cases} \quad \text{Logic}\begin{cases} \mathrm{CLP(R)} \\ \mathrm{Prolog} \end{cases} \quad \text{Functional}\begin{cases} \text{Standard ML} \\ \text{Erlang} \\ \text{Pure}\begin{cases} \text{Clean} \\ \text{Haskell} \\ \text{Idris} \end{cases} \end{cases}$$

Side effects

"A side effect introduces a dependency between the global state of the system and the behaviour of a function ... Side effects are essentially invisible inputs to, or outputs from, functions."[1]

[1] O'Sullivan, Bryan, John Goerzen and Don Stewart (2008). Real World Haskell, p. 27.

[2] Hutton, Graham (2007). Programming in Haskell, p. 87.

Side effects

"A side effect introduces a dependency between the global state of the system and the behaviour of a function … Side effects are essentially invisible inputs to, or outputs from, functions."[1]

Pure functions

In Haskell all the functions are pure functions, i.e. they "take all their input as explicit arguments, and produce all their output as explicit results."[2]

[1] O'Sullivan, Bryan, John Goerzen and Don Stewart (2008). Real World Haskell, p. 27.
[2] Hutton, Graham (2007). Programming in Haskell, p. 87.

Haskell: A Pure Functional Programming Language

Side effects

"A side effect introduces a dependency between the global state of the system and the behaviour of a function ... Side effects are essentially invisible inputs to, or outputs from, functions."[1]

Pure functions

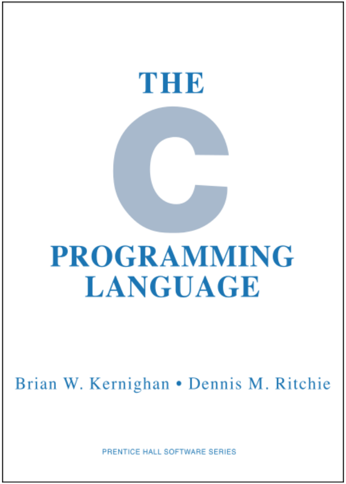
In Haskell all the functions are pure functions, i.e. they "take all their input as explicit arguments, and produce all their output as explicit results."[2]

Referential transparency

Equals can be replaced by equals.

[1] O'Sullivan, Bryan, John Goerzen and Don Stewart (2008). Real World Haskell, p. 27.
[2] Hutton, Graham (2007). Programming in Haskell, p. 87.

Haskell: A Pure Functional Programming Language



"The first program to write is the same for all languages: Print the words `hello, world`." (1978, §1.1)

Example

The following C program prints "hello, world" twice.

```c
#include <stdio.h>

int
main (void)
{
  printf ("hello, world");
  printf ("hello, world");

  return 0;
}
```

Haskell: A Pure Functional Programming Language

Example

The following C program prints "hello, world" once.

```
#include <stdio.h>

int
main (void)
{
  int x;

  x = printf ("hello, world");
  x; x;

  return 0;
}
```

Example (Lists)

Haskell has built-in syntax for lists, where a list is either:

- the empty list, written `[]`, or
- a first element `x` and a list `xs`, written `length (x : xs)`.

Haskell: A Pure Functional Programming Language

Example (Lists)

Haskell has built-in syntax for lists, where a list is either:

- the empty list, written `[]`, or
- a first element `x` and a list `xs`, written `length (x : xs)`.

Example (Pattern matching on lists)

```
length :: [Int] → Int
length []       = 0
length (x : xs) = 1 + length xs
```

Example (Parametric polymorphism)

```
length :: [a] → Int
length []       = 0
length (x : xs) = 1 + length xs
```

Lazy evaluation

Nothing is evaluated until necessary.

Lazy evaluation

Nothing is evaluated until necessary.

Example

```haskell
take :: [Int] → [a] → [a]
```

```haskell
squares :: [Int]
squares = [x ^ 2 | x ← [1..]]
```

Lazy evaluation

Nothing is evaluated until necessary.

Example

```
take :: [Int] → [a] → [a]
```

```
squares :: [Int]
squares = [x ^ 2 | x ← [1..]]
```

Which is the value of `take 5 squares`? [1,4,9,16,25]

Question

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

[3] Bove, Ana, Alexander Krauss and Mattieu Sozeau (2012). Partiality and Recursion in Interactive Theorem Provers. An Overview.

Question

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)

Remark: Most of the proof assistants lack a direct treatment for general recursive functions.[3]

[3] Bove, Ana, Alexander Krauss and Mattieu Sozeau (2012). Partiality and Recursion in Interactive Theorem Provers. An Overview.

Programming logic

A logic in which programs and specifications can be expressed and in which it can be proved or disproved that a certain program meets a certain specification.

Proof Assistants

Proof assistant

An interactive computer system which helps with the development of formal proofs.

Proof Assistants

Proof assistant

An interactive computer system which helps with the development of formal proofs.

Examples (incomplete list)

Name	Version	Language	Logic	Dependent types
Agda	2.4.2 (Aug. 2014)	Haskell	Type theory	Yes
Coq	8.4pl4 (May 2014)	OCaml	Type theory	Yes
Isabelle	Isabelle2014 (Aug.)	Standard ML	Higher-order logic	No

Automatising First-Order Logic Proofs

Automatic theorem provers for first-order logic (ATPs)

- TPTP: a language understood by many off-the-shelf ATPs
- The TPTP world: `http://www.cs.miami.edu/~tptp/`
- The CADE ATP System Competition

Our Main Contributions

1. What programming logic should we use?

Our Main Contributions

1. What programming logic should we use?

We defined and formalised the First-Order Theory of Combinators:

Our Main Contributions

1. What programming logic should we use?

We defined and formalised the First-Order Theory of Combinators:

- Programs: Type-free extended versions of Plotkin's PCF language

Our Main Contributions

1. What programming logic should we use?

We defined and formalised the First-Order Theory of Combinators:

- Programs: Type-free extended versions of Plotkin's PCF language
- Specification language: First-order logic and predicates representing the property of being a finite or a potentially infinite value

Our Main Contributions

1. What programming logic should we use?

We defined and formalised the First-Order Theory of Combinators:

- Programs: Type-free extended versions of Plotkin's PCF language
- Specification language: First-order logic and predicates representing the property of being a finite or a potentially infinite value
- Inference rules: Conversion and discrimination rules for the term language, introduction and elimination for the (co)-inductive predicates

Our Main Contributions

2. What proof assistant should we use?

Our Main Contributions

2. What proof assistant should we use?

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

Our Main Contributions

2. What proof assistant should we use?

We formalise our programming logic 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

Our Main Contributions

2. What proof assistant should we use?

We formalise our programming logic 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

3. Can (part of) the job be automatic?

Our Main Contributions

3. Can (part of) the job be automatic?

Yes! We can combine Agda interactive proofs and ATPs:

Our Main Contributions

3. Can (part of) the job be automatic?

Yes! We can combine Agda interactive proofs and ATPs:

- we provide a translation of our Agda representation of first-order formulae into TPTP so we can use them when proving the properties of our programs,

Our Main Contributions

3. Can (part of) the job be automatic?

Yes! We can combine Agda interactive proofs and ATPs:

- we provide a translation of our Agda representation of first-order formulae into TPTP 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

Our Main Contributions

3. Can (part of) the job be automatic?

Yes! We can combine Agda interactive proofs and ATPs:

- we provide a translation of our Agda representation of first-order formulae into TPTP 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.

Related Publications

- Andrés Sicard-Ramírez (2014). Reasoning about Functional Programs by Combining Interactive and Automatic Proofs. PhD thesis. Universidad de la República, Uruguay. In preparation.
- Ana Bove, Peter Dybjer and Andrés Sicard-Ramírez (2012). Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs. FoSSaCS 2012.
- Ana Bove, Peter Dybjer and Andrés Sicard-Ramírez (2009). Embedding a Logical Theory of Constructions in Agda. PLPV 2009.

Source Codes

The programs and examples described are available as Git repositories at GitHub:

- The extended version of Agda: `https://github.com/asr/eagda`.
- The Apia program: `https://github.com/asr/apia`.
- The Agda implementation of our programming logics, some first-order theories and examples of verification of functional programs: `https://github.com/asr/fotc`.

Thanks!