A Complete Characterization of Observational Equivalence in Polymorphic lambda-Calculus with General References

Eijiro Sumii

(Tohoku University)

Executive Summary

Sound and complete "proof method"
for contextual equivalence
in a language with

- Higher-order functions,
- First-class references (like ML), and
- Abstract data types

Caveat: the method is not fully automatic!

- The equivalence is (of course) undecidable in general
- Still, it successfully proved all known examples

(Very) General Motivation

1. Equations are important
   - $1 + 2 = 3$, $x + y = y + x$, $E = mc^2$, ...
2. Computing is (should be) a science
3. Therefore, equations are important in (so-called) computer science

4. Computing is described by programs
5. Therefore, equivalence of programs is important!

Program Equivalence as Contextual Equivalence

- In general, equations should be preserved under any <u>context</u>
  - E.g., x + y = y + x implies (x + y) + z = (y + x) + z by considering the context [ ] + z
- ⇒ <u>Contextual equivalence</u> (a.k.a. <u>observational equivalence</u>): Two programs "give the same result" under any context
  - Termination/divergence suffices for the "result"

Contextual Equivalence: Definition

Two programs P and Q are <u>contextually equivalent</u> if, for any context C,

C[P] terminates ⇔ C[Q] terminates

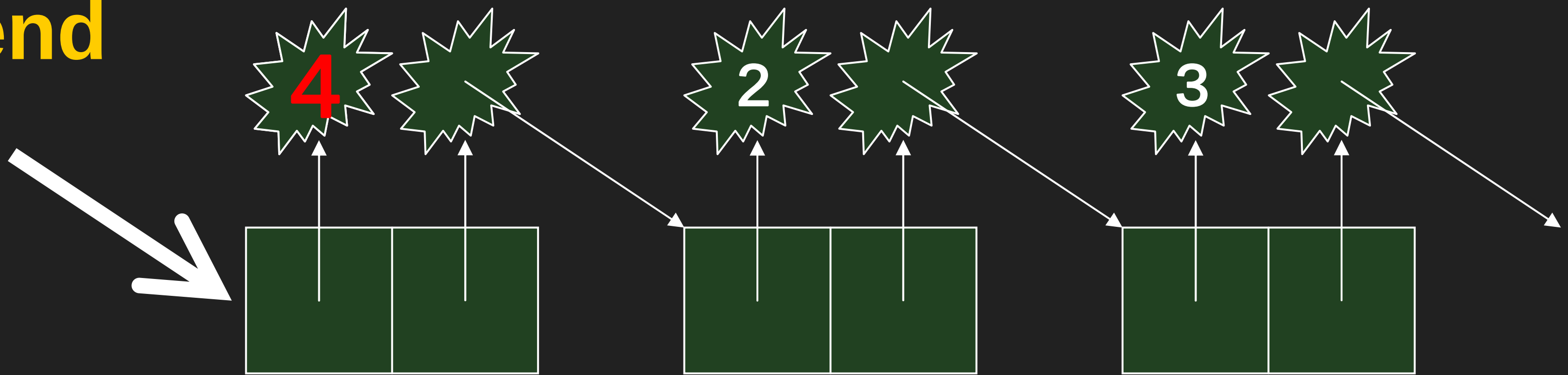
– C[P] (resp. C[Q]) means "filling in" the "hole" [ ] of C with P (resp. Q)

Example: Two Implementations of Mutable Integer Lists

```
(* pseudo-code in
   imaginary ML-like language *)
signature S
  type t (* abstract *)
  val nil : t
  val cons : int → t → t
  val setcar : t → int → unit
  (* car, cdr, setcdr, etc. omitted *)
end
```

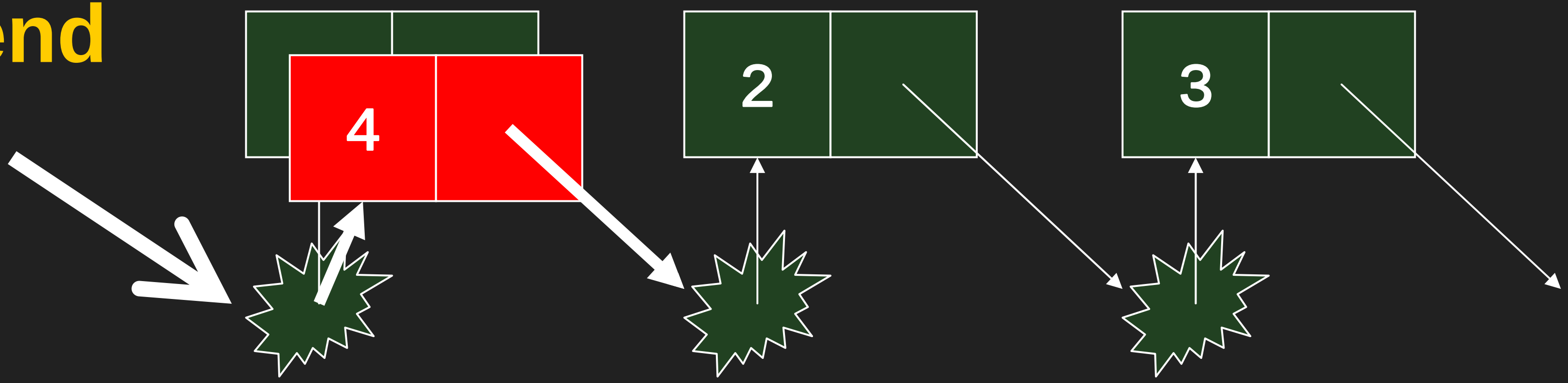
First Implementation

```
structure L
  type t = Nil | Cons of (int ref * t ref)
  let nil = Nil
  let cons a d = Cons(ref a, ref d)
  let setcar (Cons p) a =
    fst(p) := a
end
```



Second Implementation

```
structure L'
  type t = Nil | Cons of (int * t) ref
  let nil = Nil
  let cons a d = Cons(ref(a, d))
  let setcar (Cons r) a =
    r := (a, snd(!r))
end
```



The Problem

The implementations L and L' should be contextually equivalent under the interface S

How can we prove it?

- Direct proof is infeasible because of the universal quantification: "for any context C"
- Little previous work deals with both abstract data types and references (cf. [Ahmed-Dreyer-Rossberg POPL'09])
  - None is complete (to my knowledge)

Our Approach: Environmental Bisimulations

- Initially devised for λ-calculus with perfect encryption [Sumii-Pierce POPL'04]
- Successfully adapted for
  - Polymorphic λ-calculus [Sumii-Pierce POPL'05]
  - Untyped λ-calculus with references [Koutavas-Wand POPL'06] and deallocation [Sumii ESOP'09]
  - Higher-order π-calculus [Sangiorgi-Kobayashi-Sumii LICS'07]
  - Applied HOπ [Sato-Sumii APLAS'09, to appear] etc.

Our Target Language

Polymorphic λ-calculus with existential types and first-class references

M ::= ...standard λ-terms... |
pack (τ, M) as ∃α.σ |
open M as (α, x) in N |
ref M | !M | M := N | ℓ | M == N

locations

equality of locations

τ ::= ...standard polymorphic types... |
∃α.τ | τ ref

Environmental Relations

An environmental relation **X** is a set of tuples of the form:

$$(\Delta, R, s \triangleright M, s' \triangleright M', \tau)$$

Environmental Relations

An environmental relation X is a set of tuples of the form:

$$(\Delta, R, s \triangleright M, s' \triangleright M', \tau)$$

- Program M (resp. M') of type τ is running under store s (resp. s')
  - M and M' (and τ) are omitted when terminated

Environmental Relations

An <u>environmental relation</u> X is a set of tuples of the form:

$$(\Delta, R, s \triangleright M, s' \triangleright M', \tau)$$

- Program M (resp. M') of type τ is running under store s (resp. s')
  - M and M' (and τ) are omitted when terminated
- R is the <u>environment</u>: a (typed) relation between values known to the context

Environmental Relations

An <u>environmental relation</u> X is a set of tuples of the form:

$$(\Delta, R, s \triangleright M, s' \triangleright M', \tau)$$

- Program M (resp. M') of type τ is running under store s (resp. s')
  - M and M' (and τ) are omitted when terminated
- R is the <u>environment</u>: a (typed) relation between values known to the context
- Δ maps an abstract type α to (the pair of) their concrete types σ and σ'

Environmental Bisimulations for Our Calculus

An environmental relation X is an <u>environmental bisimulation</u> if it is preserved by

- execution of the programs and
- operations from the context

Formalized by the following conditions...

Environmental Bisimulations: Condition for Reduction

- If $(\Delta, R, s \triangleright M, s' \triangleright M', \tau) \in X$ and $s \triangleright M$ converges to $t \triangleright V$, then $s' \triangleright M'$ also converges to some $t' \triangleright V'$ with $(\Delta, R \cup \{(V, V', \tau)\}, t, t') \in X$

(Symmetric condition omitted)

Strictly speaking, this is a "big-step" version of environmental bisimulations

Environmental Bisimulations: Condition for Opening

- **If $(\Delta, R, s, s') \in X$ and
  $(\text{pack } (\tau, V) \text{ as } \exists\alpha.\sigma,$
  $\text{pack } (\tau', V') \text{ as } \exists\alpha.\sigma, \exists\alpha.\sigma) \in R$, then
  $(\Delta\cup\{(\alpha,\tau,\tau')\}, R\cup\{(V,V',\sigma)\}, s, s') \in X$**

Environmental Bisimulations: Condition for Dereference

- If $(\Delta, R, s, s') \in X$ and $(\ell, \ell', \sigma\ \mathrm{ref}) \in R$, then $(\Delta, R \cup \{(s(\ell), s'(\ell'), \sigma)\}, s, s') \in X$

Environmental Bisimulations: Condition for Update

- If $(\Delta, R, s, s') \in X$ and $(\ell, \ell', \sigma\ \text{ref}) \in R$, then $(\Delta, R, s\{\ell \mapsto W\}, s'\{\ell' \mapsto W'\}) \in X$ for any W and W' "synthesized" from R
  - Formally,

$$W = C[V_1,\dots,V_n]$$

$$W' = C[V'_1,\dots,V'_n]$$

    for some $(V_1,V'_1,\tau_1),\dots,(V_n,V'_n,\tau_n) \in R$ and some well-typed C

Environmental Bisimulations: Condition for Application

- If $(\Delta, R, s, s') \in X$ and $(\lambda x.M, \lambda x.M', \sigma \to \tau) \in R$, then $(\Delta, R, s \triangleright [W/x]M, s' \triangleright [W'/x]M', \tau) \in X$ for any W and W' synthesized from R

Other Conditions

- Similar conditions for allocation, location equality, projection, etc.
- <u>No</u> condition for values of abstract types

$$\text{If } (\Delta, R, s, s') \in X$$
$$\text{and } (V, V', \alpha) \in R,$$
$$\text{then } \ldots?$$

  – Context cannot operate on them

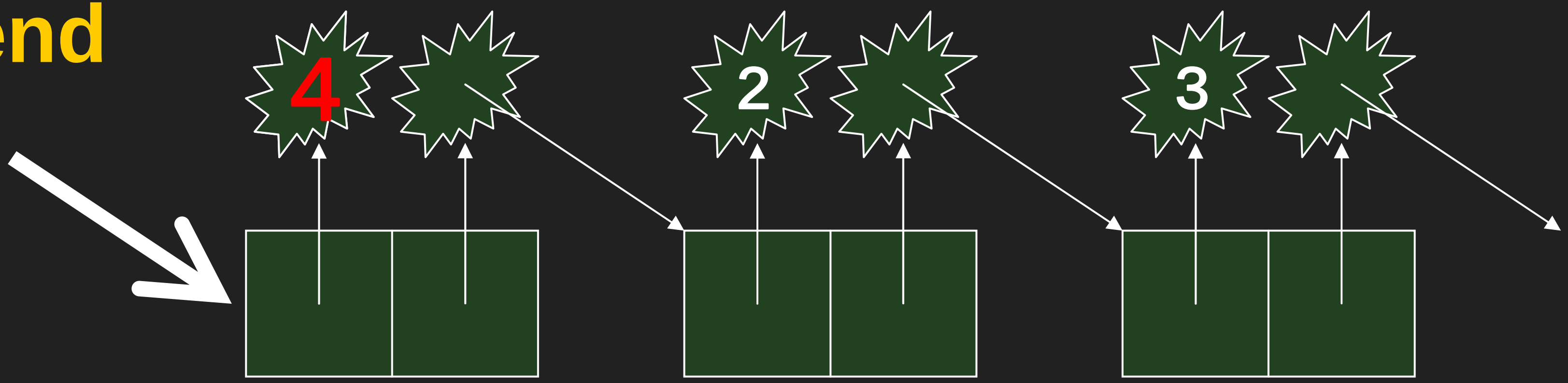
Abstract

Mutable Integer Lists Interface (Reminder)

```
(* pseudo-code in
   imaginary ML-like language *)
signature S
  type t (* abstract *)
  val nil : t
  val cons : int -> t -> t
  val setcar : t -> int -> unit
  (* setcdr, car, cdr, etc. omitted *)
end
```

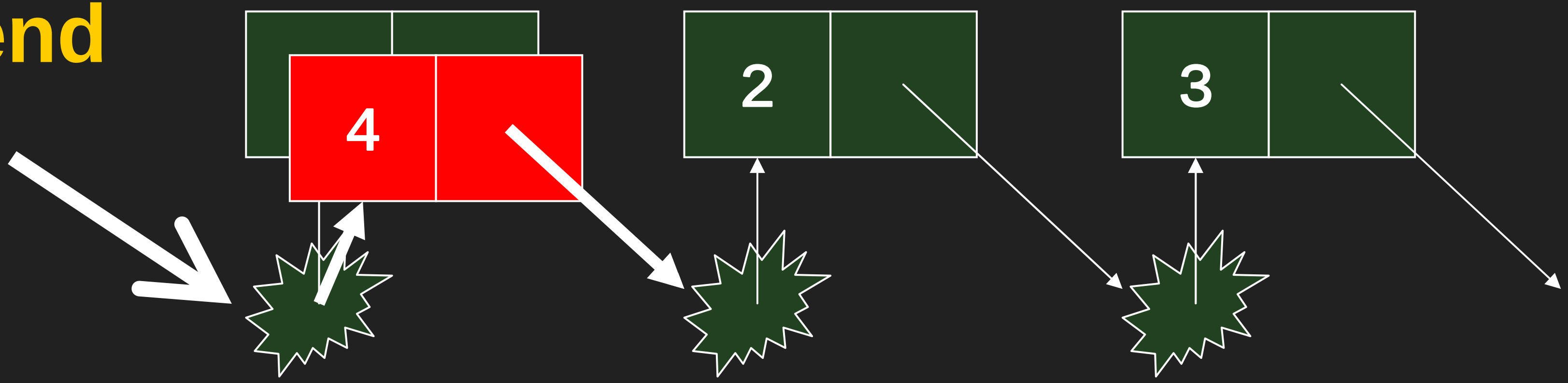
First Implementation (Reminder)

```
structure L
  type t = Nil | Cons of (int ref * t ref)
  let nil = Nil
  let cons a d = Cons(ref a, ref d)
  let setcar (Cons p) a =
    fst(p) := a
end
```



Second Implementation (Reminder)

```
structure L'
  type t = Nil | Cons of (int * t) ref
  let nil = Nil
  let cons a d = Cons(ref(a, d))
  let setcar (Cons r) a =
    r := (a, snd(!r))
end
```



Environmental Bisimulaton for The Mutable Integer Lists

$X = \{ (\Delta, R, s, s') \mid$

$\Delta = \{ (S.t, L.t, L'.t) \}$,

$R = \{ (L, L', S)$,

$(L.nil, L'.nil, S.t)$,

$(L.cons, L'.cons, int \to S.t \to S.t)$,

$(L.setcar, L'.setcar, S.t \to int \to unit)$,

$(L.Cons(\ell_i, m_i), L'.Cons(\ell'_i), S.t)$

$(L.Nil, L'.Nil, S.t) \mid i = 1, 2, 3, ..., n \}$,

$s(\ell_i) = fst(s'(\ell'_i))$ and

$(s(m_i), snd(s'(\ell'_i)), S.t) \in R$, for each $i \}$

More complicated example (1/3)

(* Adapted from [Ahmed-Dreyer-Rossberg POPL'09], credited to Thamsborg *)

pack (int ref, (ref 1, $\lambda x.V_x$)) as σ

vs. pack (int ref, (ref 1, $\lambda x.V'$)) as σ

where

V_x = $\lambda f.$ (x:=0; f(); x:=1; f(); !x)

V' = $\lambda f.$ (f(); f(); 1)

$\sigma = \exists\alpha.\ \alpha \times (\alpha \to (1 \to 1) \to \text{int})$

- f is supplied by the context
- What are the reducts of V f and V' f?

More complicated example (2/3)

$X = X_0 \cup X_1$

$X_0 = \{ (\Delta, R, t\{\ell \mapsto 0\} \triangleright N, t' \triangleright N', \text{int}) \mid$
N and N' are made of contexts in T_0,
with holes filled with elements of R $\}$

$X_1 = \{ (\Delta, R, t\{\ell \mapsto 1\} \triangleright N, t' \triangleright N', \text{int}) \mid$
N and N' are made of contexts in T_1,
with holes filled with elements of R $\}$

More complicated example (3/3)

- (C; ℓ:=1; D; !ℓ) T_0 (C; D; 1)
- (D; !ℓ) T_1 (D; 1)
- If E[zW] T_0 E'[zW], then
  E[C; ℓ:=1; D; !ℓ] T_0 E'[C; D; 1]
  (for any evaluation contexts E and E')
- If E[zW] T_0 E'[zW], then E[D; !ℓ] T_1 E'[D; 1]
- If E[zW] T_1 E'[zW], then
  E[C; ℓ:=1; D; !ℓ] T_0 E'[C; D; 1]
- If E[zW] T_1 E'[zW], then E[D; !ℓ] T_1 E'[D; 1]

Main Theorem: Soundness and Completeness

The largest environmental bisimulation ~ coincides with (a generalized form of) contextual equivalence ≡

Proof

- Soundness: Prove ~ is preserved under any context (by induction on the context)
- Completeness: Prove ≡ is an environmental bisimulation (by checking its conditions)

The Caveat

Our "proof method" is not automatic

- **Contextual equivalence in our language is undecidable**
- **Therefore, so is environmental bisimilarity**

...but it proved all known examples!

Up-To Techniques

Variants of environmental bisimulations with weaker (yet sound) conditions

- Up-to reduction (and renaming)
- Up-to context (and environment)
- Up-to allocation

Details in the paper

Related Work

- Environmental bisimulations for other languages (already mentioned)
- Bisimulations for other languages
- Logical relations
- Game semantics

None has dealt with <u>both</u> abstract data types and references

- Except [Ahmed-Dreyer-Rossberg POPL'09]

Conclusion

Summary:

Sound and complete "proof method" for contextual equivalence in polymorphic λ-calculus with existential types and references

Current and future work:

- Parametricity properties ("free theorems")
- Semantic model