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

Andrés Sicard-Ramírez\textsuperscript{1}
(joint work with Ana Bove\textsuperscript{2} and Peter Dybjer\textsuperscript{2})

\textsuperscript{1}EAFIT University, Colombia
\textsuperscript{2}Chalmers University of Technology, Sweden

Logic and Computation Seminar
EAFIT University
31 August 2012
Introduciton

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

1. What *programming logic* should we use?

A notion of **program**

A notion of **specification**
(logic with equality + induction)

A notion of **satisfaction**
(inference rules)
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)?
What *programming logic* should we use?

We propose the **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,
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 $\lambda$-calculus with few syntactic forms)


Plotkin’s PCF: A “simple” functional programming language

**Types** ⊆ σ ::= nat  
    | σ → σ  

**Terms** ⊆ t ::= x  
    | tt  
    | λx : σ. t  
    | fix_σ(t)  
    | 0  
    | succ(t)  
    | pred(t)  
    | iszero(t, t, t)  

History (very incomplete):

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

History (very incomplete):

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

History (very incomplete):

LTC: Terms

Terms $\exists t ::= x$

- $t \cdot t$ (application)
- $\lambda x. t$ ($\lambda$-abstraction)
- fix $x. t$ (fixed-point operator)
- true, false, if (partial Boolean constants)
- 0, succ, pred, iszero (partial natural number constants)
- loop (looping constant)
LTC: Formulae

Formulae $\exists A ::= \top \mid \bot$

- $A \Rightarrow A$
- $A \land A$
- $A \lor A$
- $\forall x. A$
- $\exists x. A$
- $t = t$
- $P(t, \ldots, t)$
- $\text{Bool}(t)$
- $N(t)$

truth, falsehood
binary logical connectives
quantifiers
equality
predicate
total Booleans predicate
total natural numbers
predicate
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 $\text{Bool}$ and $N$
Conversion rules for the combinators

\[
\forall t \ t'. \ \text{if} \cdot \text{true} \cdot t \cdot t' = t,
\]

\[
\forall t \ t'. \ \text{if} \cdot \text{false} \cdot t \cdot t' = t',
\]

\[
\text{pred} \cdot 0 = 0,
\]

\[
\forall t. \ \text{pred} \cdot (\text{succ} \cdot t) = t,
\]

\[
\text{iszero} \cdot 0 = \text{true},
\]

\[
\forall t. \ \text{iszero} \cdot (\text{succ} \cdot t) = \text{false},
\]

\[
\text{loop} = \text{loop},
\]

\[
\forall t \ t'. \ (\lambda x. t) \cdot t' = t[x := t'],
\]

\[
\forall t. \ \text{fix} \ x. t = t[x := \text{fix} \ x. t],
\]
Conversion rules for the combinators

\[ \forall t t'. \text{if} \cdot \text{true} \cdot t \cdot t' = t, \]
\[ \forall t t'. \text{if} \cdot \text{false} \cdot t \cdot t' = t', \]
\[ \text{pred} \cdot 0 = 0, \]
\[ \forall t. \text{pred} \cdot (\text{succ} \cdot t) = t, \]
\[ \text{iszero} \cdot 0 = \text{true}, \]
\[ \forall t. \text{iszero} \cdot (\text{succ} \cdot t) = \text{false}, \]
\[ \text{loop} = \text{loop}, \]
\[ \forall t t'. (\lambda x. t) \cdot t' = t[x := t'], \]
\[ \forall t. \text{fix} x. t = t[x := \text{fix} x. t], \]

Discrimination rules

\[ \text{true} \neq \text{false}, \]
\[ \forall t. 0 \neq \text{succ} \cdot t. \]
Introduction and elimination (expressing proof by case analysis on total Boolean values) rules for \( \text{Bool} \):
Introduction and elimination (expressing proof by mathematical induction) rules for $N$:

\[
\begin{array}{c}
\frac{N(0)}{N(t)} \quad \frac{N(t)}{N(\text{succ } t)}
\end{array}
\]

\[
[A(t)]
\]

\[
\begin{array}{c}
\vdots \\
\frac{N(t)}{A(0)} \quad \frac{A(0)}{A(\text{succ } t)} \quad \frac{A(\text{succ } t)}{A(t)}
\end{array}
\]
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
Lambda-lifting

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

\[ f \ x_1 \ldots x_n = e[f, x_1, \ldots, x_n], \]

instead of use the \( \lambda \)-abstraction and the fixed-point operator from LTC.
The grammar for the terms of FOTC is now first order:

**Terms $\ni t ::= x$**

variable

$| t \cdot t$ application

$| \text{true} \mid \text{false} \mid \text{if}$ partial Boolean constants

$| 0 \mid \text{succ} \mid \text{pred} \mid \text{iszero}$ partial natural number constants

$| \text{loop}$ looping combinator

$| f$ function

where $f$ ranges over new combinators defined by recursive equations.
Example

\[
\begin{array}{c}
\text{Even}(0) & \text{Even}(\text{succ} \cdot \text{succ} \cdot t) \\
\hline
\text{Even}(t) & A(0) & A(\text{succ} \cdot \text{succ} \cdot t) \\
\hline
\end{array}
\]

[A(t)]

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

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](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](http://www1.eafit.edu.co/asicard/slides/fotc-future-work-slides.pdf)