A Bisimulation for Type Abstraction and Recursion

Eijiro Sumii
Benjamin C. Pierce
University of Pennsylvania

Main Result

The first sound, complete, and "elementary" proof method
for contextual equivalence in λ-calculus with full recursive, existential, and universal types

- Based on bisimulations
- No restriction to inductive or predicative types
- No domain theory or category theory required
- No admissibility or TT-closure required

Overview of the Talk

- Background
- Previous methods and their problems
  - Logical relations
  - Applicative bisimulations
- Our method, step by step
- Related work and future work

Background

- *Abstraction* or *information hiding* is crucial for developing complex systems
  - Including computer programs!
- *Type abstraction* is the primary method of information hiding in programming languages
  - Born in early 70's [Liskov 73, Morris 73, etc.]
  - Evolved to more sophisticated mechanisms such as modules, objects, components, etc.

A Classical Example

```
(*  in ML-like pseudo-code...  *)
  interface Complex =
    type t
    fun make : real ´ real ® t
    fun mul : t ´ t ® t
    fun re : t ® real
  end
```

An Implementation

```
(*  by Cartesian coordinates  *)
  module Cartesian : Complex =
    type t = real´real
    fun make(x,y) = (x,y)
    fun mul((x1,y1),(x2,y2)) =
      (x1 ´ x2 - y1 ´ y2, x1 ´ y2 + y1 ´ x2)
    fun re(x,y) = x
  end
```

Another Implementation

```
(*  by Polar coordinates  *)
  module Polar : Complex =
    type t = real ´ real
    fun make(x,y) =
      (sqrt(x ´ x + y ´ y), atan2(y,x))
    fun mul((r1,q1),(r2,q2)) =
      (r1 ´ r2, q1 + q2)
    fun re(r,q) = r ´ cos(q)
  end
```

Abstraction as Equivalence

- The two implementations **`Cartesian`** and **`Polar`** are *contextually equivalent* under the interface **`Complex`**

$$\vdash \texttt{Cartesian} \simeq \texttt{Polar} : \texttt{Complex}$$

I.e., they give the same result under any well-typed context in the language

- In this talk, "result" means only the final output value (or divergence)
  - Ignoring timing, energy, rounding errors, etc.

Question: How to Prove it?

Direct proof is difficult
because of infinite number of
"well-typed contexts"

$\Downarrow$

Proof methods have been studied:

- Logical relations
- Bisimulations

Overview of the Talk

- Background
- Previous methods and their problems
  - Logical relations
  - Applicative bisimulations
- Our method, step by step
- Related work and future work

Logical Relations for Type Abstraction [Reynolds 83, Mitchell 91]

Relations between programs,
defined by induction on their types

- Constants are related iff they are equal
- Tuples are related iff the elements are related
- Functions are related
  iff they map related arguments to related results
- Values of abstract type a can be assigned an arbitrary relation $j(a)$ <u>as long as all the other conditions are satisfied</u>

Logical Relations for Type Abstraction: Example

Let

$$\varphi(\textbf{Complex.t}) =$$
$$\{\ ((\mathbf{x},\mathbf{y}),(\mathbf{r},\theta))\ \mid$$
$$\mathbf{x} = \mathbf{r} \times \cos(\theta),\ \ \mathbf{y} = \mathbf{r} \times \sin(\theta)\ \}$$

Then

$$\varphi \vdash \textbf{Cartesian} \sim \textbf{Polar} : \textbf{Complex}$$

Contextual equivalence follows from soundness of logical relations

Problems with Logical Relations

- Become complex with recursion
  - Recursive functions complicate the soundness proof [Reynolds, Pitts]
  - Recursive types complicate the definition of logical relations [Birkedal-Harper-Crary]
    - Problematic since these also constrain contexts!

ß

Requires non-trivial argument about continuity (called *admissibility*) for each use, not just in the meta theory

Intuition: The gap between initiality and terminality

Overview of the Talk

- Background
- Previous methods and their problems
  - Logical relations
  - Applicative bisimulations
- Our method, step by step
- Related work and future work

Another Approach: Applicative Bisimulations

- Adopted from bisimulations in process calculi to untyped λ-calculus [Abramsky 90]
- Also adopted for (polymorphic) object calculi [Gordon-Rees]

Applicative Bisimulations: Definition

(for cbv λ-calculus <u>without</u> type abstraction)

A bisimulation is a relation between values s.t.

1. Bisimilar constants are equal
2. Bisimilar tuples have bisimilar elements
3. Bisimilar functions return bisimilar results <u>when applied to the same argument</u>

Problems with Applicative Bisimulations

- Soundness proof is difficult [Howe 96]
- Cannot prove any interesting equivalence of abstract data types
  - **`Cartesian.re`** and **`Polar.re`** do <u>not</u> return the same real number when applied to the <u>same</u> argument

This Work

- Sound and <u>complete</u> bisimulations for λ-calculus with <u>full recursive</u>, <u>existential</u>, and universal types
- Soundness proof simpler than Howe's method
  - Price: stronger condition for functions (necessary for existential types)

Overview of the Talk

- Background
- Previous methods and their problems
  - Logical relations
  - Applicative bisimulations
- Our method, step by step
- Related work and future work

First Try

- Bisimilar functions return bisimilar results when applied to <u>bisimilar</u> arguments

We are not done yet:

This is not sound because contexts can "compose" bisimilar values to make up more complex arguments

Second Try

- Bisimilar functions return bisimilar results when applied to $C[\mathbf{v_1},...,\mathbf{v_n}]$ and $C[\mathbf{v_1}',...,\mathbf{v_n}']$
  - for any bisimilar $\mathbf{v_1},...,\mathbf{v_n}$ and $\mathbf{v_1}',...,\mathbf{v_n}'$, and
  - for any <u>value</u> context C of appropriate type

Example: "Bisimulation" between `Cartesian` and `Polar`

R = { (**Cartesian**, **Polar**, **Complex**),

(**Cartesian.make**, **Polar.make**,
real ´ real ® Complex.t),

(**Cartesian.mul**, **Polar.mul**,
Complex.t ´ Complex.t ® Complex.t),

(**Cartesian.re**, **Polar.re**,
Complex.t ® real) }

È { (**(x,y)**,**(r,q)**, **Complex.t**) |
x = **r** ´ cos(q), **y** = **r** ´ sin(q) }

È { (**z**, **z**, **real**) | **z** : **real**}

Last Problem

Union of bisimulations is
no longer a bisimulation!
$\Downarrow$
Standard co-induction does not work

Counter-example: The union of

- The previous bisimulation R between `Cartesian` and `Polar`, and
- Its inverse R^{-1} (i.e., the bisimulation between `Polar` and `Cartesian`)
  - Wouldn't be even type-safe in general!

Solution

Consider <u>sets of relations</u>
as bisimulations

- Intuition: Each relation in a bisimulation represents a "world"

E.g., for the previous R between **`Cartesian`** and **`Polar`**,

- { R } is a bisimulation
- { R^{-1} } is another bisimulation
- { R, R^{-1} } is yet another bisimulation
- { R È R^{-1} } is <u>not</u> a bisimulation

Formal Definition (1/2)

- A *concretion environment* D is a partial map from abstract types a to pairs (s, s') of concrete types
  - Represents the implementations of abstract types in the lhs and rhs of equivalence
- A *typed value relation* R is a set of triples (v, v', t)

Formal Definition (2/2)

- A *bisimulation* X is a set of pairs ($\mathcal{D}$, R) with conditions for each type of values

  E.g., for every ($\mathcal{D}$, R)$\in$X, if

  $(\text{pack } s, v \text{ as } \exists a.t,\ \text{pack } s', v' \text{ as } \exists a.t,\ \exists a.t) \in R$

  then we have

  $$(\mathcal{D} \cup \{(a, s, s')\},\ R \cup \{(v, v', t)\}) \in X$$

  - Accounts for the *generativity* of existential types (i.e., opening the same package twice yields incompatible contents)

Example

X = { (Æ,R_0), (Δ,R_1), (Δ,R_2), (Δ,R_3) }

where

R_0 = { (pack int,(3,even) as $a.a´(a®bool),
pack bool,(true,not) as $a.a´(a®bool),
$a.a´(a®bool)) }

D = { (a, int, bool) }

R_1 = R_0 È { ((3,even), (true,not), a´(a®bool)) }

R_2 = R_1 È { (3, true, a) } È { (even, not, a®bool) }

R_3 = R_2 È { (false, false, bool) }

Intuition: Knowledge of the context increased by observations

Soundness and Completeness

- Generalize contextual equivalence to a "set of relations" as well
- Then, it coincides with the largest bisimulation (*bisimilarity*)
  - Completeness: by straightforward co-induction
  - Soundness: from the fact that evaluation preserves "bisimilar values in a context"
    - Much simpler than Howe's method, thanks to the stronger condition on functions (which is necessary for existential types)

Summary

- Sound and complete bisimulation for λ-calculus with universal, existential, and recursive types

- Other examples in the paper include:
  - Object encoding
    (using non-inductive recursive types)
  - Generative functors

Overview of the Talk

- Background
- Previous methods and their problems
  - Logical relations
  - Applicative bisimulations
- Our method, step by step
- Related work and future work

Related Work (1/2)

- Traditional logical relations and applicative bisimulations
- Logical relations for simply typed λ-calculus with dynamic sealing (a.k.a. perfect encryption) [Sumii-Pierce 01]
- Bisimulations for untyped λ-calculus with dynamic sealing [Sumii-Pierce 03]
  - Present work concerns static type abstraction instead of dynamic sealing, requiring careful treatment of type variables

Related Work (2/2)

Bisimulations for p-calculi with information hiding [Pierce-Sangiorgi-97, Abadi-Gordon-98, Abadi-Fournet-01, etc.]

- Similar spirit, different results because of the difference between p and l
  - Our formalism is more "uniform" and "monolithic" because functions are terms in l (while processes are not messages in p)
    - Cf. higher-order p-calculus and context bisimulation [Sangiorgi-92]
  - Completeness is trickier in p since the language is more imperative and low-level
    - Either (i) incompleteness known, (ii) "proof" found wrong, or (iii) no proof published

Future Work

- Applications to other forms of information hiding
  - E.g. secrecy typing [Abadi-97, Heintze-Riecke-98]
- *Fully abstract* encoding between various forms of information hiding
  - E.g. from polymorphic λ-calculus to untyped λ-calculus with perfect encryption
    [Pierce-Sumii 00, Sumii-Pierce 03]
- Programming language mechanisms based on these connections?