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
  - $\rightsquigarrow$ Check that the optimized program has the same behaviour.
- Ensure that program transformations are sound
  - $\rightsquigarrow$ Compiler optimizations.
- Analyse a commit modifying a fragment of a program
  - $\rightsquigarrow$ Regression analysis
  - $\rightsquigarrow$ 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 program are the same.

But Wait...
What kind of equivalence are we talking about ?

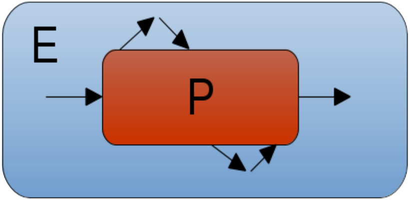
Simplest solution:

- Provide the same input to the two programs
- Check that the outputs of the two program are the same.

Does this work ?

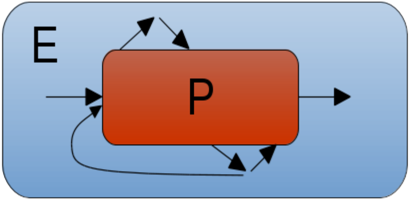
Reasoning on Programs

Reasoning on Programs



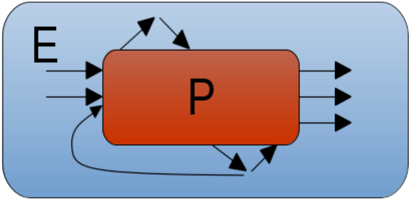
- Compositional approach
  - ⇝ "Open" programs
- Higher-order programs
  - ⇝ Functional programming

Reasoning on Programs



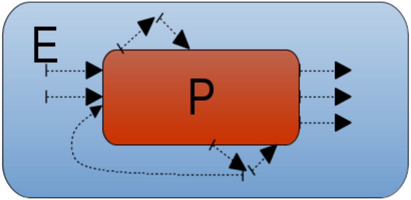
- Compositional approach
  - ⇝ "Open" programs
- Higher-order programs
  - ⇝ Functional programming

Reasoning on Programs



- Compositional approach
  - $\rightsquigarrow$ "Open" programs
- Higher-order programs
  - $\rightsquigarrow$ Functional programming
- Side effects
  - $\rightsquigarrow$ 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:

```
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` ?

Private memory cells

These two classes are equivalent:

```
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; }
}
```

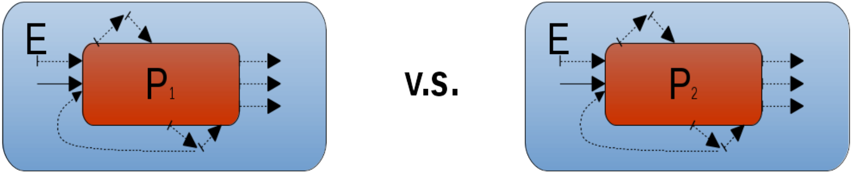
count cannot be accessed directly.

1 Introducing Program Equivalence

2 Contextual Equivalence for RefML

3 An Overview of SyTeCi

Contextual Equivalence



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,
  - ⇝ Need to characterize the **observational** power of contexts;
  - ⇝ Need to reason on all the contexts E.

Contextual Equivalence : An Ubiquitous Notion

- Pure λ-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

For what kind of Language: RefML

A typed functional programs : $\texttt{let rec sum}(\mathtt{n}, \mathtt{g}) = \mathtt{sum}(\mathtt{n} - 1, \mathtt{g}) + \mathtt{g}(\mathtt{n})$

For what kind of Language: RefML

A typed functional programs : $\texttt{let rec sum}(\mathtt{n}, \mathtt{g}) = \mathtt{sum}(\mathtt{n} - 1, \mathtt{g}) + \mathtt{g}(\mathtt{n})$

with Integers and Booleans: $\texttt{if b then 0 else n} + 1$

with pairs: $\langle u, v \rangle$

For what kind of Language: RefML

A typed functional programs : $\texttt{let rec sum}(\texttt{n}, \texttt{g}) = \texttt{sum}(\texttt{n} - 1, \texttt{g}) + \texttt{g}(\texttt{n})$

with Integers and Booleans: $\texttt{if b then 0 else n} + 1$
with pairs: $\langle u, v \rangle$

with full ground references: $\mathrm{ref}\, 2, \mathrm{ref}\,(\mathrm{ref}\, \textbf{true})$
stored in heap via locations: $(\mathrm{ref}\, v, h) \rightarrow (\ell, h \cdot [\ell \mapsto v])$
(ℓ fresh in the heap h)
mutable: $x := !x + 1$

For what kind of Language: RefML

A typed functional programs : $\texttt{let rec sum(n, g)} = \texttt{sum(n} - 1, \texttt{g)} + \texttt{g(n)}$

with Integers and Booleans: $\texttt{if b then 0 else n} + 1$
with pairs: $\langle u, v\rangle$

with full ground references: $\mathrm{ref}\,2, \mathrm{ref}\,(\mathrm{ref}\,\textbf{true})$
stored in heap via locations: $(\mathrm{ref}\,v, h) \to (\ell, h \cdot [\ell \mapsto v])$
(ℓ 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:

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

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

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

Contextual Equivalence

Contextual equivalence of M_1, M_2:

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

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

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

Representation Independence

The two following programs are equivalent:

$$
\begin{array}{l|l}
\texttt{let } \texttt{c}_1 \ = \texttt{ref0 in} & \texttt{let } \texttt{c}_2 \ = \texttt{ref0 in} \\
\texttt{let inc } () = \texttt{c}_1 := \boxed{!c_1 + 1} \texttt{ in} & \texttt{let inc } () = \texttt{c}_2 := \boxed{!c_2 - 1} \texttt{ in} \\
\texttt{let get } () = \boxed{!c_1} \texttt{ in} & \texttt{let get } () = \boxed{-!c_2} \texttt{ in} \\
\langle \texttt{inc}, \texttt{get} \rangle & \langle \texttt{inc}, \texttt{get} \rangle
\end{array}
$$

Need a relational invariant between c_1 and c_2:

$$\forall x_1, x_2. c_1 \mapsto_1 x_1 \wedge c_2 \mapsto_2 x_2 \Rightarrow x_1 = -x_2$$

Invariant are not enough !

$\texttt{let x} = \texttt{ref0}$ $\texttt{in let isc}_1(\texttt{f} : \mathrm{Unit} \to \mathrm{Unit}) =$ $\quad \texttt{x} := \texttt{1}; \texttt{f}(); !\texttt{x}$	$\texttt{let isc}_2(\texttt{f} : \mathrm{Unit} \to \mathrm{Unit}) =$ $\quad \texttt{f}(); \texttt{1}$

Invariant are not enough !

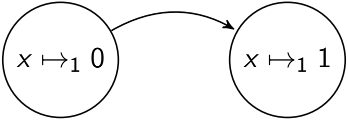
$\texttt{let x} = \texttt{ref0}$ $\texttt{in let isc}_1(\texttt{f} : \text{Unit} \to \text{Unit}) =$ $\quad \texttt{x} := 1; \texttt{f}(); !\texttt{x}$	$\texttt{let isc}_2(\texttt{f} : \text{Unit} \to \text{Unit}) =$ $\quad \texttt{f}(); 1$

Need transition system of Invariants !

Invariant are not enough !

$$\begin{array}{l|l} \texttt{let x} = \texttt{ref0} & \\ \texttt{in let isc}_1(\texttt{f} : \mathrm{Unit} \to \mathrm{Unit}) = & \texttt{let isc}_2(\texttt{f} : \mathrm{Unit} \to \mathrm{Unit}) = \\ \quad \texttt{x} := 1; \texttt{f}(); !\texttt{x} & \quad \texttt{f}(); 1 \end{array}$$

Need transition system of Invariants !



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

$\rightsquigarrow$ Synchronization point on the callback $\texttt{f}()$

A subtle notion of equivalence

The following programs are not contextually equivalent:

$\mathtt{let\ c_1}\ = \mathtt{ref0}$ $\mathtt{in\ let\ inc}\ (\mathtt{f} : \mathrm{Unit} \to \mathrm{Unit}) =$ $\quad \mathtt{f}();\ c_1 :=!c_1 + 1$ $\mathtt{in\ let\ get}\ ()\ =!\mathtt{c_1}$ $\mathtt{in}\ \langle \mathtt{inc}, \mathtt{get} \rangle$	$\mathtt{let\ c_2}\ = \mathtt{ref0}$ $\mathtt{in\ let\ inc}\ (\mathtt{f} : \mathrm{Unit} \to \mathrm{Unit}) =$ $\quad \mathtt{let\ n} =!\mathtt{c_2}\ \mathtt{in}$ $\quad \mathtt{f}();\ c_2 := n + 1$ $\mathtt{in\ let\ get}\ () =!\mathtt{c_2}$ $\mathtt{in}\ \langle \mathtt{inc}, \mathtt{get} \rangle$

A subtle notion of equivalence

The following programs are not contextually equivalent:

$$\begin{array}{l}
\texttt{let } \texttt{c}_1 \; = \texttt{ref}0 \\
\texttt{in let inc } (\texttt{f} : \mathrm{Unit} \to \mathrm{Unit}) = \\
\qquad \texttt{f}(); \; \boxed{c_1 := !c_1 + 1} \\
\texttt{in let } \mathsf{get} \; () \; = !\texttt{c}_1 \\
\texttt{in } \langle \texttt{inc}, \mathsf{get} \rangle
\end{array}
\quad\Bigg|\quad
\begin{array}{l}
\texttt{let } \texttt{c}_2 \; = \texttt{ref}0 \\
\texttt{in let inc } (\texttt{f} : \mathrm{Unit} \to \mathrm{Unit}) = \\
\qquad \texttt{let n} = !\texttt{c}_2 \texttt{ in} \\
\qquad \texttt{f}(); \; \boxed{c_2 := n + 1} \\
\texttt{in let } \mathsf{get} \; () = !\texttt{c}_2 \\
\texttt{in } \langle \texttt{inc}, \mathsf{get} \rangle
\end{array}$$

Because of reentrant calls !

$$\begin{array}{rcl}
C[\bullet] & \triangleq & \texttt{let } \langle \texttt{inc}, \mathsf{get} \rangle \; = \; \bullet \texttt{ in let d} \; = \; \mathsf{get}() \texttt{ in} \\
& & \quad \texttt{inc}(\texttt{fun}() \Rightarrow \texttt{inc } (\texttt{fun}() \Rightarrow ())); \\
& & \quad \texttt{if } \mathsf{get}() \; \neq \; \texttt{d} + 2 \texttt{ then } \Omega \texttt{ else } ()
\end{array}$$

can discriminate them.

Callback with lock

With a lock, they are contextually equivalent:

$$
\begin{array}{l}
\texttt{let c}_1 = \texttt{ref0} \\
\texttt{in let lock}_1 = \texttt{ref false} \\
\texttt{in let inc } (\texttt{f} : \mathrm{Unit} \to \mathrm{Unit}) = \\
\quad \texttt{if } (\texttt{not !lock}_1) \texttt{ then } \{ \\
\qquad \texttt{lock}_1 := \texttt{true}; \\
\qquad \texttt{f}(); \ \boxed{c_1 := !c_1 + 1} \\
\qquad \texttt{lock}_1 := \texttt{false}; \\
\quad \} \\
\quad \texttt{else } () \\
\texttt{in let get } () = !\texttt{c}_1 \\
\texttt{in } \langle \texttt{inc}, \texttt{get} \rangle
\end{array}
$$

$$
\begin{array}{l}
\texttt{let c}_2 = \texttt{ref0} \\
\texttt{in let lock}_2 = \texttt{ref false} \\
\texttt{in let inc } (\texttt{f} : \mathrm{Unit} \to \mathrm{Unit}) = \\
\quad \texttt{if } (\texttt{not !lock}_2) \texttt{ then } \{ \\
\qquad \texttt{lock}_2 := \texttt{true}; \\
\qquad \texttt{let n} = !\texttt{c}_2 \\
\qquad \texttt{in f}(); \ \boxed{c_2 := n + 1} \\
\qquad \texttt{lock}_2 := \texttt{false}; \\
\quad \} \\
\quad \texttt{else } () \\
\texttt{in let get } () = !\texttt{c}_2 \\
\texttt{in } \langle \texttt{inc}, \texttt{get} \rangle
\end{array}
$$

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”.
  - $\rightsquigarrow$ Kripke Logical Relations, Environmental/Open/Parametric Bisimulations
- Decidable fragment: finite data-types and low-order types
  - $\rightsquigarrow$ Algorithmic Game Semantics
- But no general automated tools.

So how to prove automatically such examples of contextual equivalence ?

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

Presenting a method to prove contextual equivalence automatically: SyTeCi

Contents

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.
  - $\rightsquigarrow$ No higher-order values anymore;
  - $\rightsquigarrow$ 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.
  - $\rightsquigarrow$ 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.

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.

SyTeCi: Basic Ideas (I/III)

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:
  - $\rightsquigarrow$ Player answer: the program returns a value (boolean, integer, function);
  - $\rightsquigarrow$ Player question: the program calls a function provided by the environment (callbacks);
  - $\rightsquigarrow$ Opponent answer: the environment returns a value after a callback;
  - $\rightsquigarrow$ 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.

SyTeCi: Basic Ideas (II/III)

Generate step-by-step the interactions between each program and any environment

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

SyTeCi: Basic Ideas (III/III)

Try to synchronize the interactions between the two programs.

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

Try to synchronize recursive calls between the two programs:

- $\rightsquigarrow$ Only work for similar programs;
- $\rightsquigarrow$ Allow circular reasoning
- $\rightsquigarrow$ May need to rewrite the program with trusted equivalence.

SyTeCi: Basic Ideas (III/III)

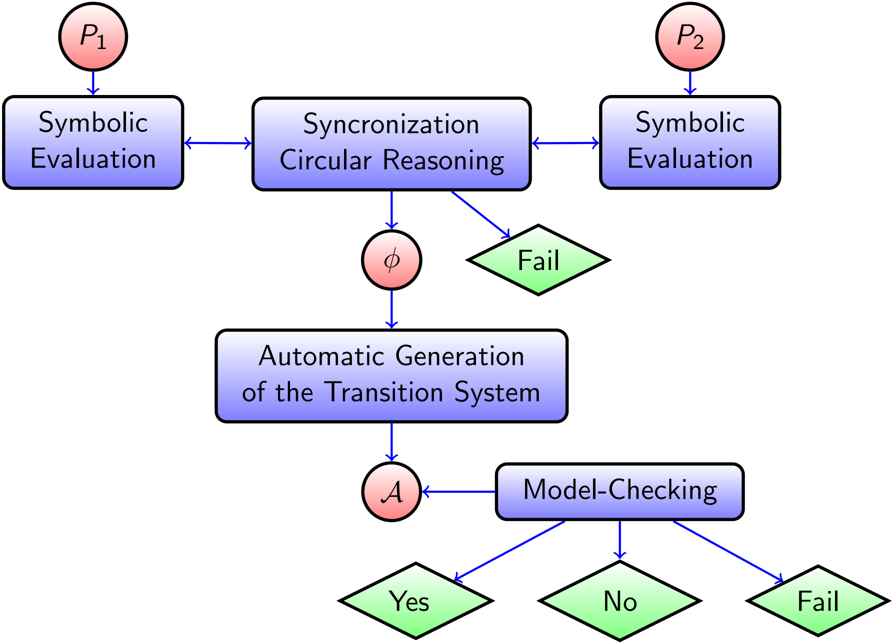
Try to synchronize the interactions between the two programs.

- $\rightsquigarrow$ Using ideas from Open Bisimulations and Kripke Logical Relations;
- $\rightsquigarrow$ If a synchronization fails, then generate an "inconsistent state".

Try to synchronize recursive calls between the two programs:

- $\rightsquigarrow$ Only work for similar programs;
- $\rightsquigarrow$ Allow circular reasoning
- $\rightsquigarrow$ May need to rewrite the program with trusted equivalence.

Symbolic Kripke Open Relations


P_1
P_2
Symbolic Evaluation
Syncronization Circular Reasoning
Symbolic Evaluation
ϕ
Fail
Automatic Generation of the Transition System
$\mathcal{A}$
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.

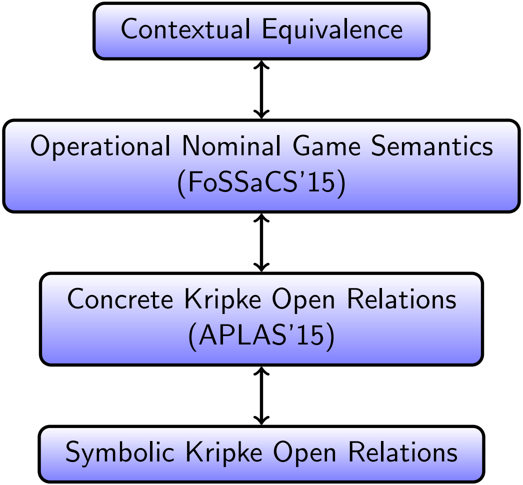
- If constructed by hand: may be clever and find the right invariants.
- $\mathcal{A}$ is infinite in general
  - $\rightsquigarrow$ Because of infinite datatypes $(\mathrm{Int}, \mathrm{Loc})$.
- Automatic construction of $\mathcal{A}$:
  - $\rightsquigarrow$ Finite description of it
  - $\rightsquigarrow$ $\mathcal{A}$ have a stack to deal with recursion and well-bracketed interactions.

Checking Non-reachability

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



Completeness: only for recursion-free programs.

In the Future

- More precise model-checking techniques
  - $\rightsquigarrow$ Counter-example guided abstraction refinement (CEGAR).
- Interactive mode to help the tool finding synchronization points.
- Bounded checking
  - $\rightsquigarrow$ on the number of unwinding of fixed points,
  - $\rightsquigarrow$ on the number of reentrant calls.
- More general language
  - $\rightsquigarrow$ Polymorphism, recursive types.