Verification of Functional Programs
I. First-Order Theory of Combinators

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

[1]EAFIT University, Colombia

[2]Chalmers University of Technology, Sweden

Logic and Computation Seminar
EAFIT University
31 August 2012

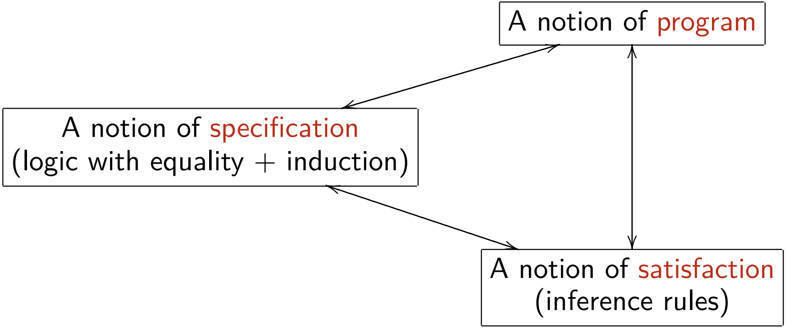
Introduction

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?

1. What programming logic should we use?



Introduction

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

1. What programming logic should we use?
2. What proof assistant should we use?

Introduction

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

1. What programming logic should we use?
2. What proof assistant should we use?
3. Can part of the job be automatic?
    - Can we use automatic theorem provers for first-order logic (ATPs)?
    - Can we use Satisfiability Modulo Theories (SMT) solvers?
    - Can we use inductive theorem provers (ITPs)?

First-Order Theory of Combinators

What programming logic should we use?

We propose the First-Order Theory of Combinators.

First-Order Theory of Combinators

What programming logic should we use?

We propose the First-Order Theory of Combinators.

Features:

1. general recursion (structural, non-structural, nested and higher-order recursion),

First-Order Theory of Combinators

What programming logic should we use?

We propose the First-Order Theory of Combinators.

Features:

1. general recursion (structural, non-structural, nested and higher-order recursion),
2. higher-order functions,

First-Order Theory of Combinators

What programming logic should we use?

We propose the First-Order Theory of Combinators.

Features:

1. general recursion (structural, non-structural, nested and higher-order recursion),
2. higher-order functions,
3. partial and total correctness, and

What programming logic should we use?

We propose the First-Order Theory of Combinators.

Features:

1. general recursion (structural, non-structural, nested and higher-order recursion),
2. higher-order functions,
3. partial and total correctness, and
4. inductive and coinductive predicates.

Haskell: A very large language

Haskell

(represented by a data type with hundreds of constructors)

full translation

Core

(typed λ-calculus with few syntactic forms)

Source: Marlow and Peyton-Jones (2012). "The Glasgow Haskell Compiler".

Plotkin's PCF: A "simple" functional programming language

$\text{Types} \ni \sigma ::= \mathsf{nat}$	natural numbers
$\mid \sigma \to \sigma$	function type
$\text{Terms} \ni t ::= x$	variable
$\mid t\ t$	application
$\mid \lambda x : \sigma .\ t$	λ-abstraction
$\mid \mathsf{fix}_\sigma(t)$	fixed-point operator
$\mid 0$	zero
$\mid \mathsf{succ}(t)$	succesor function
$\mid \mathsf{pred}(t)$	predecessor function
$\mid \mathsf{iszero}(t, t, t)$	conditional

Source: Plotkin (1977). "LCF Considered as a Programming Language".

A Logical Theory of Constructions (LTC) for type-free PCF

History (very incomplete):

1. Aczel (1977). "The Strength of MartinLöf's Intuitionistic Type Theory with One Universe".

A Logical Theory of Constructions (LTC) for type-free PCF

History (very incomplete):

1. Aczel (1977). “The Strength of MartinLöf’s Intuitionistic Type Theory with One Universe”.
2. Dybjer (1985). “Program Verification in a Logical Theory of Constructions”.

A Logical Theory of Constructions (LTC) for type-free PCF

History (very incomplete):

1. Aczel (1977). "The Strength of MartinLöf's Intuitionistic Type Theory with One Universe".
2. Dybjer (1985). "Program Verification in a Logical Theory of Constructions".
3. Bove, Dybjer and Sicard-Ramírez (2009). "Embedding a Logical Theory of Constructions in Agda".

LTC: Terms

$\mathsf{Terms} \ni t ::= x$	variable
$\mid t \cdot t$	application
$\mid \lambda x.\, t$	λ-abstraction
$\mid \mathsf{fix}\ x.\, t$	fixed-point operator
$\mid \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}$	partial Boolean constants
$\mid 0 \mid \mathsf{succ} \mid \mathsf{pred} \mid \mathsf{iszero}$	partial natural number constants
$\mid \mathsf{loop}$	looping constant

LTC: Formulae

$\text{Formulae} \ni A ::= \top \mid \bot$	truth, falsehood
$\mid A \Rightarrow A \mid A \wedge A \mid A \vee A$	binary logical connectives
$\mid \forall x.A \mid \exists x.A$	quantifiers
$\mid t = t$	equality
$\mid P(t, \dots, t)$	predicate
$\mid Bool(t)$	total Booleans predicate
$\mid N(t)$	total natural numbers predicate

LTC: Inference rules

Axioms and axiom schemata of LTC

1. Axioms for the intuitionistic logical constants
2. Conversion rules for the combinators
3. Discrimination rules
4. Introduction and elimination rules for $Bool$ and N

LTC: Conversion and discrimination rules

Conversion rules for the combinators

$$
\begin{aligned}
\forall t\ t'.\ \mathsf{if} \cdot \mathsf{true} \cdot t \cdot t' &= t, \\
\forall t\ t'.\ \mathsf{if} \cdot \mathsf{false} \cdot t \cdot t' &= t', \\
\mathsf{pred} \cdot 0 &= 0, \\
\forall t.\ \mathsf{pred} \cdot (\mathsf{succ} \cdot t) &= t, \\
\mathsf{iszero} \cdot 0 &= \mathsf{true}, \\
\forall t.\ \mathsf{iszero} \cdot (\mathsf{succ} \cdot t) &= \mathsf{false}, \\
\mathsf{loop} &= \mathsf{loop}, \\
\forall t\ t'.\ (\lambda x.\ t) \cdot t' &= t[x := t'], \\
\forall t.\ \mathsf{fix}\ x.\ t &= t[x := \mathsf{fix}\ x.\ t],
\end{aligned}
$$

LTC: Conversion and discrimination rules

Conversion rules for the combinators

$$
\begin{aligned}
\forall t\ t'.\ \mathsf{if}\cdot\mathsf{true}\cdot t\cdot t' &= t,\\
\forall t\ t'.\ \mathsf{if}\cdot\mathsf{false}\cdot t\cdot t' &= t',\\
\mathsf{pred}\cdot 0 &= 0,\\
\forall t.\ \mathsf{pred}\cdot(\mathsf{succ}\cdot t) &= t,\\
\mathsf{iszero}\cdot 0 &= \mathsf{true},\\
\forall t.\ \mathsf{iszero}\cdot(\mathsf{succ}\cdot t) &= \mathsf{false},\\
\mathsf{loop} &= \mathsf{loop},\\
\forall t\ t'.\ (\lambda x.\, t)\cdot t' &= t[x := t'],\\
\forall t.\ \mathsf{fix}\ x.\ t &= t[x := \mathsf{fix}\ x.\ t],
\end{aligned}
$$

Discrimination rules

$$
\begin{aligned}
\mathsf{true} &\neq \mathsf{false},\\
\forall t.\ 0 &\neq \mathsf{succ}\cdot t.
\end{aligned}
$$

LTC: Rules for $Bool$

Introduction and elimination (expressing proof by case analysis on total Boolean values) rules for $Bool$:

$$\frac{}{Bool(\mathsf{true})} \qquad \frac{}{Bool(\mathsf{false})}$$

$$\frac{Bool(t) \qquad A(\mathsf{true}) \qquad A(\mathsf{false})}{A(t)}$$

LTC: Rules for N

Introduction and elimination (expressing proof by mathematical induction) rules for N:

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

$$\frac{N(t) \qquad A(0) \qquad \begin{matrix}[A(t)] \\ \vdots \\ A(\mathsf{succ} \cdot t)\end{matrix}}{A(t)}$$

First-Order Theory of Combinators (FOTC)

Source: Bove, Dybjer and Sicard-Ramírez (2012). "Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs".

- First stage: A first-order theory

First-Order Theory of Combinators (FOTC)

Source: Bove, Dybjer and Sicard-Ramírez (2012). "Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs".

- First stage: A first-order theory
- Second stage: Add of new inductively defined predicates

First-Order Theory of Combinators (FOTC)

Source: Bove, Dybjer and Sicard-Ramírez (2012). "Combining Interactive and Automatic Reasoning in First Order Theories of Functional Programs".

- First stage: A first-order theory
- Second stage: Add of new inductively defined predicates
- Third stage: Add of co-inductively defined predicates

FOTC: A first-order theory

Lambda-lifting

Add a new function symbol for each recursive function definition of the form

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

instead of use the λ-abstraction and the fixed-point operator from LTC.

FOTC: Terms

The grammar for the terms of FOTC is now first order:

$\mathsf{Terms} \ni t ::= x$	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
$\mid \mathsf{loop}$	looping combinator
$\mid f$	function

where f ranges over new combinators defined by recursive equations.

FOTC: Add of new inductively defined predicates

Example

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

$$\frac{Even(t) \qquad A(0) \qquad \begin{array}{c}[A(t)] \\ \vdots \\ A(\mathsf{succ} \cdot \mathsf{succ} \cdot t)\end{array}}{A(t)}$$

FOTC: Add of co-inductively defined predicates

Methodology:

- The inductively defined predicates are defined as the least fixed-point of the operator associated with their introduction rules.

FOTC: Add of co-inductively defined predicates

Methodology:

- The inductively defined predicates are defined as the least fixed-point of the operator associated with their introduction rules.
- The co-inductively defined predicates are defined as the greatest fixed-point of the operator associated with their introduction rules.

Examples of verification

- Non-structural recursion: Program that computes the greatest common divisor of two natural numbers using Euclid's algorithm
- Nested recursion: Properties and termination of McCarthy91 function
- Higher-order recursion: The mirror function for Rose trees
- Co-recursive function: The map-iterate property
- Induction and co-induction: The alternating bit protocol
- A non-terminating function: The Collatz function

Missing topics

- Consistency of LTC
- Characterization of the (co-)inductively generated predicates
- Consistency of FOTC

Associated talks

1. What proof assistant should we use?
   Using Agda as a logical framework for FOTC.

Associated talks

1. What proof assistant should we use?
   Using Agda as a logical framework for FOTC.
2. Can part of the job be automatic?
   `agda2atp`: An Haskell program for proving first-order formulae written in Agda using ATPs, via the translation of the Agda formulae to the TPTP format.

   GitHub repository: https://github.com/asr/fotc.

Associated talks

1. What proof assistant should we use?
   Using Agda as a logical framework for FOTC.
2. Can part of the job be automatic?
   `agda2atp`: An Haskell program for proving first-order formulae written in Agda using ATPs, via the translation of the Agda formulae to the TPTP format.

   GitHub repository: `https://github.com/asr/fotc`.
3. Future work: Theoretical, integration, and/or implementation.
   See `http://www1.eafit.edu.co/asicard/slides/fotc-future-work-slides.pdf`