From Bounded Checking to Verification of Equivalence via Symbolic Up-To Techniques

Yu-Yang Lin (TCD)

joint work with Vasileios Koutavas (TCD) and Nikos Tzevelekos (QMUL)

Implementation Available

Tool called “Hobbit”: https://github.com/LaifsV1/Hobbit

Contextual Equivalence

A relation over **program terms** which holds when the related terms are **interchangeable in any program context**.[Morris'68]

i.e. $M \equiv N$,

if for every context C $\quad \boxed{C\ \boxed{M}} \longrightarrow n \quad$ iff $\quad \boxed{C\ \boxed{N}} \longrightarrow n$

Setting: higher-order language with local state (e.g. like ML)

Challenges:

- Equivalence problem is undecidable
- Many infinities: **infinitely** many contexts to enumerate, **infinitely large** contexts
  - base-type domains (e.g. integers), arithmetic, recursion, ...
- Unknown code for higher-order types
- Existing Theory made for manual proofs, not practical algorithmic verification

Overview of Our Approach

1) Labelled Transition System (LTS)

2) Environmental Bisimulation on the LTS

3) Bounded Model Checking

4) Bisimulation up-to techniques to finitise exploration

Labelled Transition System

Uses **symbolic higher-order values** based on Game Semantics

Two-player game between **program** and **environment**:

- **Proponent:** program term
- **Opponent:** environment
- Moves are **applications** (app / app) and **returns** (ret / ret)

Bisimulation Checking: exploration of traces in the product graph (deterministic setting)

Each trace is a sequence of moves

Bounded Model Checking

Depth-bounded *Symbolic Execution* of the Environmental Bisimulation

- Uses **symbolic first-order values**
- **Precise** and **Exhaustive** exploration up to the bound
- Returns:
  - true positive (equivalent)
  - true negative (not equivalent)
  - bound exhausted

Good* for inequivalences, but what if we want to prove equivalences?

*: deterministic setting

Verifying Equivalence: Dealing with Infinities

Infinities emerging from the Opponent:

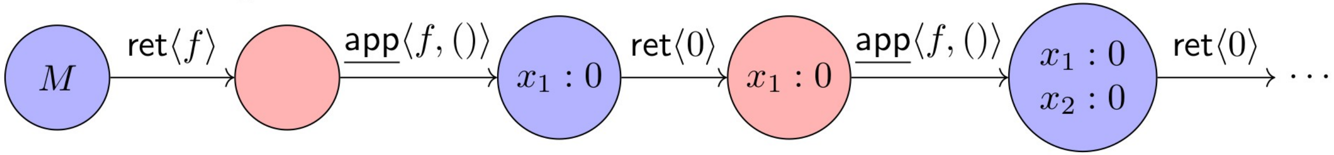
- Opponent may **(A)** **sequence calls** or **(B)** **nest calls** to the same function

Cycle Detection:

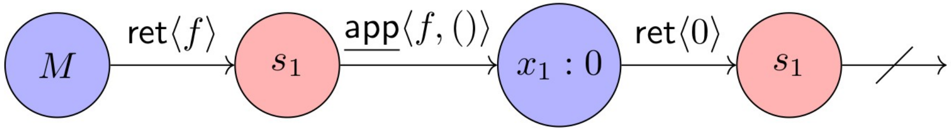
- Memoisation: a cycle exists if we arrive at a configuration previously seen
- Normalisation: configurations may be identical up to permutation and alpha-equivalence
- Garbage collection: identical configurations may differ in unused locations

$M =$ `fun () -> ref x = 0 in !x` $N =$ `fun () -> 0`

Without Garbage Collection:



With Garbage Collection:



Bisimulation Up-To Techniques

Memoisation + Normalisation + GC rarely finds enough cycles to finitise the exploration

Up-to techniques: powerful theoretical techniques for hand-written proofs

- Advancements had not been integrated in verification tools before

Bisimulation:

$C1 \approx C2$ iff

- $C1 \Rightarrow C1'$ implies $\exists C2'.\ C2 \Rightarrow C2'$ and $C1' \approx C2'$
- Vice-versa

Bisimulation up to X:

$C1 \approx C2$ iff

- $C1 \Rightarrow C1'$ implies $\exists C2'.\ C2 \Rightarrow C2'$ and $C1'\ X(\approx)\ C2'$
- Vice-versa

Novel Bisimulation Up-To Techniques

Many existing theoretical up-to techniques **seem to be less useful for our setting**

More techniques are needed to address many examples of infinite LTS's

We introduce **three novel up-to techniques**:

- **Up to Separation:** inspired by the *frame rule* in *separation logic*
- **Up to Re-entry:** avoids re-entry of functions that do not affect the state
- **Up to Invariants:** uses *state invariants*

Up to Separation

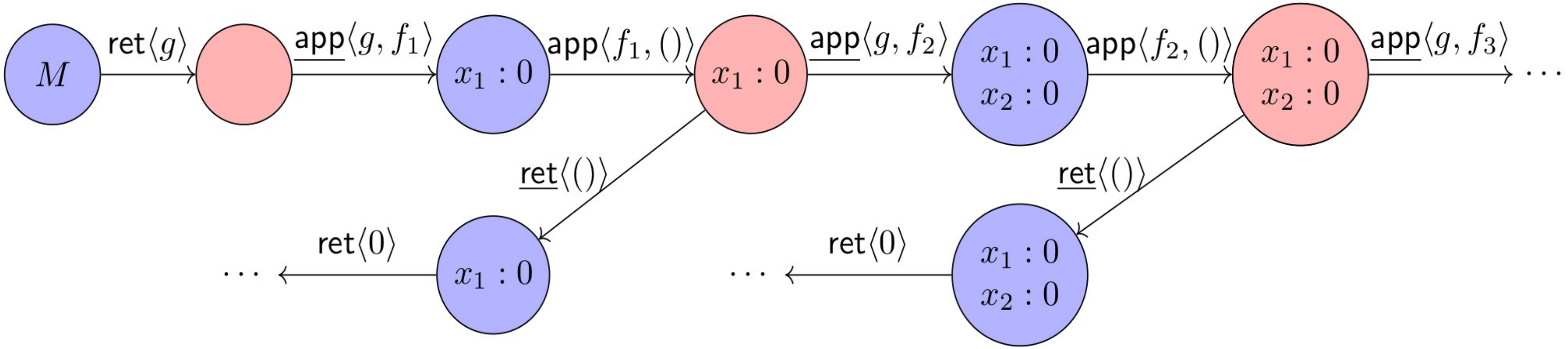
Intuition: function calls that explore different parts of the state can be explored independently

Corollary: a function that manipulates only its local state can be explored independently from itself

i.e. it suffices to call functions without shared state once

e.g. $M =$ `fun f -> ref x = 0 in f (); !x` $N =$ `fun f -> f (); 0`

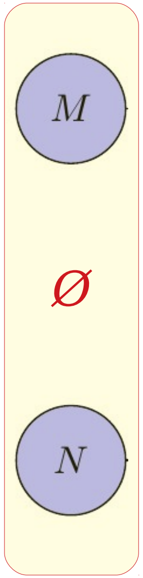
Without Separation:



Up to Separation: Example

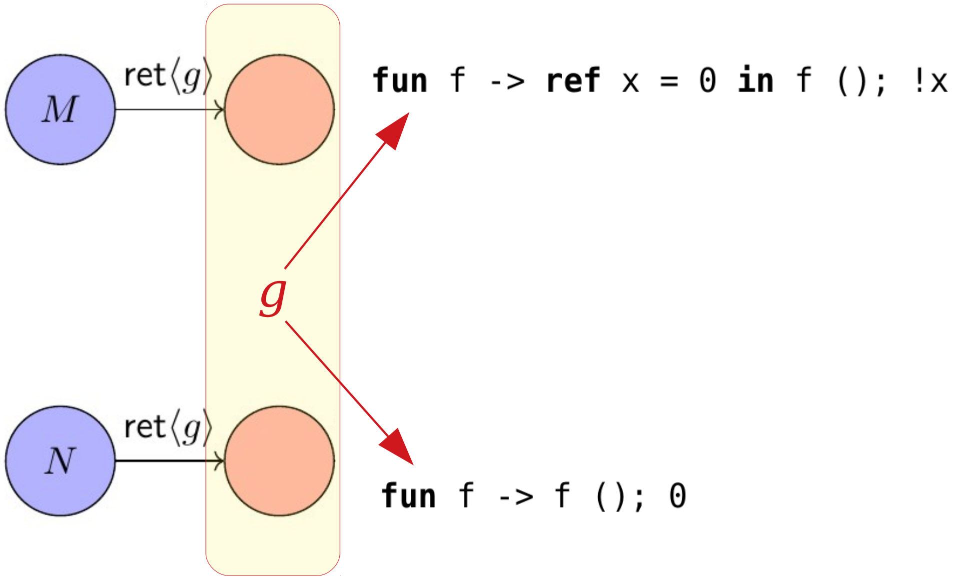
$M =$ `fun f -> ref x = 0 in f (); !x` $N =$ `fun f -> f (); 0`

Environment's Knowledge



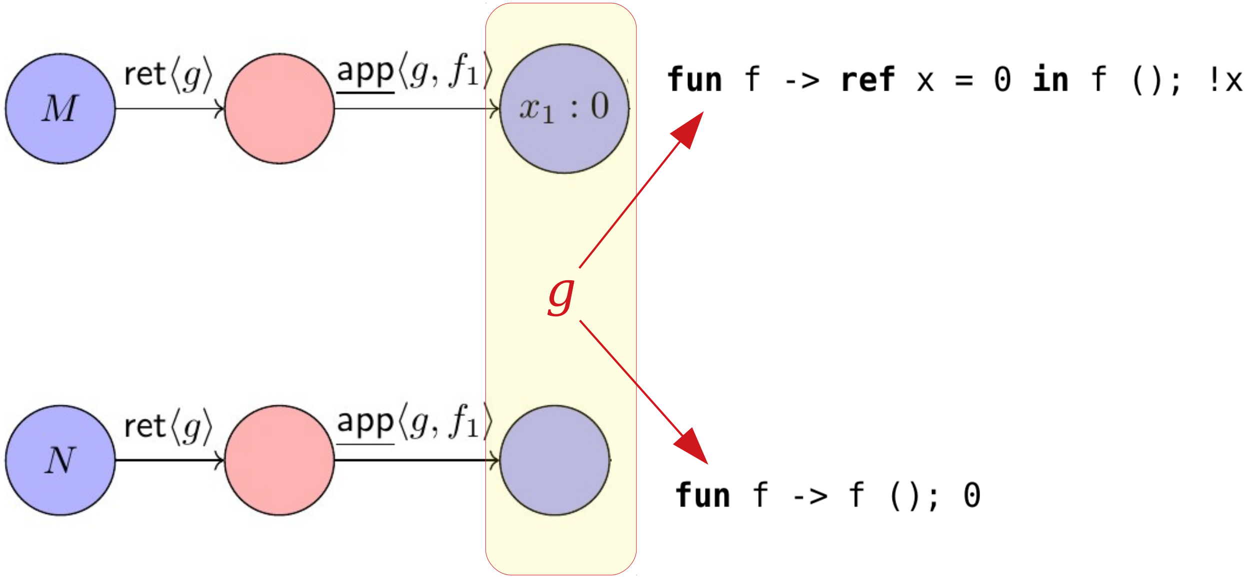
Up to Separation: Example

$M =$ `fun f -> ref x = 0 in f (); !x` $\qquad N =$ `fun f -> f (); 0`



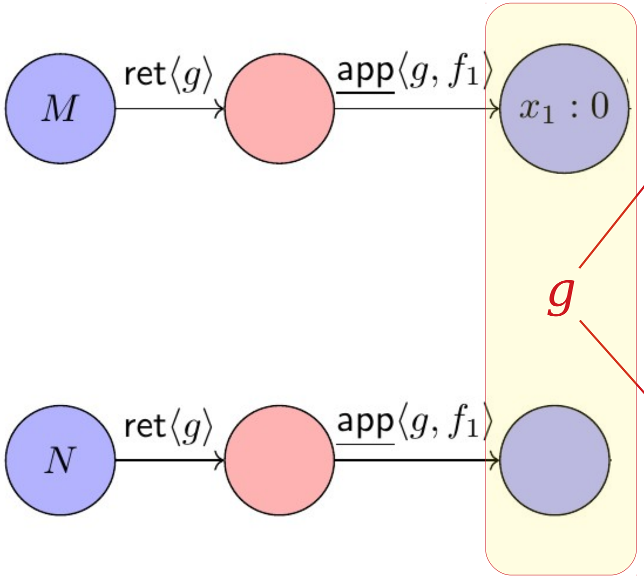
Up to Separation: Example

$M =$ **fun** f -> **ref** x = 0 **in** f (); !x $\qquad N =$ **fun** f -> f (); 0



Up to Separation: Example

$M =$ **fun** f -> **ref** x = 0 **in** f (); !x $\quad\quad N =$ **fun** f -> f (); 0



uses only local reference ‘x’

```
fun f -> ref x = 0 in f (); !x
```

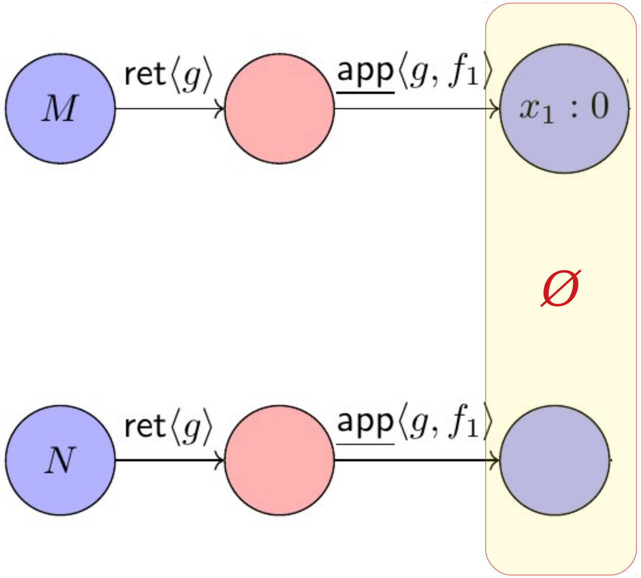
neither function body manipulates references shared with it

```
fun f -> f (); 0
```

no references used

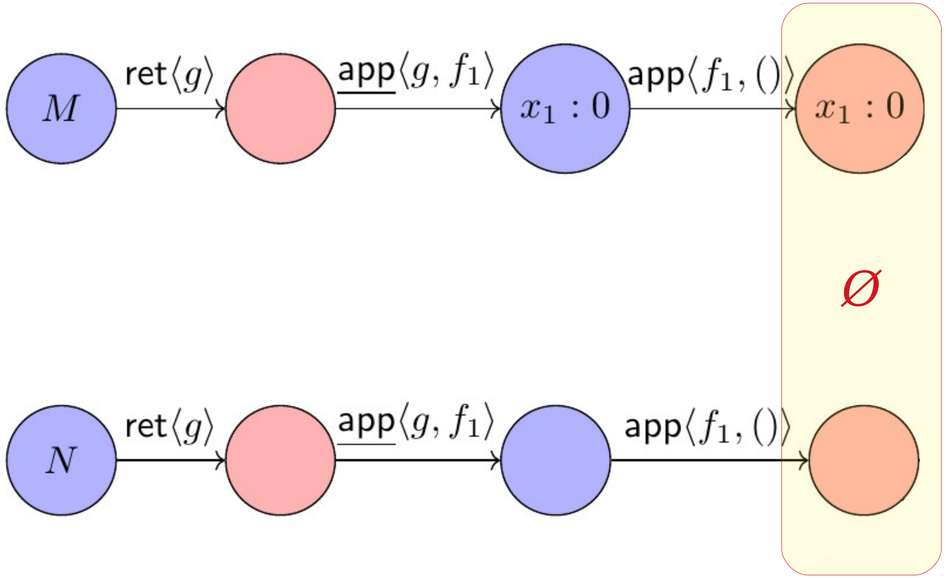
Up to Separation: Example

$M =$ **fun** `f -> ` **ref** `x = 0` **in** `f (); !x` $N =$ **fun** `f -> f (); 0`



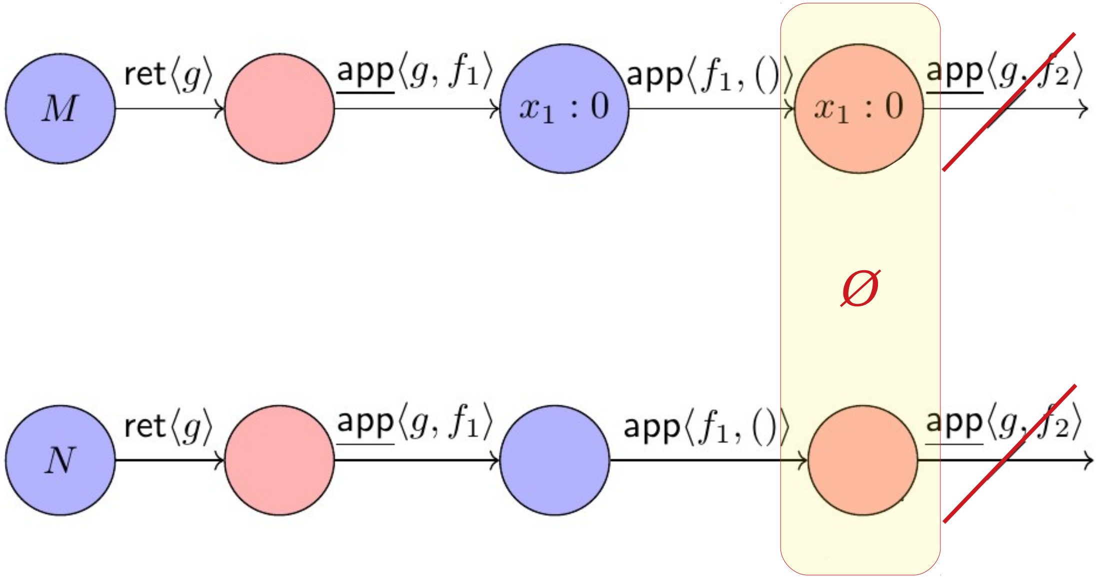
Up to Separation: Example

$M =$ **fun** f -> **ref** x = 0 **in** f (); !x $\qquad N =$ **fun** f -> f (); 0



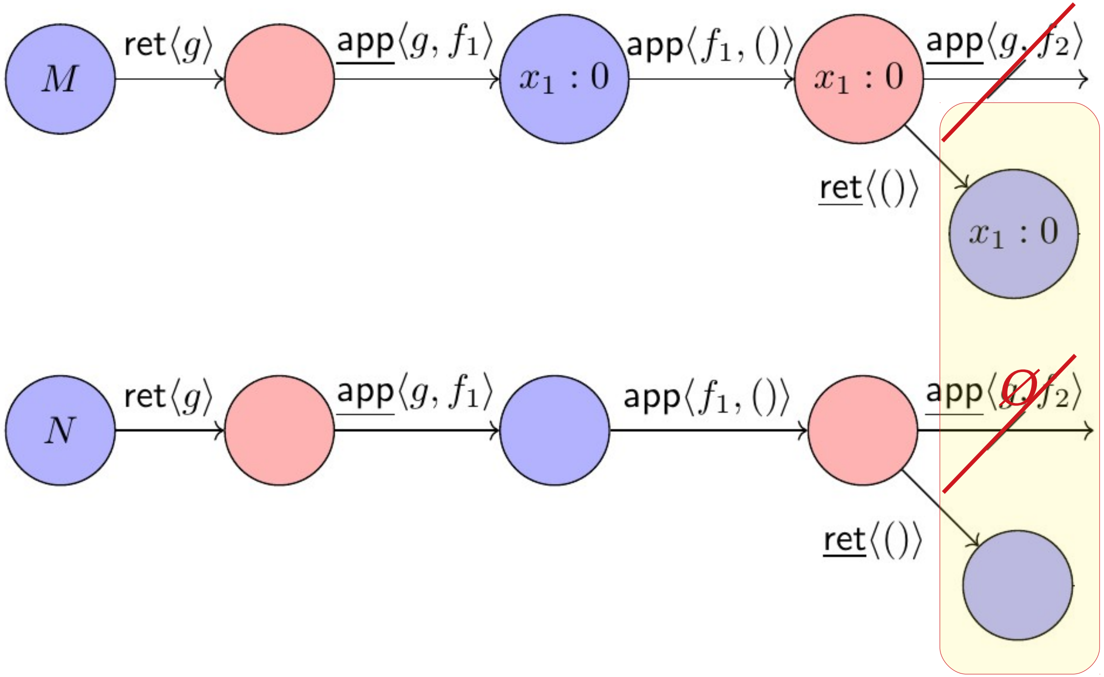
Up to Separation: Example

$M =$ **fun** f -> **ref** x = 0 **in** f (); !x $\qquad N =$ **fun** f -> f (); 0



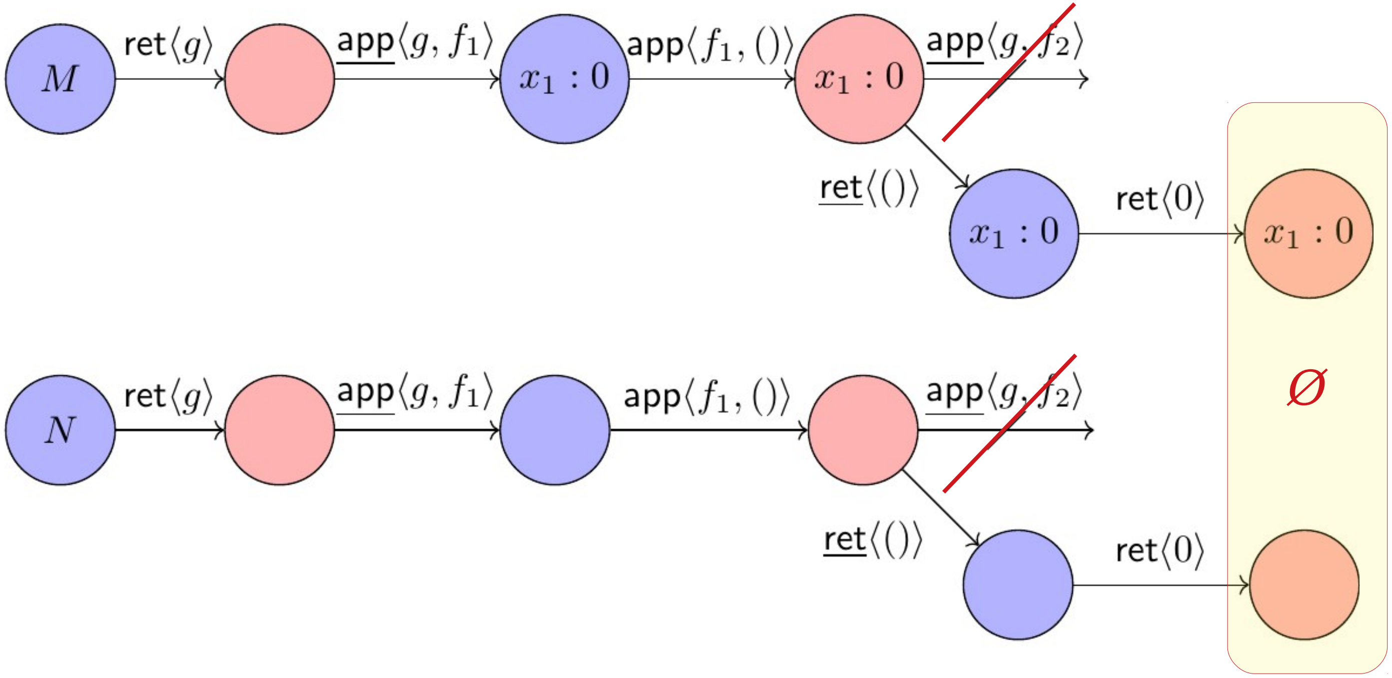
Up to Separation: Example

$M =$ **fun** f -> **ref** x = 0 **in** f (); !x $\qquad N =$ **fun** f -> f (); 0



Up to Separation: Example

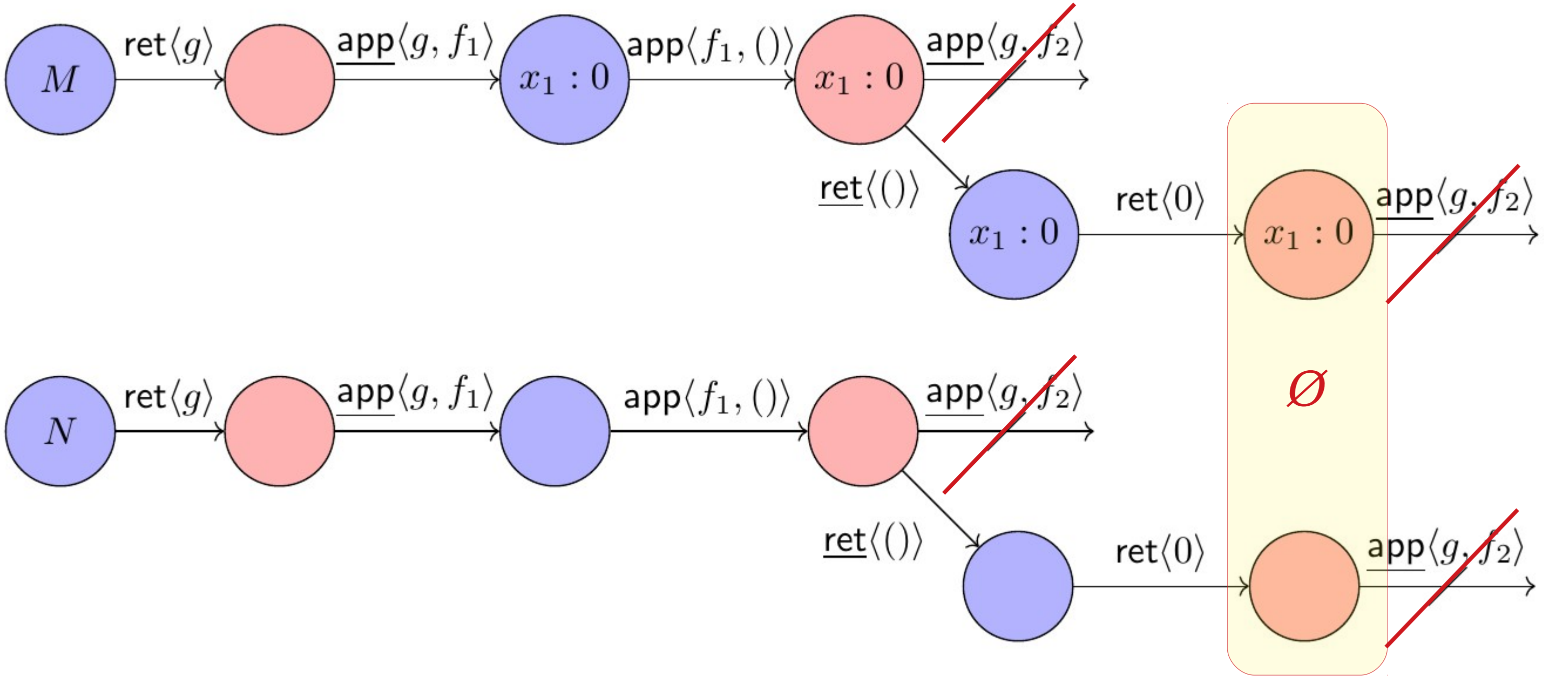
$M =$ `fun f -> ref x = 0 in f (); !x` $N =$ `fun f -> f (); 0`



Up to Separation: Example

$M =$ **fun** `f -> ` **ref** `x = 0` **in** `f (); !x`

$N =$ **fun** `f -> f (); 0`



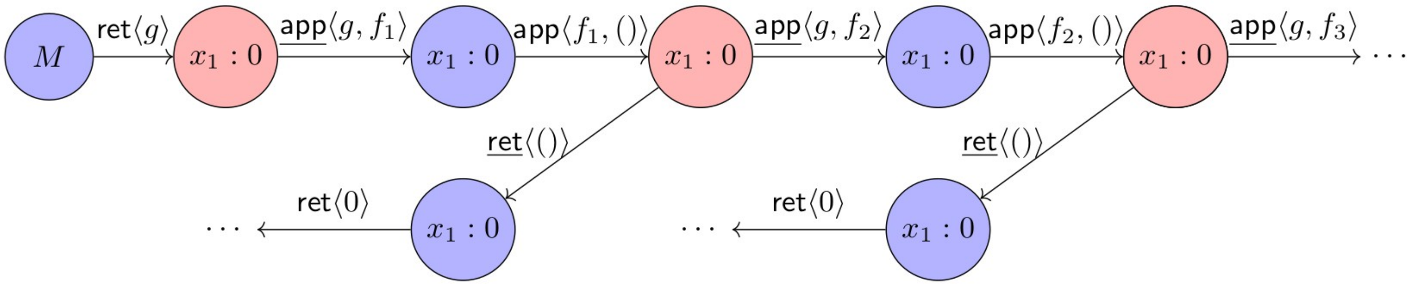
Up to Function Re-entry

Intuition: when applying a previously seen call, if the state at the nested call is equivalent to the state at the original call and the state does not change from the original call, we can skip the nested call.

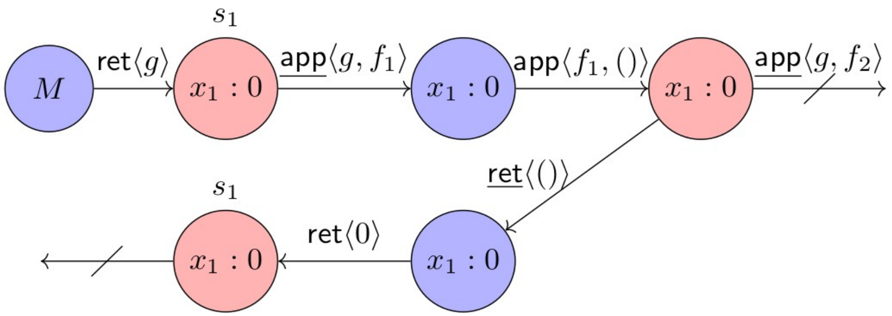
e.g.

$M =$ `ref x = 0 in fun f -> f (); !x` $\qquad N =$ `fun f -> f (); 0`

Without Up to Re-entry:



With Up to Re-entry:



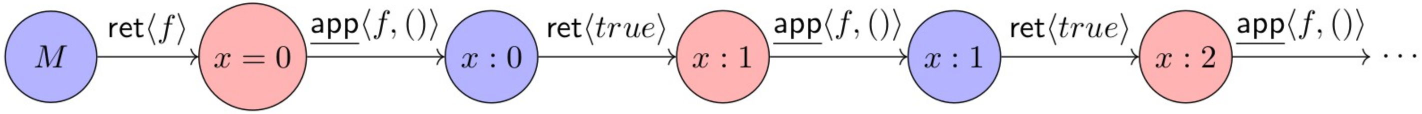
Up to State Invariants

Intuition: values stored in references can be abstracted using invariants

- transforms configurations into *classes of configurations* that satisfy the invariants
- user annotations: we do not try to automatically infer invariants

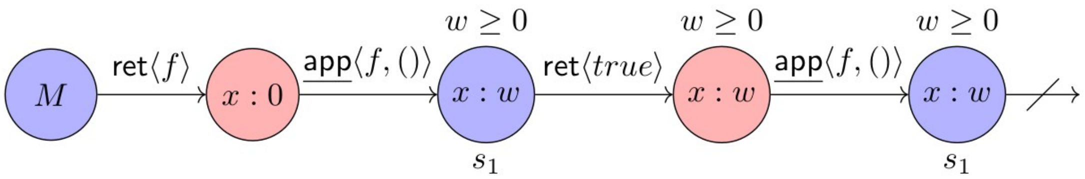
e.g. **Without Abstraction:**

$M =$ `ref x = 0 in fun () -> x++; !x > 0`
$N =$ `fun () -> true`



With Abstraction Invariant: $\exists w.(!x = w) \wedge (w \geq 0)$

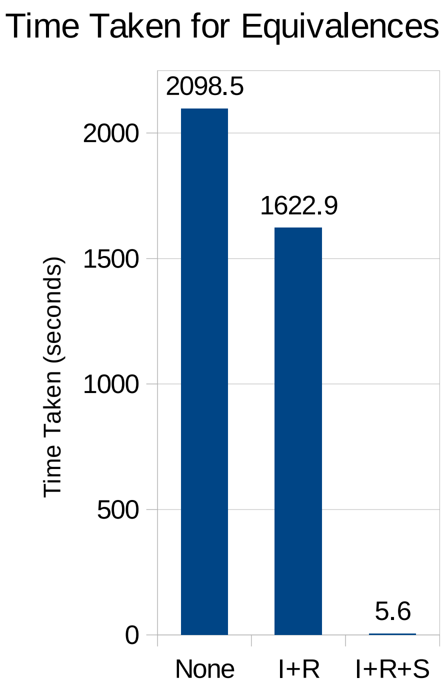
$M =$ `ref x = 0 in fun () { w | x as w | w >= 0 } -> x++; !x > 0`
$N =$ `fun () -> true`

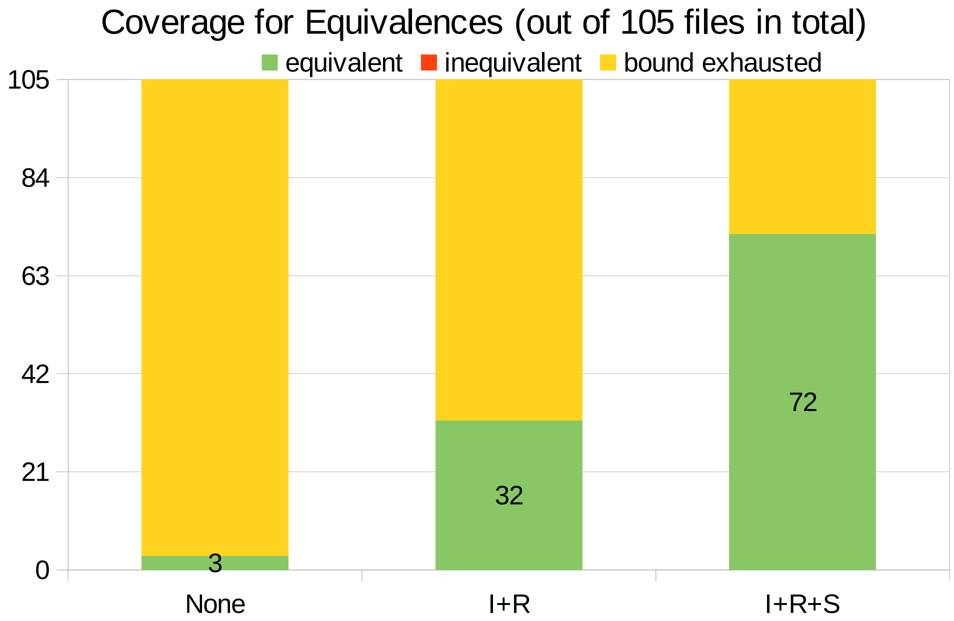


Performance of Up-To Techniques: Equivalences

Equivalences: 105 examples (~3300 lines of code)

- including all the Meyer and Sieber examples

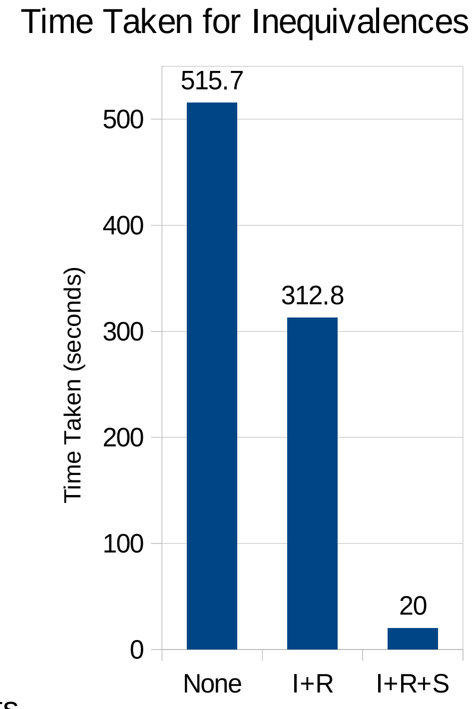


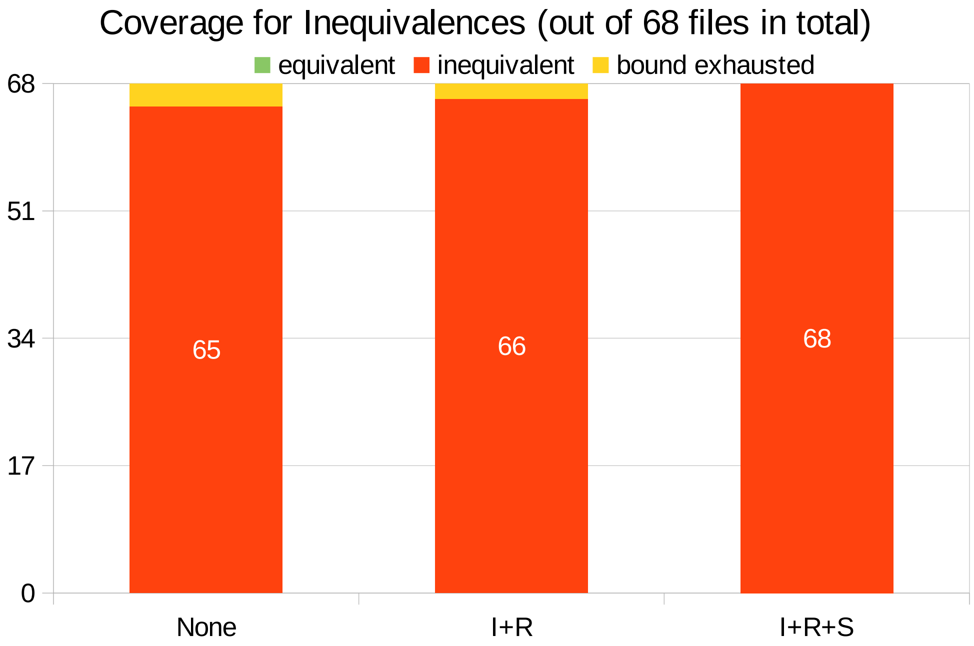


I: Up-To Invariants
R: Up-To Reentry
S: Up-To Separation

Performance of Up-To Techniques: Inequivalences

Inequivalences: 68 examples (~2300 lines of code)





I: Up-To Invariants
R: Up-To Reentry
S: Up-To Separation

Related Work

SyTeCi: for stateful higher-order terms

- Proves equivalences that Hobbit cannot (e.g. well-bracketed state)
  - but Hobbit proves equivalences that SyTeCi cannot
- Guarantees soundness only on a **subset of the language**
- Slower than Hobbit; Horn clauses also harder to solve than CNF SAT

Rêve, SymDiff, and RVT: first-order C variants with global state

- **Only first-order language** with **global state** is a simpler setting
- Better at complexities arising from internal term transitions (e.g. recursion)

We are working on technology for internal recursion, well-bracketing, and more state invariants

Thank You!

Hobbit – **H**igher-**O**rder **B**ounded **Bi**simulation **T**ool:

https://github.com/LaifsV1/Hobbit