Formalizing Program Equivalences in Dependent Type Theory

a non-technical introduction

Alberto Momigliano
DI, Milano

joint work with Giorgio Marabelli

ICTCS, September 11, 2009, Como

Formalizing Math in a Proof Assistant

- Do you believe in your proofs?

  (I don't, but I'm mathematically challenged)
- Some theorems have proofs so complicated that they cannot be inspected/understood by humans (the Four-Color Theorem, the Kepler Conjecture).
- *Computed assisted (interactive) theorem proving* (**Proof Assistants**) have emerged as an answer to this problem and provide formal, inspectable proofs with the highest guarantee of certainty:
  - The Four-Color Theorem, implemented by Werner and Gonthier in **Coq**
  - The Kepler conjecture implemented as *Flyspeck* by Hayes at al., 2013 in **HOL4**

Formalizing CS in a Proof Assistant

- One angle for the use of PA in CS is the mechanized verification of programming languages theory and artifacts:
  - Lambda/objects/process calculi.
  - static analyzers (e.g., type checkers), interpreters, compilers...
- Meta-theoretic properties: type soundness, compiler correctness, termination, program equivalence...
- Those proofs do not require deep math, but, for realistic languages, have so many cases that humans (read PhD students) cannot be trusted. Or such proofs are simply not done in details.
- One of many success stories of verified PL meta-theory:
  - **CompCert** (Leroy 2009, Coq): a fully verified compiler for most of C (ISO C99), generating efficient code for the PowerPC, ARM, etc.

Equivalence of functional programs

- Our model of pure higher-order functional programming is **PCFL**, a λ-calculus with recursion over lazy lists:

$$\begin{array}{llll} \text{Type} & \tau & ::= & \top \mid \tau \to \tau \mid \tau \text{ list} \\ \text{Exp} & m & ::= & x \mid \lambda x.\, m \mid m_1\ m_2 \mid \text{fix } x.\ m \mid \langle\rangle \mid \text{nil} \mid \text{cons } m_1\ m_2 \\ & & & \mid \text{lcase } m \text{ of } \{\text{nil} \Rightarrow m_h \mid \text{cons } h\ t \Rightarrow m_t\} \end{array}$$

  lazy evaluation brings in an interesting notion of **infinite** computation that we can observe

- An example in concrete syntax: those two reverse functions are (Kleene) equivalent:

```
let rec slowrev xs =
  match xs with
   [] -> []
   |y:ys -> slowrev ys @ [y]
```

```
let frev xs =
   let rec aux xs acc =
     match xs with
       [] -> acc
       |y:ys -> aux ys (y:acc)
   aux xs []
```

Basic desiderata for representing PL theory in a PA

A shameless plug: see Krebbers, Pientka and myself's entry *"Programming Languages"* in the forthcoming Springer book *"Proof Assistants and their Applications in Mathematics and Computer Science"*, Blanchette and Mahboubi eds.

- Representing binding syntax adequately
  - These two programs **should** be the same (α-equivalence):

    `let fst x y = x` `let fst y x = y`
- Good infrastructure for ubiquitous notions such as typing contexts, environments, substitutions, renamings etc.
- Support for defining (co)recursive functions over syntax and (co)inductive definitions of judgments such as static/dynamic semantics (SOS) etc.
  - both need to be compatible with the way we represent the syntax — not obvious!

Approaches to syntax

Choosing the right way to represent syntax is paramount to a successful verification effort. Lots to choose from:

- **Raw** terms and variables as strings: inadequate and hellish (still people — Leroy, not to name names — use it all the time)
- **de Bruijn** terms: variables are pointers to their binding site:
  - the standard for meta-programming, but difficult to read and a lot of external machinery for reasoning about it
- **Nominal logic**: consider terms modulo equivariance (i.e., permutations of names):
  - very trendy, but not well supported by PAs (just the encoding on top of Isabelle/HOL, so it's classical logic)
- **Higher-order abstract syntax**: object level binders mapped to the binder of the meta-logic:
  - the best (AFAIK), but not consistent with standard PA

PA: what to choose?

- A specialized framework based HOAS syntax and geared towards PL-theory: **Twelf/Abella/Beluga**

  Pros: great support for binding syntax and related abstractions;

  Cons: few libraries, little automation, no current support for higher-order logic.

PA: what to choose?

- A specialized framework based HOAS syntax and geared towards PL-theory: **Twelf/Abella/Beluga**

  Pros: great support for binding syntax and related abstractions;
  Cons: few libraries, little automation, no current support for higher-order logic.

- A full-fledged, general purpose PA, say **Coq** — the de facto standard (like it or not) — building on appropriate libraries for PL meta-theory

PA: what to choose?

- A specialized framework based HOAS syntax and geared towards PL-theory: **Twelf/Abella/Beluga**

  Pros: great support for binding syntax and related abstractions;
  Cons: few libraries, little automation, no current support for higher-order logic.

- A full-fledged, general purpose PA, say **Coq** — the de facto standard (like it or not) — building on appropriate libraries for PL meta-theory

When studying the meta-theory of program equivalences in higher-order functional programming, we may need more:

More desiderata

- All relations of interest concern **well-typed** terms and we'd like to preserve that invariance automatically (*intrinsic typing*).
  - $\Rightarrow$ Via Coq's dependent types, we can define object terms to depend on object types, so that ill-typed terms cannot even be represented. We use **well-typed, well-scoped De Bruijn terms** (simple modification of the existing Coq theory by Benton at al. (2007)
- A meta-logic general enough to deal with relations as first class citizen:
  - $\Rightarrow$ comes for free from Coq's higher-order logic and from its **type classes** mechanism.
- Sophisticated **coinductive** reasoning — vanilla *guarded induction* not enough
  - $\Rightarrow$ the **PACO** (parameterized coinduction) library by Hur et al. (2013) — basically, reasoning directly via greatest fixed points.

So, Coq it is.

What is a program equivalence?

- Two program expressions (or components if not stand-alone) are semantically equivalent if we cannot tell them apart.
- We do that by running experiments and observe their behavior.
- We identify an experiment with a **context** $\mathcal{C}$, that is a PCFL expression with a *hole* (a logical variable)
- component **passes** a test if when plugged into a context, the resulting whole program converges to an (observable) value:

$$M \equiv_{\text{ctx}} N \text{ iff } \forall \mathcal{C} \forall V, \mathcal{C}[M] \Downarrow V \leftrightarrow \mathcal{C}[N] \Downarrow V$$

- What do we mean by observable?

  Ground We observe convergence only at ground values

  Applicative We observe it at every type (functions are values)

  The former is coarser than the latter.

Why does contextual equivalence matter?

- Compiler optimization, and more in general compiler full abstraction
- interchangeability of abstract data types, even . . .
- . . . security properties (e.g., confidentiality of a variable or more in general *non-interference*) can be expressed in terms of contextual equivalence:

  $\Rightarrow$ An **attacker** can be seen as a **context**
- . . . , see the annual "Workshop on Program Equivalence and Relational Reasoning"

Properties of contextual equivalence

- Easy to show that it is a **congruence**, that is an equivalence relation that it is also **compatible**, i.e., it respects the constructors of the object language — this enables equational reasoning, good.
- Yet, the quantification on **every** context makes it hard to use in practice. The more for low-level languages where the notion of context is **not** inductive.
- Thus, we have to consider other forms of equivalences, provided we can prove them to coincide with ctx equivalence.
- One such from concurrency theory is **bisimilarity**.

On bisimilarity

- Roughly, two functions are bisimilar if they take equal arguments to bisimilar results.
  $m \approx_{\sigma \supset \sigma'} n$ iff whenever $m \Downarrow \lambda x.\, p$ for any p, there exists a q such that $n \Downarrow \lambda y.\, q$ and for every $r{:}\sigma$, $p[r/x] \preccurlyeq_{\sigma'} q[r/y]$ and vice versa.
- In presence of non-termination, bisimilarity is defined **coinductively** to break the circularity. Hence we can use a coinduction to establish two programs equivalent.
- The definition is **not** by quantification on contexts, but type-driven.
- Great, but we still need to:
  1. Show that it is a congruence
  2. Show that it coincides with ctx equivalence.

1. We do this using **Howe's** technique: introduce yet another relation (*the candidate relation*), which is easy to see is a congruence and show it coincides with bisimilarity.

Bisimilarity **is** context equivalence

- The notion of **context** is "unpleasantly concrete" for formalizations (Pitts):

$$\text{Ctx} \quad \mathcal{C} \quad ::= \quad \circ \mid \lambda\, x.\mathcal{C} \mid \mathcal{C}_1\ m \mid m\ \mathcal{C}_2 \mid \ldots$$

- Filling a hole in a context does not respect α-equivalence.
  Let $\mathcal{C} = \lambda x. \circ$. Then $\mathcal{C}\{x\} = \lambda x.\, x \neq \mathcal{C}\{y\} = \lambda x.\, y$
- An alternative "context-less" definition of contextual equivalence: the **largest compatible and adequate** (that is respecting convergence) relation on well-typed terms (Gordon, 1995).
  - This definition make essential use of Coq's impredicativity
- The proof that "context-less" contextual equivalence is a bisimulation is where Coq's native support for coinduction breaks down and where we use PACO.

Formal Results

We have formally defined and verified:

- A **general theory of dependent relations**, to be instantiated with open and closed terms.
- That **ground and applicative bisimilarity are congruences**.
- The **coincidence** of the concrete and the context-less representation of contextual equivalence.
- The **coincidence** of applicative/ground bisimilarity with applicative/ground contextual equivalence.
  - Around 110 theorems and 70 definitions for circa 2300 lines of code including sparse comments

Conclusions and Future work

In conclusion:

- Though both authors being Coq novices, we managed to push the state of the art a little further closing the circle between various forms and encodings of bisimilarity and contextual equivalence.
- The general setup of dependent relations is promising.

Conclusions and Future work

In conclusion:

- Though both authors being Coq novices, we managed to push the state of the art a little further closing the circle between various forms and encodings of bisimilarity and contextual equivalence.
- The general setup of dependent relations is promising.

What's next:

- Extending the results to a language with arbitrary recursive types:
  - idea: same object types, but view them **coinductively** (infinite trees)
- Tackle more challenging equivalences such as *probabilistic* context equivalence
  - For this we need some expertise with constructive real numbers

That's all, folks

Thanks!