SyTeCi: An Automated Tool to Prove Contextual Equivalence of Higher-Order Programs with References

Guilhem Jaber

Plume, LIP, Labex MILYON

Séminaire LSC
8 mars 2018
1 Introducing Program Equivalence

2 Contextual Equivalence for RefML

3 An Overview of SyTeCi
Why Study Program Equivalence?

- Specify the behavior of a program using a simpler program
  ↦ Check that the optimized program has the same behaviour.

- Ensure that program transformations are sound
  ↦ Compiler optimizations.

- Analyse a commit modifying a fragment of a program
  ↦ Regression analysis
  ↦ Correctness of refactorisations of programs.
A Simple Example

let rec fact₁ n =
  if (n ≤ 1) then 1
  else n * (fact₁ (n - 1))

let fact₂ n =
  let acc = ref 1 in
  let rec aux m =
    if (m ≤ 1) then ()
    else (acc := m * !acc; aux (m - 1))
  in aux n; !acc
But Wait...
What kind of equivalence are we talking about?
But Wait...
What kind of equivalence are we talking about?

Simplest solution:
- Provide the same input to the two programs
- Check that the outputs of the two programs are the same.
But Wait...
What kind of equivalence are we talking about?

Simplest solution:
- Provide the same input to the two programs
- Check that the outputs of the two programs are the same.

Does this work?
Reasoning on Programs
Reasoning on Programs

- Compositional approach
  - “Open” programs

- Higher-order programs
  - Functional programming
Reasoning on Programs

- Compositional approach
  ≻ “Open” programs

- Higher-order programs
  ≻ Functional programming
Compositional approach
  ⇾ “Open” programs

Higher-order programs
  ⇾ Functional programming

Side effects
  ⇾ References: mutable memory cell
Reasoning on Programs

- Compositional approach
  - “Open” programs

- Higher-order programs
  - Functional programming

- Side effects
  - References: mutable memory cell

- Abstractions: Modules, Polymorphism, Objects, . . .
  - Observational Power of the Environment.
Reentrant calls

When a program is called again in the middle of its execution.

- Because of concurrent executions of the same program
  - → Not considered in this talk.
- Because of callbacks
  - → Higher order programs.

Example:

```c
void sort (void *array, size_t nb, size_t size,
          int (*compare) (void const *a, void const *b));
```

What if `sort` is called in `compare`?
These two classes are equivalent:

```java
public class Counter1 {
    private int count;
    public Counter() { this.count = 0; }
    public void inc() { this.count++; }
    public int get() { return this.count; }
}

public class Counter2 {
    private int count;
    public Counter() { this.count = 0; }
    public void inc() { this.count--; }
    public int get() { return -this.count; }
}
```

count cannot be accessed directly.
1 Introducing Program Equivalence

2 Contextual Equivalence for RefML

3 An Overview of SyTeCi
Also known as **Observational Equivalence**:

- Programs as black-boxes,
- Put in contexts $E$ (a.k.a. environments: program with a hole),
- No context is be able to discriminate them,

  $\rightsquigarrow$ Need to characterize the **observational** power of contexts;

  $\rightsquigarrow$ Need to reason on all the contexts $E$. 
Contextual Equivalence: An Ubiquitous Notion

- Pure $\lambda$-calculus
  - $\rightsquigarrow$ Separation results: Bohm theorem

- Polymorphism
  - $\rightsquigarrow$ Relational Parametricity

- Denotational Semantics
  - $\rightsquigarrow$ Full Abstraction problem

- Security Protocols
  - $\rightsquigarrow$ Via process algebras
  - $\rightsquigarrow$ Indistinguishably properties
A typed functional programs: let rec sum(n, g) = sum(n - 1, g) + g(n)
A typed functional programs: let rec sum(n, g) = sum(n - 1, g) + g(n)

with Integers and Booleans: if b then 0 else n + 1

with pairs: \langle u, v \rangle
A typed functional programs: \[\text{let rec } \text{sum}(n, g) = \text{sum}(n - 1, g) + g(n)\]

with Integers and Booleans: \[\text{if } b \text{ then } 0 \text{ else } n + 1\]

with pairs: \[\langle u, v \rangle\]

with full ground references: \[\text{ref } 2, \text{ref (ref } \text{true})\]

stored in heap via locations: \[(\text{ref } v, h) \rightarrow (\ell, h \cdot [\ell \mapsto v])\]

\[(\ell \text{ fresh in the heap } h)\]

mutable: \[x := x + 1\]
A typed functional programs: \( \text{let rec sum}(n, g) = \text{sum}(n - 1, g) + g(n) \)

with Integers and Booleans: \( \text{if } b \text{ then } 0 \text{ else } n + 1 \)

with pairs: \( \langle u, v \rangle \)

with full ground references: \( \text{ref } 2, \text{ref } (\text{ref } \text{true}) \)

stored in heap via locations: \( \text{(ref } v, h) \rightarrow (\ell, h \cdot [\ell \mapsto v]) \)

(\( \ell \) fresh in the heap \( h \))

mutable: \( x :=!x + 1 \)

No pointer arithmetic: \( \ell + 1 \) is ill-typed

But equality test: \( \ell_1 == \ell_2 \) is well-typed
Contextual Equivalence

Contextual equivalence of $M_1, M_2$:

$$\forall E. \forall h. (E[M_1] \Downarrow, h) \iff (E[M_2] \Downarrow, h)$$

Observation $(M \Downarrow, h)$ : Termination.
Contextual Equivalence

Contextual equivalence of $M_1, M_2$:

$$\forall E. \forall h. (E[M_1] \downarrow, h) \iff (E[M_2] \downarrow, h)$$

Observation $(M \downarrow, h)$: Termination.

- Robust w.r.t the choice of observation.
- Depend on the language contexts are written in.

Undecidable in general $\Rightarrow$ Even in a finitary setting (finite datatypes, no recursion): Murawski & Tzevelekos, ICALP'12.

$\Rightarrow$ Because of the universal quantification over any context $E$. 

13 / 28
Contextual Equivalence

Contextual equivalence of $M_1, M_2$:

$$\forall E. \forall h. (E[M_1] \downarrow, h) \iff (E[M_2] \downarrow, h)$$

Observation $(M \downarrow, h)$: Termination.

- Robust w.r.t the choice of observation.
- Depend on the language contexts are written in.
- Undecidable in general
  - Even in a finitary setting (finite datatypes, no recursion): Murawski & Tzevelekos, ICALP’12.
  - Because of the universal quantification over any context $E$. 

The two following programs are equivalent:

\[
\text{let } c_1 = \text{ref0 in}
\text{let inc () = } c_1 := !c_1 + 1 \text{ in}
\text{let get () = } !c_1 \text{ in}
\langle\text{inc, get}\rangle
\]

\[
\text{let } c_2 = \text{ref0 in}
\text{let inc () = } c_2 := !c_2 - 1 \text{ in}
\text{let get () = } -!c_2 \text{ in}
\langle\text{inc, get}\rangle
\]

Need a relational invariant between \(c_1\) and \(c_2\):

\[
\forall x_1, x_2. c_1 \mapsto_1 x_1 \land c_2 \mapsto_2 x_2 \Rightarrow x_1 = -x_2
\]
Invariant are not enough!

$$\begin{align*}
\text{let } x &= \text{ref0} \\
\text{in let } \text{isc}_1(f : \text{Unit} \rightarrow \text{Unit}) = \\
x &:= 1; f(); !x
\end{align*}$$

$$\begin{align*}
\text{let } \text{isc}_2(f : \text{Unit} \rightarrow \text{Unit}) = \\
f(); 1
\end{align*}$$
Invariant are not enough!

\[
\begin{align*}
\text{let } x &= \text{ref0} \\
\text{in let } \text{isc}_1(f : \text{Unit} \rightarrow \text{Unit}) &= \\
&\quad x := 1; f(); \!x
\end{align*}
\]

\[
\begin{align*}
\text{let } \text{isc}_2(f : \text{Unit} \rightarrow \text{Unit}) &= \\
&\quad f(); 1
\end{align*}
\]

Need transition system of Invariants!
Invariant are not enough!

```ocaml
let x = ref 0
in let isc1 (f : Unit -> Unit) =
    x := 1; f(); !x
```

```ocaml
let isc2 (f : Unit -> Unit) =
    f(); 1
```

Need transition system of Invariants!

The transition system reflects the control flow between the programs and their environments.

⇝ Synchronization point on the callback $f()$
A subtle notion of equivalence

The following programs are not contextually equivalent:

```ocaml
let c₁ = ref0
in let inc (f : Unit → Unit) = f(); c₁ := !c₁ + 1
in let get () = !c₁
in ⟨inc, get⟩
```

```ocaml
let c₂ = ref0
in let inc (f : Unit → Unit) = let n = !c₂ in f(); c₂ := n + 1
in let get () = !c₂
in ⟨inc, get⟩
```

Because of reentrant calls, the programs C[•] ≜ let ⟨inc, get⟩ = • in let d = get () in inc (fun () ⇒ inc (fun () ⇒ ())). If get () ≠ d + 2 then Ω else () can discriminate them.
A subtle notion of equivalence

The following programs are not contextually equivalent:

let $c_1 = \text{ref} 0$

in

let inc ($f : \text{Unit} \rightarrow \text{Unit}$) =

\[
f() ; \quad c_1 := !c_1 + 1
\]

in

let get () =!$c_1$

in \langle \text{inc}, \text{get} \rangle

------

let $c_2 = \text{ref} 0$

in

let inc ($f : \text{Unit} \rightarrow \text{Unit}$) =

\[
\text{let } n = !c_2 \text{ in } f(); \quad c_2 := n + 1
\]

in

let get () =!$c_2$

in \langle \text{inc}, \text{get} \rangle

Because of reentrant calls!

\[
C[\bullet] \triangleq \text{let } \langle \text{inc}, \text{get} \rangle = \bullet \text{ in let } d = \text{get()} \text{ in }
\]

\[
\text{inc}(\text{fun()} \Rightarrow \text{inc} (\text{fun()} \Rightarrow ()));
\]

\[
\text{if get()} \neq d + 2 \text{ then } \Omega \text{ else } ()
\]

can discriminate them.
With a lock, they are contextually equivalent:

```ocaml
let c1 = ref 0
in let lock1 = ref false
in let inc (f : Unit → Unit) =
    if (not !lock1) then {
        lock1 := true;
        f(); c1 := !c1 + 1
        lock1 := false;
    } else ()
in let get () = !c1
in ⟨inc, get⟩
```

```ocaml
let c2 = ref 0
in let lock2 = ref false
in let inc (f : Unit → Unit) =
    if (not !lock2) then {
        lock2 := true;
        let n = !c2
        in f(); c2 := n + 1
        lock2 := false;
    } else ()
in let get () = !c2
in ⟨inc, get⟩
```
So how to prove automatically such examples of contextual equivalence?
So how to prove automatically such examples of contextual equivalence?

- Many techniques for proofs “on paper”.
  - Kripke Logical Relations, Environmental/Open/Parametric Bisimulations
- Decidable fragment: finite data-types and low-order types
  - Algorithmic Game Semantics
- But no general automated tools.
So how to prove automatically such examples of contextual equivalence?

- Many techniques for proofs “on paper”.
  - Kripke Logical Relations, Environmental/Open/Parametric Bisimulations
- Decidable fragment: finite data-types and low-order types
  - Algorithmic Game Semantics
- But no general automated tools.

Presenting a method to prove contextual equivalence automatically: SyTeCi
1 Introducing Program Equivalence

2 Contextual Equivalence for RefML

3 An Overview of SyTeCi
SyTeCi: A general sound tool to check contextual equivalence of RefML programs.

Reduce contextual equivalence to non-reachability of “inconsistent states” in a transition system $\mathcal{A}$ of memory configurations.

- No higher-order values anymore;
- Use non-determinism to represent the possible behaviour of all contexts.
SyTeCi: A general sound tool to check contextual equivalence of RefML programs.

- Reduce contextual equivalence to non-reachability of “inconsistent states” in a transition system $\mathcal{A}$ of memory configurations.
  - $\leadsto$ No higher-order values anymore;
  - $\leadsto$ Use non-determinism to represent the possible behaviour of all contexts.

- Paths to inconsistent states correspond to possible contexts that discriminate the two programs.
SyTeCi: A general sound tool to check contextual equivalence of RefML programs.

- Reduce contextual equivalence to non-reachability of “inconsistent states” in a transition system $\mathcal{A}$ of memory configurations.
  - $\rightsquigarrow$ No higher-order values anymore;
  - $\rightsquigarrow$ Use non-determinism to represent the possible behaviour of all contexts.

- Paths to inconsistent states correspond to possible contexts that discriminate the two programs.

- Check if inconsistent states are reachable from the initial state in $\mathcal{A}$.
  - $\rightsquigarrow$ Via model-checking;
  - $\rightsquigarrow$ May introduce over-approximations;
  - $\rightsquigarrow$ But works on most examples of the literature.
Behind the stage: Operational Nominal Game Semantics

- The interaction between a program and its environment is represented by a *trace*.

- Four kinds of basic interaction:
  - Player answer: the program returns a value (boolean, integer, function);
  - Player question: the program calls a function provided by the environment (callbacks);
  - Opponent answer: the environment returns a value after a callback;
  - Opponent question: the environment calls a function provided by the program.

- Higher-order values (i.e. functions) represented by free variables (i.e. atoms) in traces.
Generate step-by-step the interactions between each program and any environment

⇝ Using **symbolic evaluation** of the programs;
⇝ Generate constraints on the evolution of the heap for each possible execution path;
⇝ Handle open terms to deal with functions provided by the environment;
⇝ Non-determinism to represent the possible behaviours of all the environments;
⇝ Block on recursive calls in order to terminate.
Try to synchronize the interactions between the two programs.

\[\text{Using ideas from Open Bisimulations and Kripke Logical Relations;}
\]
\[\text{If a synchronization fails, then generate an “inconsistent state”.}
\]

Try to synchronize recursive calls between the two programs:

\[\text{Only work for similar programs;}
\]
\[\text{Allow circular reasoning}
\]
\[\text{May need to rewrite the program with trusted equivalence.}
\]
Try to synchronize the interactions between the two programs.

⇝ Using ideas from Open Bisimulations and Kripke Logical Relations;
⇝ If a synchronization fails, then generate an “inconsistent state”.

Try to synchronize recursive calls between the two programs:

⇝ Only work for similar programs;
⇝ Allow circular reasoning
⇝ May need to rewrite the program with trusted equivalence.

Symbolic Kripke Open Relations
Symbolic Evaluation

Syncronization Circular Reasoning

Symbolic Evaluation

$\phi$

Fail

Automatic Generation of the Transition System

$A$

Model-Checking

Yes  No  Fail
Constructing the WTS

$\mathcal{A}$ describe the evolution of the memory configurations w.r.t. the common control flow between the two programs and their environment.

- If constructed by hand: may be clever and find the right invariants.

- $\mathcal{A}$ is infinite in general
  - $\Rightarrow$ Because of infinite datatypes (Int, Loc).

- Automatic construction of $\mathcal{A}$:
  - $\Rightarrow$ Finite description of it
  - $\Rightarrow$ $\mathcal{A}$ have a stack to deal with recursion and well-bracketed interactions.
Model-checking techniques (Work In Progress):

- Predicate abstraction to finitize the transition system.

- SMT-solver (z3) to deal with arithmetic constraints.

- Summarization techniques for pushdown systems.

- “Abstracting abstract machines” techniques (Might & Van Horn) for the unboundness of the heap.
Soundness and Completeness

Contextual Equivalence

Operational Nominal Game Semantics (FoSSaCS’15)

Concrete Kripke Open Relations (APLAS’15)

Symbolic Kripke Open Relations

Completeness: only for recursion-free programs.
In the Future

- More precise model-checking techniques
  - Counter-example guided abstraction refinement (CEGAR).

- Interactive mode to help the tool finding synchronization points.

- Bounded checking
  - on the number of unwinding of fixed points,
  - on the number of reentrant calls.

- More general language
  - Polymorphism, recursive types.