Can we automate proof of contextual equivalence using Logical Relations?

Guilhem Jaber

PPS, IRIF, Université Paris Diderot

Workshop on Program Equivalence
April 12th 2016
Goals

- Present techniques to prove *contextual equivalence* of programs:
  - Program as a black-box
  - the context can call it many times and whenever he wants,
Goals

- Present techniques to prove *contextual equivalence* of programs:
  - Program as a black-box
  - the context can call it many times and whenever he wants,
- for a functional language:
  - Open programs and callbacks,
  - Reentrant calls,
Goals

- Present techniques to prove *contextual equivalence* of programs:
  - ➞ Program as a black-box
  - ➞ the context can call it many time and whenever he wants,

- for a functional language:
  - ➞ Open programs and callbacks,
  - ➞ Reentrant calls,

- with references:
  - ➞ Local states,
  - ➞ Disclosure between the program and the context.
Goals

- Present techniques to prove *contextual equivalence* of programs:
  - Program as a black-box
  - the context can call it many times and whenever he wants,
- for a functional language:
  - Open programs and callbacks,
  - Reentrant calls,
- with references:
  - Local states,
  - Disclosure between the program and the context.

Kripke Logical Relations
Goals

- Present techniques to prove *contextual equivalence* of programs:
  - Program as a black-box
  - the context can call it many time and whenever he wants,
- for a functional language:
  - Open programs and callbacks,
  - Reentrant calls,
- with references:
  - Local states,
  - Disclosure between the program and the context.

**Kripke Logical Relations**

Can we automate them?
A typed functional programs:

```
fun test(n, g) = n + g(1)
```
A typed functional programs:

\[
\text{fun test}(n, g) = n + g(1)
\]

with Integers and Booleans:

\[
\text{if } b \text{ then } 0 \text{ else } n + 1
\]

with pairs:

\[
\langle u, \nu \rangle
\]
For what kind of Language: RefML

A typed functional programs:

fun test(n, g) = n + g(1)

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

with pairs:
⟨u, v⟩

with higher-order references:
ref 2, ref (λx. M)
(ref v, h) → (ℓ, h · [ℓ ↦ v])
(ℓ fresh in h)
x :=!x + 1
A typed functional programs:

fun test(n, g) = n + g(1)

with Integers and Booleans:

if b then 0 else n + 1

with pairs:

⟨u, v⟩

with higher-order references:

ref 2, ref (λx. M)

(ref v, h) → (ℓ, h · [ℓ ↦ v])

(ℓ fresh in h)

mutable:

x :=! x + 1

No pointer arithmetic:

(ℓ + 1) is ill-typed

But equality test:

ℓ₁ == ℓ₂ is well-typed
A typed functional program:

fun test(n, g) = n + g(1)

with Integers and Booleans:

if b then 0 else n + 1

with pairs:

⟨u, ν⟩

with higher-order references:

ref 2, ref (λx. M)

stored in heap via locations:

(ref ν, h) → (ℓ, h · [ℓ ↦ ν])

(ℓ fresh in h)

x :=!x + 1

mutable:

No pointer arithmetic:

(ℓ + 1) is ill-typed

But equality test:

ℓ₁ == ℓ₂ is well-typed

Full recursion (via Landin “knot”).
Contextual Equivalence

Contextual equivalence of $M_1, M_2$:

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

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

Contextual equivalence of $M_1, M_2$:

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

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

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

Contextual equivalence of $M_1, M_2$:

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

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
Contextual Equivalence

Contextual equivalence of $M_1, M_2$:

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

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

Many techniques to reason on contextual equivalence:

- Algorithmic Game Semantics,
- Environmental Bisimulations, Open Bisimulations,
- Kripke Logical Relations.
fun test₁(f₁) = f₁() is not equivalent to
fun test₂(f₂) = f₂(); f₂()
fun test₁(f₁) = f₁() is not equivalent to
fun test₂(f₂) = f₂(); f₂()

⇝ Contexts can check how many time f₁, f₂ are called.
⇝ Callbacks are fully observable!

\[ C[\bullet] \overset{def}{=} \text{let } x = \text{ref } 0 \text{ in } \bullet (\lambda . x := !x + 1); \text{if } !x > 1 \text{ then } \Omega \text{ else } () \]
can discriminate them.
fun \text{add}_1(f_1) = (f_1 \ 1) + (f_1 \ 2) \quad \text{is not equivalent to}
fun \text{add}_2(f_2) = (f_2 \ 2) + (f_2 \ 1)
fun \( \text{add}_1(f_1) = (f_1 1) + (f_1 2) \) is not equivalent to
fun \( \text{add}_2(f_2) = (f_2 2) + (f_2 1) \)

\[\implies\] Arguments given to callbacks must be related.

\[
C[\bullet] \overset{def}{=} \text{let } x = \text{ref } 0 \text{ in } \bullet (\lambda y. x := y); \text{if } !x == 1 \text{ then } \Omega \text{ else }()
\]
can discriminate them.
fun const₁() = let x = ref0 in 1  is equivalent to
  fun const₂() = 1

⇝ The creation of the reference bounded to \(x\) is not observable by the
context.

⇝ It is private to the term!
fun discl_1(f) = let x = ref0 in fx; x := 1

is not equivalent to

fun discl_2(f) = let x = ref0 in fx; x := 2
Disclosure of Locations (II/II)

fun discl_1(f) = let x = ref0 in fx; x := 1

is not equivalent to

fun discl_2(f) = let x = ref0 in fx; x := 2

⇝ The reference bound to x is disclosed to the context.
⇝ It can look inside afterwards to see what is stored.

C[•] \overset{def}{=} let z = ref(ref 0) in • (λy.z := y); if !!z == 1 then Ω else()
can discriminate them.
The two following programs are equivalent:

```ocaml
let c_1 = ref0
    fun inc_1() . c_1 := !c_1 + 1
    fun get_1() . !c_1
  in ⟨inc_1, get_1⟩

let c_2 = ref0
    fun inc_2() . c_2 := !c_2 - 1
    fun get_2() . -!c_2
  in ⟨inc_2, get_2⟩
```

Need a relational invariant between $c_1$ and $c_2$. 
Are the two following programs equivalent?

```
let c_1 = ref0
  fun inc_1(f) = f(); c_1 := !c_1 + 1
  fun get_1() = !c_1
in ⟨inc_1, get_1⟩

let c_2 = ref0
  fun inc_2(f) = let n = !c_2 in f(); c_2 := n + 1
  fun get_2() = !c_2
in ⟨inc_2, get_2⟩
```

No, because of reentrant calls!
Are the two following programs equivalent?

```
let c1 = ref0
  fun inc1(f) = f(); c1 := !c1 + 1
  fun get1() = !c1
in ⟨inc1, get1⟩
```

```
let c2 = ref0
  fun inc2(f) = let n = !c2 in f(); c2 := n + 1
  fun get2() = !c2
in ⟨inc2, get2⟩
```

No, because of reentrant calls!

\[ C[\bullet] \overset{\text{def}}{=} \text{let } ⟨\text{inc, get}⟩ = \bullet \text{ in let } d = \text{get()} \text{ in inc}(\lambda_. \text{inc} (\lambda x.x)); \text{ if get()} \neq d + 2 \text{ then } \Omega \text{ else } () \]

can discriminate them.
Logical Relations

Binary relations $\mathcal{E} \left[ \tau \right]$, $\mathcal{V} \left[ \tau \right]$ on closed terms and values

inductively defined on types.

$\mathcal{V} \left[ \text{Int} \right] \overset{\text{def}}{=} \{(n, n) \mid n \in \mathbb{Z}\}$

$\mathcal{V} \left[ \tau \rightarrow \sigma \right] \overset{\text{def}}{=} \{\left(\lambda x_1.M_1, \lambda x_2.M_2\right) \mid \forall (v_1, v_2) \in \mathcal{V} \left[ \tau \right].
\left(\left(\lambda x_1.M_1\right)v_1, \left(\lambda x_2.M_2\right)v_2\right) \in \mathcal{E} \left[ \sigma \right]\}$

$\mathcal{E} \left[ \tau \right] \overset{\text{def}}{=} \{\left(M_1, M_2\right) \mid (M_1 \uparrow \land M_2 \uparrow) \land\n\lor\left(\left(M_1 \mapsto^* v_1\right) \land \left(M_2 \mapsto^* v_2\right) \land (v_1, v_2) \in \mathcal{V} \left[ \tau \right]\}\}$
Operational reasoning for functions with local state

Andrew Pitts and Ian Stark

Abstract

Languages such as ML or Lisp permit the use of recursively defined function expressions with locally declared storage locations. Although this can be very convenient from a programming point of view it severely complicates the properties of program equivalence even for relatively simple fragments of such languages—such as the simply typed fragment of Standard ML with integer-valued references considered here. This paper presents a method for reasoning about contextual equivalence of programs involving this combination of functional and procedural features. The method is based upon the use of a certain kind of logical relation parameterised by relations between program states. The form of this logical relation is novel, in as much as it involves relations not only between program expressions, but also between program continuations (also known as evaluation contexts). The authors found this approach necessary in order to establish the ‘Fundamental Property of logical relations’ in the presence of both dynamically allocated local state and recursion. The logical relation characterises contextual equivalence and yields a proof of the best known context lemma for this kind of language—the Mason-Talcott ‘ciu’ theorem. Moreover, it is shown that the method can prove examples where such a context lemma is not much help and which involve representation independence, higher order memoising functions, and profiling functions.

1. Introduction

Lisp and ML are functional programming languages because they treat functions as values on a par with more concrete forms of data: functions can be passed as arguments, can be returned as the result of computation, can be recursively defined, and so on. They are also procedural languages because they permit the use of references (or ‘cells’, or ‘locations’) for storing values: references can be created dynamically and their contents read and updated as expressions are evaluated. This paper presents a method for reasoning about contextual equivalence of programs involving this combination of functional and procedural features. What emerges is an operationally-based form of reasoning about functions with local state that seems to be both intuitive and theoretically powerful. Throughout we assume a passing familiarity with the language Standard ML (Milner, Tofte, and Harper 1990) and its associated terminology. If in difficulty, see (Paulson 1991).
Extension to languages with references.

\[ \sim \Rightarrow \text{Need } \textit{worlds } w, \text{ i.e. invariants on heaps,} \]

\[ \sim \Rightarrow \text{Extensible on disjoint parts: } w' \sqsubseteq w \]

\[ \sim \Rightarrow \text{Parametrize the definition of logical relations with such worlds.} \]
Kripke Logical Relations

Extension to languages with references.

⇝ Need worlds $w$, i.e. invariants on heaps,

⇝ Extensible on disjoint parts: $w' \sqsubseteq w$

⇝ Parametrize the definition of logical relations with such worlds.

$$\mathcal{E}[\tau] w \overset{def}{=} \left\{ (M_1, M_2) \mid \forall (h_1, h_2) : w . (\langle M_1, h_1 \rangle \uparrow \land \langle M_2, h_2 \rangle \uparrow) \right.$$ \begin{align*} &\lor \left( \langle \langle M_1, h_1 \rangle \rightarrow^* (v_1, h_1') \rangle \land \langle \langle M_2, h_2 \rangle \rightarrow^* (v_2, h_2') \rangle \right) \\
&\lor \exists w' \sqsubseteq w . (h_1', h_2') \in w' \land (v_1, v_2) \in V[\tau] w' \right\}$$
Kripke Logical Relations

Extension to languages with references.

⇝ Need worlds $w$, i.e. invariants on heaps,

⇝ Extensible on disjoint parts: $w' \sqsupseteq w$

⇝ Parametrize the definition of logical relations with such worlds.

$$\mathcal{E} \{ \tau \} w \overset{\text{def}}{=} \left\{ (M_1, M_2) \mid \forall (h_1, h_2) : w.((M_1, h_1) \uparrow \land (M_2, h_2) \uparrow) \lor (((M_1, h_1) \mapsto^* (v_1, h'_1)) \land ((M_2, h_2) \mapsto^* (v_2, h'_2))) \exists w' \sqsupseteq w.((h'_1, h'_2) \in w' \land (v_1, v_2) \in \mathcal{V} [\tau] w') \right\}$$

$$\mathcal{V} [\tau \rightarrow \sigma] w \overset{\text{def}}{=} \left\{ (\lambda x_1. M_1, \lambda x_2. M_2) \mid \forall w' \sqsupseteq w. \forall (v_1, v_2) \in \mathcal{V} [\tau] w'.((\lambda x_1. M_1)v_1, (\lambda x_2. M_2)v_2) \in \mathcal{E} [\sigma] w' \right\}$$
Equivalence of the Representation Independence example (I/II)

let $c_1 = \text{ref0}$

fun $\text{inc}_1() = c_1 := !c_1 + 1$

fun $\text{get}_1() = !c_1$

in $\langle \text{inc}_1, \text{get}_1 \rangle$

let $c_2 = \text{ref0}$

fun $\text{inc}_2() = c_2 := !c_2 - 1$

fun $\text{get}_2() = -!c_2$

in $\langle \text{inc}_2, \text{get}_2 \rangle$
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)

- Prove \((\text{inc}_1, \text{inc}_2) \in V[\text{Unit} \rightarrow \text{Unit}] w\)
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)

- Prove \((\text{inc}_1, \text{inc}_2) \in V[\text{Unit} \to \text{Unit}] \) \( w \)
  \( \leadsto \) Take \((h_1, h_2) \in w\),
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)

- Prove \((\text{inc}_1, \text{inc}_2) \in \mathcal{V}[\text{Unit} \to \text{Unit}] \) \( w \)
  - \( \leadsto \) Take \((h_1, h_2) \in w\),
  - \( \leadsto \) \((\text{inc}_i(), h_i) \mapsto^* ((()), h_i')\).
Equivalence of the Representation Independence example (I/II)

- Take $w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\}$

- Prove $(\text{inc}_1, \text{inc}_2) \in \mathcal{V}\left[\text{Unit} \rightarrow \text{Unit}\right] w$
  - Take $(h_1, h_2) \in w,$
  - $(\text{inc}_i(), h_i) \mapsto^* ((), h'_i),$
  - $h'_1 = h_1[c \mapsto (h_1(c) + 1)]$ and $h'_2 = h_2[c \mapsto (h_2(c) - 1)],$
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)

- Prove \((\text{inc}_1, \text{inc}_2) \in V[\text{Unit} \rightarrow \text{Unit}] \) \( w \)
  
  \( \leadsto \) Take \((h_1, h_2) \in w, \)
  
  \( \leadsto (\text{inc}_i(), h_i) \mapsto^* ((), h'_i), \)
  
  \( \leadsto h'_1 = h_1[c \mapsto (h_1(c) + 1)] \) and \( h'_2 = h_2[c \mapsto (h_2(c) - 1)], \)
  
  \( \leadsto \) So that \((h'_1, h'_2) \in w! \)
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)

- Prove \((\text{inc}_1, \text{inc}_2) \in V[\text{Unit} \rightarrow \text{Unit}] w\)
  \(\leadsto\) Take \((h_1, h_2) \in w\),
  \(\leadsto\) \((\text{inc}_i(), h_i) \mapsto^* (((), h_i'),\)
  \(\leadsto\) \(h'_1 = h_1[c \mapsto (h_1(c) + 1)]\) and \(h'_2 = h_2[c \mapsto (h_2(c) - 1)]\),
  \(\leadsto\) So that \((h'_1, h'_2) \in w\)!

- Prove \((\text{get}_1, \text{get}_2) \in V[\text{Unit} \rightarrow \text{Int}] w\)
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)

- Prove \((\text{inc}_1, \text{inc}_2) \in V[\text{Unit} \to \text{Unit}]\) \( w \)
  \[\sim \quad \text{Take } (h_1, h_2) \in w, \]
  \[\sim \quad (\text{inc}_i(), h_i) \mapsto^* (((), h_i'), \]
  \[\sim \quad h'_1 = h_1[c \mapsto (h_1(c) + 1)] \text{ and } h'_2 = h_2[c \mapsto (h_2(c) - 1)], \]
  \[\sim \quad \text{So that } (h'_1, h'_2) \in w! \]

- Prove \((\text{get}_1, \text{get}_2) \in V[\text{Unit} \to \text{Int}]\) \( w \)
  \[\sim \quad \text{Take } (h_1, h_2) \in w, \]
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)

- Prove \( (\text{inc}_1, \text{inc}_2) \in \mathcal{V} \llbracket \text{Unit} \to \text{Unit} \rrbracket \ w \)
  \( \leadsto \) Take \( (h_1, h_2) \in w, \)
  \( \leadsto (\text{inc}_i(), h_i) \mapsto^* ((), h'_i), \)
  \( \leadsto h'_1 = h_1[c \mapsto (h_1(c) + 1)] \text{ and } h'_2 = h_2[c \mapsto (h_2(c) - 1)], \)
  \( \leadsto \) So that \( (h'_1, h'_2) \in w \)

- Prove \( (\text{get}_1, \text{get}_2) \in \mathcal{V} \llbracket \text{Unit} \to \text{Int} \rrbracket \ w \)
  \( \leadsto \) Take \( (h_1, h_2) \in w, \)
  \( \leadsto (\text{get}_1(), h_1) \mapsto^* (h_1(c_1), h_1) \text{ and } (\text{get}_2, h_2) \mapsto^* (-h_2(c_2), h_2), \)
Equivalence of the Representation Independence example (I/II)

- Take \( w = \{(h_1, h_2) \mid h_1(c_1) = -h_2(c_2)\} \)

- Prove \((\text{inc}_1, \text{inc}_2) \in V[\text{Unit} \to \text{Unit}] w\)
  
  \[\leadsto \text{Take } (h_1, h_2) \in w,\]
  
  \[\leadsto (\text{inc}_i(), h_i) \mapsto^* ((), h'_i),\]
  
  \[\leadsto h'_1 = h_1[c \mapsto (h_1(c) + 1)] \text{ and } h'_2 = h_2[c \mapsto (h_2(c) - 1)],\]
  
  \[\leadsto \text{So that } (h'_1, h'_2) \in w!\]

- Prove \((\text{get}_1, \text{get}_2) \in V[\text{Unit} \to \text{Int}] w\)
  
  \[\leadsto \text{Take } (h_1, h_2) \in w,\]
  
  \[\leadsto (\text{get}_1(), h_1) \mapsto^* (h_1(c_1), h_1) \text{ and } (\text{get}_2, h_2) \mapsto^* (-h_2(c_2), h_2),\]
  
  \[\leadsto \text{Need to prove } h_1(c_1) = -h_2(c_2): \text{ Straightforward from } (h_1, h_2) \in w!\]
let x = ref0 in fun awk₁(f) = x := 1; f(); !x ≃ fun awk₂(f) = f(); 1
let x = ref0 in fun awk₁(f) = x := 1; f(); !x ≻ fun awk₂(f) = f(); 1

Need transition system of Invariants!
Invariants are not Enough

\[
\text{let } x = \text{ref0 in fun awk}_1(f) = x := 1; f(); !x \simeq \text{fun awk}_2(f) = f(); 1
\]

Need transition system of Invariants!

\[
\begin{array}{c}
\text{Future world } w' \sqsubseteq w \text{ can either:}
\end{array}
\]

- Add a new transition system of invariants on a disjoint part of the heaps,
- Take a transition to get the following invariant.
Invariants are not Enough

```ml
let x = ref0 in fun awk1(f) = x := 1; f(); !x ≃ fun awk2(f) = f(); 1
```

Need transition system of Invariants!

Future world $w'$ $\sqsupseteq w$ can either:
- Add a new transition system of invariants on a disjoint part of the heaps,
- Take a transition to get the following invariant.
The Impact of Higher-Order State and Control Effects on Local Relational Reasoning

Derek Dreyer
MPI-SWS
dreyer@mpi-sws.org

Georg Neis
MPI-SWS
veis@mpi-sws.org

Lars Birkedal
IT University of Copenhagen
birkedal@itu.dk

Abstract
Reasoning about program equivalence is one of the oldest problems in semantics. In recent years, useful techniques have been developed, based on bisimulations and logical relations, for reasoning about equivalence in the setting of increasingly realistic languages—languages nearly as complex as ML or Haskell. Much of the recent work in this direction has considered the interesting representation independence principles enabled by the use of local state, but it is also important to understand the principles that powerful features like higher-order state and control effects disable. This latter topic has been broached extensively within the framework of game semantics, resulting in what Abramsky dubbed the “semantic cube”: fully abstract game-semantic characterizations of various axes in the design space of ML-like languages. But when it comes to reasoning about many actual examples, game semantics does not yet supply a useful technique for proving equivalences.

In this paper, we marry the aspirations of the semantic cube to the powerful proof method of step-indexed Kripke logical relations. Building on recent work of Ahmed, Dreyer, and Rossberg, we define the first fully abstract logical relation for an ML-like language with recursive types, abstract types, general references and call/cc. We then show how, under orthogonal restrictions to the expressive power of our language—namely, the restriction to first-order state and/or the removal of call/cc—we can enhance the proving power of our possible-worlds model in correspondingly orthogonal ways, and we demonstrate this proving power on a range of interesting examples. Central to our story is the use of state transition systems to model the way in which properties of local state evolve over time.

1. Introduction
Reasoning about program equivalence is one of the oldest problems in semantics, with applications to program verification (“Is an optimized program equivalent to some reference implementation?”), compiler correctness (“Does a program transformation preserve the semantics of the source program?”), representation independence (“Can we modify the internal representation of an abstract data type without affecting the behavior of clients?”), and more besides.

The canonical notion of program equivalence for many applications is observational (or contextual) equivalence. Two programs are observationally equivalent if no program context can distinguish them by getting them to exhibit observably different input/output behavior. Reasoning about observational equivalence directly is difficult, due to the universal quantification over program contexts. Consequently, there has been a huge amount of work on developing useful models and logics for observational equivalence, and in recent years this line of work has scaled to handle increasingly realistic languages—languages nearly as complex as ML or Haskell, with features like general recursive types, general (higher-order) mutable references, and first-class continuations.

The focus of much of this recent work—e.g., environmental bisimulations [36, 17, 32, 35], normal form bisimulations [34, 16], step-indexed Kripke logical relations [4, 2, 3]—has been on establishing some effective techniques for reasoning about programs that actually use the interesting, semantically complex features (state, continuations, etc.) of the languages being modeled. For instance, most of the work on languages with state concerns the various kinds of representation independence principles that arise due to the use of local state as an abstraction mechanism.
How to automate reasoning on Kripke Logical Relations

Obstacles to automate Kripke Logical Relations:

- Definition of $\mathcal{V}[\tau \to \sigma]$ quantify over pairs of values in $\mathcal{V}[\tau]$,
- Future worlds $w' \sqsupseteq w$ can add arbitrary new systems of invariants.

Solutions:
Obstacles to automate Kripke Logical Relations:
- Definition of $\mathcal{V}[[\tau \rightarrow \sigma]]$ quantify over pairs of values in $\mathcal{V}[[\tau]]$,
- Future worlds $w' \sqsupseteq w$ can add arbitrary new systems of invariants.

Solutions:
- Reason on open terms with functional names (i.e. uninterpreted functions)
  - $\rightsquigarrow$ Make the full control flow apparent in the operational reduction,
  - $\rightsquigarrow$ Keep track of related functional names in an environment $e$, 

(Joint Work with N. Tabareau) (Published at APLAS'15)
Obstacles to automate Kripke Logical Relations:
- Definition of $\mathcal{V} [\tau \rightarrow \sigma]$ quantify over pairs of values in $\mathcal{V} [\tau]$, 
- Future worlds $w' \sqsubseteq w$ can add arbitrary new systems of invariants.

Solutions:
- Reason on open terms with functional names (i.e. uninterpreted functions)
  - $\rightsquigarrow$ Make the full control flow apparent in the operational reduction,
  - $\rightsquigarrow$ Keep track of related functional names in an environment $e$,
- Fix a “World Transition System” $\mathcal{A}$:
  - $\rightsquigarrow$ Transitions specify only the evolution of worlds
  - $\rightsquigarrow$ Transition system represents the control flow between the term and its environment.
Obstacles to automate Kripke Logical Relations:
- Definition of $\mathcal{V}[[\tau \rightarrow \sigma]]$ quantify over pairs of values in $\mathcal{V}[[\tau]]$.
- Future worlds $w' \sqsubseteq w$ can add arbitrary new systems of invariants.

Solutions:
- Reason on open terms with functional names (i.e. uninterpreted functions)
  - Make the full control flow apparent in the operational reduction,
  - Keep track of related functional names in an environment $e$,
- Fix a “World Transition System” $\mathcal{A}$:
  - Transitions specify only the evolution of worlds
  - Transition system represents the control flow between the term and its environment.
Obstacles to automate Kripke Logical Relations:

- Definition of $\mathcal{V}[[\tau \rightarrow \sigma]]$ quantify over pairs of values in $\mathcal{V}[[\tau]]$,
- Future worlds $w' \sqsupseteq w$ can add arbitrary new systems of invariants.

Solutions:

- Reason on open terms with functional names (i.e. uninterpreted functions)
  - Make the full control flow apparent in the operational reduction,
  - Keep track of related functional names in an environment $e$,
- Fix a “World Transition System” $\mathcal{A}$:
  - Transitions specify only the evolution of worlds
  - Transition system represents the control flow between the term and its environment.

Give rise to our definition of "Kripke Open Bisimulations" (Joint Work with N. Tabareau) (Published at APLAS’15)
Obstacles to automate Kripke Logical Relations:

- Definition of $\mathcal{V}[\tau \rightarrow \sigma]$ quantify over pairs of values in $\mathcal{V}[\tau]$,
- Future worlds $w' \sqsubseteq w$ can add arbitrary new systems of invariants.

Solutions:

- Reason on open terms with functional names (i.e. uninterpreted functions)
  - $\Rightarrow$ Make the full control flow apparent in the operational reduction,
  - $\Rightarrow$ Keep track of related functional names in an environment $e$,
- Fix a “World Transition System” $\mathcal{A}$:
  - $\Rightarrow$ Transitions specify only the evolution of worlds
  - $\Rightarrow$ Transition system represents the control flow between the term and its environment.

Give rise to our definition of “Kripke Open Bisimulations”
(Joint Work with N. Tabareau)
(Published at APLAS’15)
Relations on Terms

\[(M_1, M_2) \in \mathcal{E}_A [\tau]_e w \text{ when, for all } (h_1, h_2) \in w,\]
Relations on Terms

$$(M_1, M_2) \in \mathcal{E}_A \llbracket \tau \rrbracket_e w$$ when, for all \((h_1, h_2) \in w\),

- Either both \((M_1, h_1) \uparrow\) and \((M_2, h_2) \uparrow\),
(M_1, M_2) \in \mathcal{E}_\mathcal{A} [\tau]_e w \text{ when, for all } (h_1, h_2) \in w,

- Either both \((M_1, h_1) \uparrow\) and \((M_2, h_2) \uparrow\),

- Or both \((M_1, h_1) \mapsto^* (v_1, h'_1)\) and \((M_2, h_2) \mapsto^* (v_2, h'_2)\) and there exists \(w' \sqsupseteq \mathcal{A} w\) with \((h'_1, h'_2) \in w'\) and \((v_1, v_2) \in \mathcal{V}_\mathcal{A} [\tau]_e w'\).
(M₁, M₂) ∈ Eₐ [τ]ₑ w when, for all (h₁, h₂) ∈ w,
- Either both (M₁, h₁) ↑ and (M₂, h₂) ↑,
- Or both (M₁, h₁) ↦* (ν₁, h′₁) and (M₂, h₂) ↦* (ν₂, h′₂) and there exists w′ ⊒ₐ w with (h′₁, h′₂) ∈ w′ and (ν₁, ν₂) ∈ Vₐ [τ]ₑ w′,
- Or both (M₁, h₁) ↦* (K₁[f₁ ν₁], h′₁) and (M₂, h₂) ↦* (K₂[f₂ ν₂], h′₂) with (f₁, f₂, σ → σ′) ∈ e and there exists w′ ⊒ₐ w with (h′₁, h′₂) ∈ w′ and (K₁, K₂) ∈ Kₐ [σ′, τ]ₑ w′, (ν₁, ν₂) ∈ Vₐ [σ]ₑ w′
Relations on Terms

$(M_1, M_2) \in \mathcal{E}_A \llbracket \tau \rrbracket_e w$ when, for all $(h_1, h_2) \in w$,

- Either both $(M_1, h_1) \uparrow$ and $(M_2, h_2) \uparrow$,

- Or both $(M_1, h_1) \mapsto^* (v_1, h'_1)$ and $(M_2, h_2) \mapsto^* (v_2, h'_2)$ and there exists $w' \sqsupseteq_A w$ with $(h'_1, h'_2) \in w'$ and $(v_1, v_2) \in \mathcal{V}_A \llbracket \tau \rrbracket_e w'$,

- Or both $(M_1, h_1) \mapsto^* (K_1[f_1\ v_1], h'_1)$ and $(M_2, h_2) \mapsto^* (K_2[f_2\ v_2], h'_2)$ with $(f_1, f_2, \sigma \rightarrow \sigma') \in e$ and there exists $w' \sqsupseteq_A w$ with $(h'_1, h'_2) \in w'$ and $(K_1, K_2) \in \mathcal{K}_A \llbracket \sigma', \tau \rrbracket_e w'$, $(v_1, v_2) \in \mathcal{V}_A \llbracket \sigma \rrbracket_e w'$,

- (+ deferred divergence).
Soundness and Completeness

**Theorem**

Two terms $M_1, M_2$ of type $\tau$ are contextually equivalent iff there exists a WTS $A$ s.t. $(M_1, M_2) \in \mathcal{E}_A \left[\tau\right] w_0$.

- **Proof**: Correspondence with “simple” bisimulations on the LTS generating traces.

- **Soundness**:
  \[ \rightsquigarrow \text{WTS } A \text{ as an abstraction of the LTS generating traces.} \]

- **Completeness**:
  \[ \rightsquigarrow \text{Not for free: no more biorthogonality,} \]
  \[ \rightsquigarrow \text{Need } \textit{exhaustive} \text{ WTS,} \]
  \[ \rightsquigarrow \text{First complete method which does not rely on closure in its definition.} \]
Further in Abstraction: Symbolic and Temporal Reasoning

- $E_A[\tau]_w$ uses operational reduction
  - $\Rightarrow$ Replace it with symbolic execution,
  - $\Rightarrow$ Generate arithmetic constraints,
  - $\Rightarrow$ Block on callbacks and recursive calls.

- Quantification over future worlds
  - $\Rightarrow$ Abstract over it using temporal logic,
  - $\Rightarrow$ Temporal modalities $\Box \phi, X(\phi)$.

- Give rise to Temporal Symbolic Kripke Open Bisimulations $E[\tau]$.
  - $\Rightarrow$ Model check $A \models E[\tau]$. 
An Example: Callback with lock

let \( b_1 = \text{ref} \, \text{true} \), \( c_1 = \text{ref} \, 0 \),

fun inc_1(f) = if !b_1 then \{
  b_1 := \text{false};
  f(); \!c_1 := \!c_1 + 1;
  b_1 := \text{true}
\} \text{ else } (),

fun get_1() = \!c_1

in \langle \text{inc}_1, \text{get}_1 \rangle

let \( b_2 = \text{ref} \, \text{true} \), \( c_2 = \text{ref} \, 0 \),

fun inc_2(f) = if !b_2 then \{
  b_2 := \text{false};
  let n = \!c_2 \text{ in } f(); \!c_2 := n + 1;
  b_2 := \text{true}
\} \text{ else } (),

fun get_2() = \!c_2

in \langle \text{inc}_2, \text{get}_2 \rangle
An Example: Callback with lock

let b₁ = ref true, c₁ = ref 0,
fun inc₁(f) = if !b₁ then \{ b₁ := false; 
    f(); c₁ := !c₁ + 1; 
    b₁ := true \} else (),
fun get₁() = !c₁ 
in ⟨inc₁, get₁⟩

let b₂ = ref true, c₂ = ref 0,
fun inc₂(f) = if !b₂ then \{ b₂ := false; 
    let n = !c₂ in f(); c₂ := n + 1; 
    b₂ := true \} else (),
fun get₂() = !c₂ 
in ⟨inc₂, get₂⟩
\( E[\tau](M_{1}^{clb}, M_{2}^{cbl}) \) is equal to

\[
\begin{align*}
&\mathcal{H}N_{0}.(N_{1} \mathcal{l}_{2}.(N_{1} \mathcal{l}_{1}.(N_{2} \mathcal{l}_{4}.(N_{2} \mathcal{l}_{3}.(X((l_{2} \mapsto 1 0) \land (l_{1} \mapsto 1 \text{true}) \land (l_{4} \mapsto 2 0) \land (l_{3} \mapsto 2 \text{true}) \land \\
(\square(\mathcal{H}N_{5}.(\forall x_{6}, x_{7}, x_{8}, x_{9}.(((l_{2} \mapsto 1 x_{6}) \land (l_{1} \mapsto 1 x_{7}) \land (l_{4} \mapsto 2 x_{8}) \land (l_{3} \mapsto 2 x_{9}))) \Rightarrow \\
((X(((x_{7} = \text{true}) \land (x_{9} = \text{true})) \Rightarrow ((l_{2} \mapsto 1 x_{6}) \land (l_{1} \mapsto 1 \text{false}) \land (l_{4} \mapsto 2 x_{8}) \land (l_{3} \mapsto 2 \text{false}) \land \\
(\square_{\text{pub}}(\forall x_{10}, x_{11}, x_{13}, x_{14}.(((l_{2} \mapsto 1 x_{10}) \land (l_{1} \mapsto 1 x_{11}) \land (l_{4} \mapsto 2 x_{13}) \land (l_{3} \mapsto 2 x_{14}))) \Rightarrow \\
(X(\forall x_{12}, x_{15}.(((x_{12} = x_{10} + 1) \land (x_{15} = x_{8} + 1)) \Rightarrow ((l_{2} \mapsto 1 x_{12}) \land (l_{1} \mapsto 1 \text{true}) \land \\
(l_{4} \mapsto 2 x_{15}) \land (l_{3} \mapsto 2 \text{true}) \land (P_{\text{pub}}(N_{5})))))))))))) \land (\text{not}((x_{7} = \text{true}) \land (x_{9} = \text{false})))) \land \\
(\text{not}((x_{7} = \text{false}) \land (x_{9} = \text{true}))) \land (X(((x_{7} = \text{false}) \land (x_{9} = \text{false})) \Rightarrow \\
((l_{2} \mapsto 1 x_{6}) \land (l_{1} \mapsto 1 x_{7}) \land (l_{4} \mapsto 2 x_{8}) \land (l_{3} \mapsto 2 x_{9}) \land (P_{\text{pub}}(N_{5})))))))) \land \\
(\square(\mathcal{H}N_{1} 6.(\forall [x_{17}, x_{18}, x_{19}, x_{20}].(((l_{2} \mapsto 1 x_{17}) \land (l_{1} \mapsto 1 x_{18}) \land (l_{4} \mapsto 2 x_{19}) \land (l_{3} \mapsto 2 x_{20})) \Rightarrow \\
(X((l_{2} \mapsto 1 x_{17}) \land (l_{1} \mapsto 1 x_{18}) \land (l_{4} \mapsto 2 x_{19}) \land (l_{3} \mapsto 2 x_{20}) \land (x_{17} = x_{19}) \land (P_{\text{pub}}(N_{1} 6)))))))) \land \\
(P_{\text{pub}}(N_{0}))))))))
\end{align*}
\]
(assert (exists ((s21 Int)(h22 Heap)(h23 Heap)(12 Int)(11 Int)) (and (not (= 11 12)))
(exists ((14 Int)(13 Int)) (and (not (= 13 14)) (exists ((s25 Int)(h26 Heap)(h27 Heap)(S28 LocSpan))
(and (TransPriv s21 s25 h22 h23 h26 h27 S28) (and (= (select h26 12) 0) (= (select h26 11) 0)
(= (select h27 14) 0) (= (select h27 13) 0) (and (forall ((s29 Int)(h30 Heap)(h31 Heap)(S32 LocSpan))
(=> (TransPrivT s25 s29 h26 h27 h30 h31 S32) (forall ((x6 Int)(x7 Int)(x8 Int)(x9 Int))
(=> (and (= (select h30 12) x6) (= (select h30 11) x7) (= (select h31 14) x8) (= (select h31 13) x9) )
(and (exists ((s33 Int)(h34 Heap)(S36 LocSpan)) (and (TransPriv s29 s33 h30 h34 h35 S36)
(=> (and (= x7 0) (= x9 0) ) (and (= (select h34 12) x6) (= (select h34 11) 1) (= (select h35 14) x8)
(= (select h35 13) 1) (forall ((s37 Int)(h38 Heap)(S40 LocSpan))
(=> (TransPubT s33 s37 h34 h35 h38 h39 S40) (forall ((x10 Int)(x11 Int)(x13 Int)(x14 Int))
(=> (and (= (select h38 12) x10) (= (select h38 11) x11) (= (select h39 14) x13) (= (select h39 13) x12)
(exists ((s41 Int)(h42 Heap)(S44 LocSpan)) (and (TransPriv s37 s41 h38 h39 h42 h43 S44)
(forall ((x12 Int)(x15 Int)) (=> (and (= x12 (+ x10 1)) (= x15 (+ x8 1))) )
(=> (TransPub s41 s42 h42 h43 S44))))))))))))
(not (and (= x7 0) (= x9 1)) (not (and (= x7 1) (= x9 0)))
(exists ((s45 Int)(h46 Heap)(S48 LocSpan)) (and (TransPriv s29 s45 h30 h31 h46 h47 S48)
(=> (and (= x7 1) (= x9 1)) (and (= (select h46 12) x6) (= (select h46 11) x7) (= (select h47 14) x8)
(= (select h47 13) x9) (TransPub s29 s45 h30 h46 h47 S48))))))))
(forall ((s49 Int)(h50 Heap)(S52 LocSpan)) (=> (TransPrivT s25 s49 h26 h27 h50 h51 S52)
(forall ((x17 Int)(x18 Int)(x19 Int)(x20 Int)) (=> (and (= (select h50 12) x17) (= (select h50 11) x18)
(= (select h51 14) x19) (= (select h51 13) x20) )
(exists ((s53 Int)(h54 Heap)(S56 LocSpan)) (and (TransPriv s49 s53 h50 h51 h54 h55 S56)
(= (select h55 14) x19) (= (select h55 13) x20) (= x17 x19) (TransPub s49 s53 h50 h51 h54 h55 S56)))))))
(TransPub s21 s25 h22 h23 h26 h27 S28))))))))

(check-sat)
Going further: SyTeCi

- Combine Symbolic, Temporal with Circular reasoning
  - Deal with recursive functions,
  - Synchronization of recursive calls,
  - Simple heuristics to unfold recursive functions,
  - Circular proof system.

- Computation of the transitive closure of the WTS $\mathcal{A}$
  - Undecidable in general,
  - Over-approximations via abstractions.

- Automatic Computation of $\mathcal{A}$. 