Hobbit:
A Tool for Contextual Equivalence Checking Using Bisimulation Up-to Techniques

Vasileios Koutavas
Yu-Yang Lin
Trinity College Dublin

Nikos Tzevelekos
Queen Mary
University of London

Contextual Equivalence of Higher-Order Programs

A relation over program terms which holds when the related terms are interchangeable in any program context.

Setting: higher-order language with local state

Challenge: Program Contexts

- Need to model the interface between a program and its environment
- Potentially infinitely many contexts to enumerate (on top of redundancy)
- Higher-order terms with local state allow non-trivial function re-entrancy

Approach: Symbolic Environmental Bisimulations

- Environmental Bisimulations
- (Symbolic Execution) Game Semantics

Symbolic Environmental Bisimulation

Interface between program and environment:

- modelled as a two-player game
  - **Proponent:** program term
  - **Opponent:** environment
- moves are **applications** (app / app) and **returns** (ret / ret)

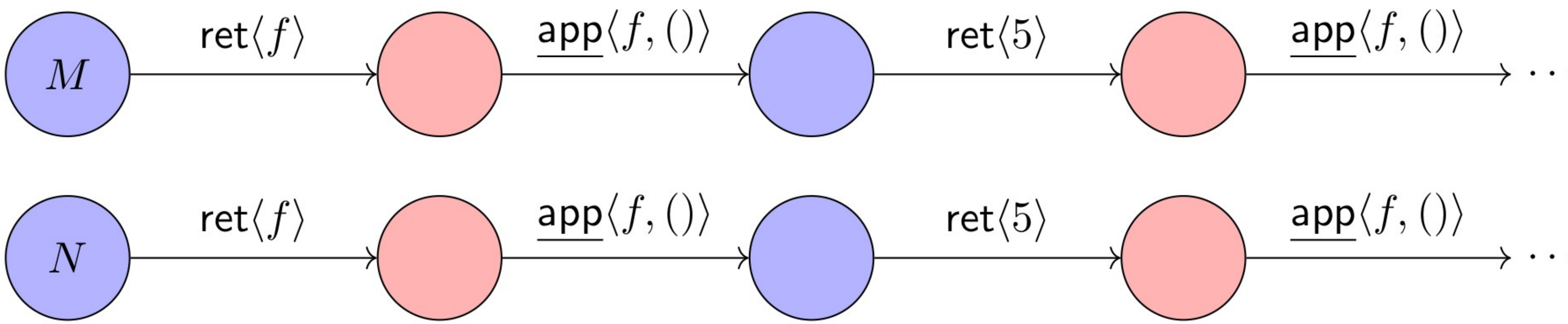
Each path in the computation tree is a trace

Each trace is a sequence of moves

Simple Examples

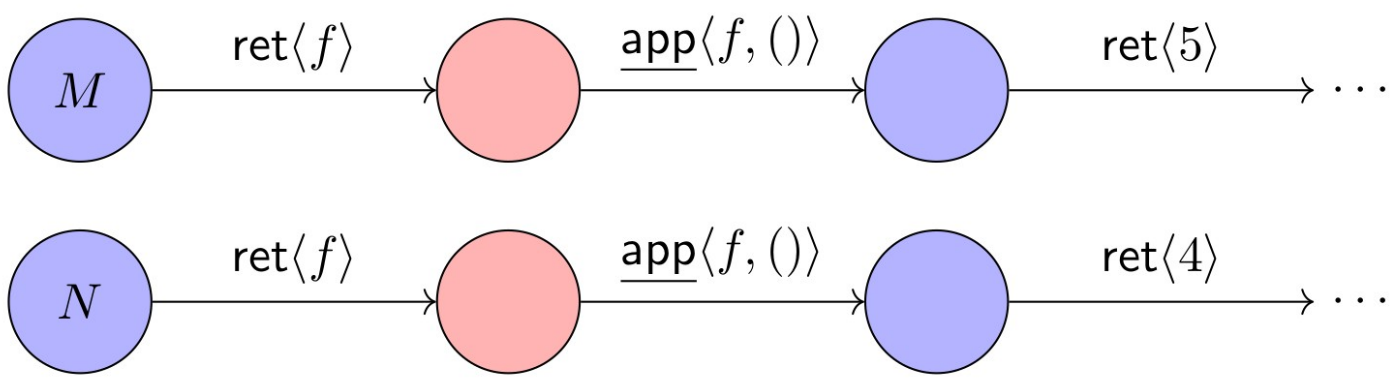
Equivalence:

$M =$ `fun () -> (); 5` $\qquad N =$ `fun () -> 5`



Inequivalence:

$M =$ `fun () -> (); 5` $\qquad N =$ `fun () -> 4`



Bounded Model Checking

Hobbit is a *Bounded Model Checking (BMC)* tool based on depth-bounded *Symbolic Execution* of the Bisimulation Game

- **Precise exploration** of the computation tree up to the bound (no false positives or negatives)
- **Exhaustive exploration** the computation tree up to the bound

***Good* for inequivalences:** counterexamples are relatively easy to find

- Hobbit found 68 out of 68 inequivalences, taking under 0.3s per inequivalence on average, with bounds of up to 202 calls

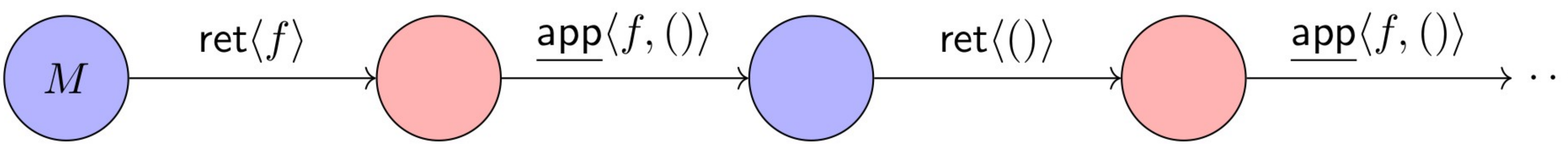
But what if we want to prove equivalences?

Proving Equivalence

Challenge: requires exploring potentially infinite computation trees

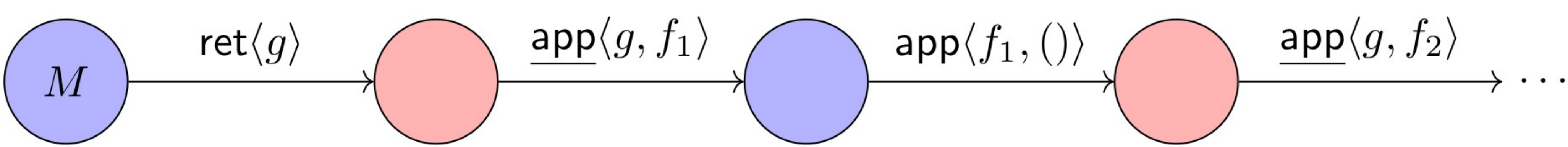
- environment may call repeatedly with changing state

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



- evaluation stack may grow indefinitely (e.g. re-entrant calls)

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



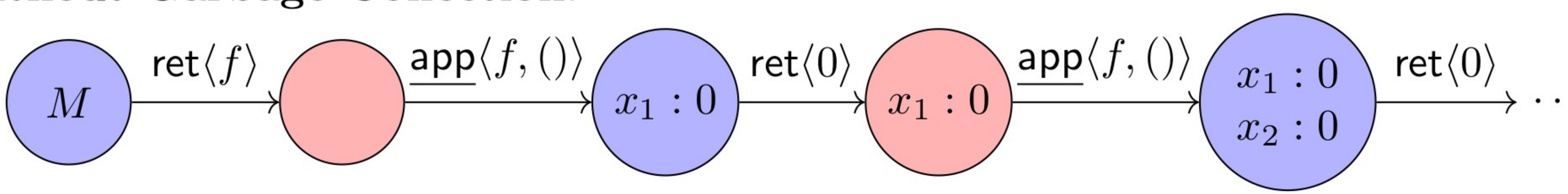
Finitising the Exploration

One Approach: loop detection

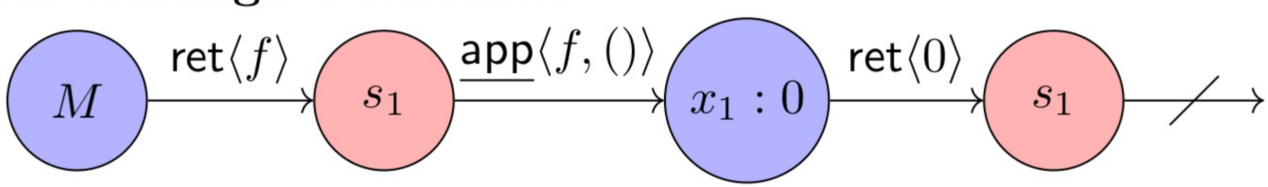
- memoisation: a loop exists if we arrive at a configuration previously seen
- normalisation: configurations may be identical up to permutation and alpha-equivalence
- garbage collection: identical configurations may differ in unused locations

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

Without Garbage Collection:



With Garbage Collection:



Up-To Techniques

Memoisation alone rarely finds enough loops to finitise the exploration:

Up-to techniques: powerful theoretical techniques for hand-written proofs of contextual equivalence, especially useful in higher-order terms with state

- Reduces state space of the exploration
- Advancements yet to be integrated in verification tools

Hobbit implements standard and 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 lightweight *state invariants*

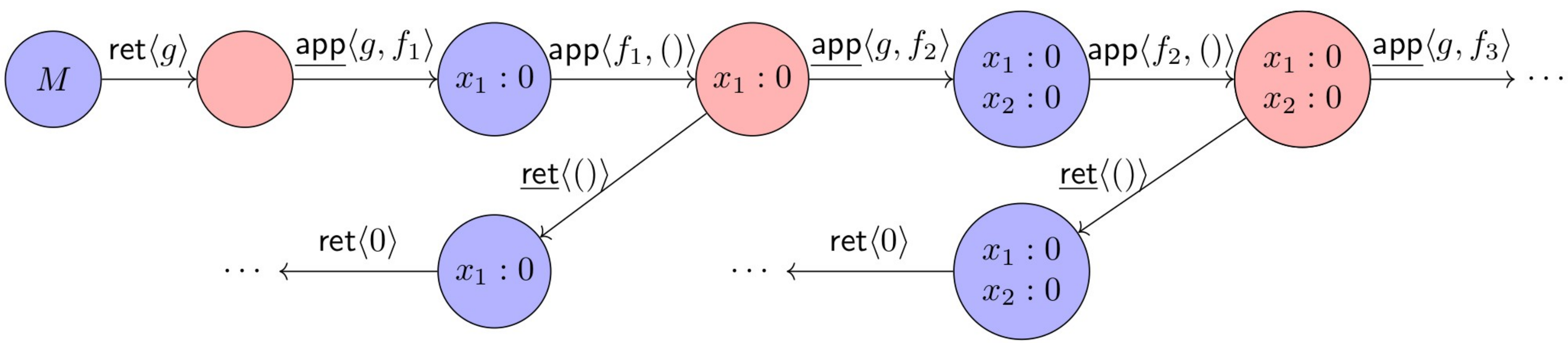
Up to Separation

Intuition: Functions that explore separate regions of the state can be explored independently

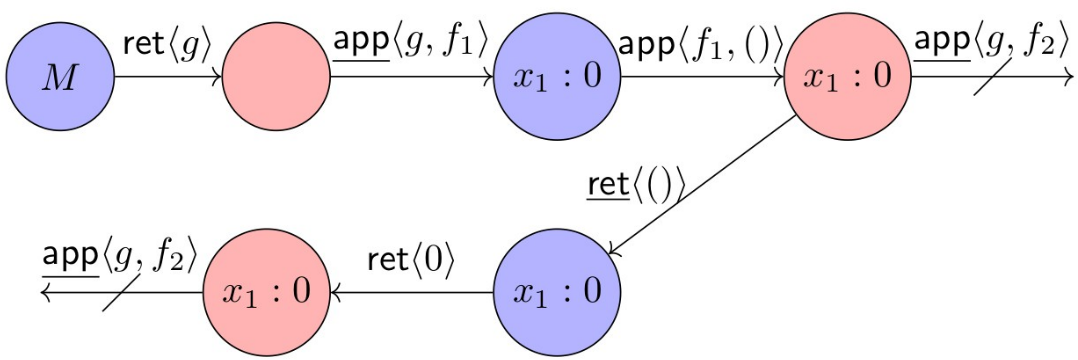
- **Corollary:** a function that manipulates only its own local references may be explored independently of itself; i.e. it suffices to call it once

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

Without Separation:



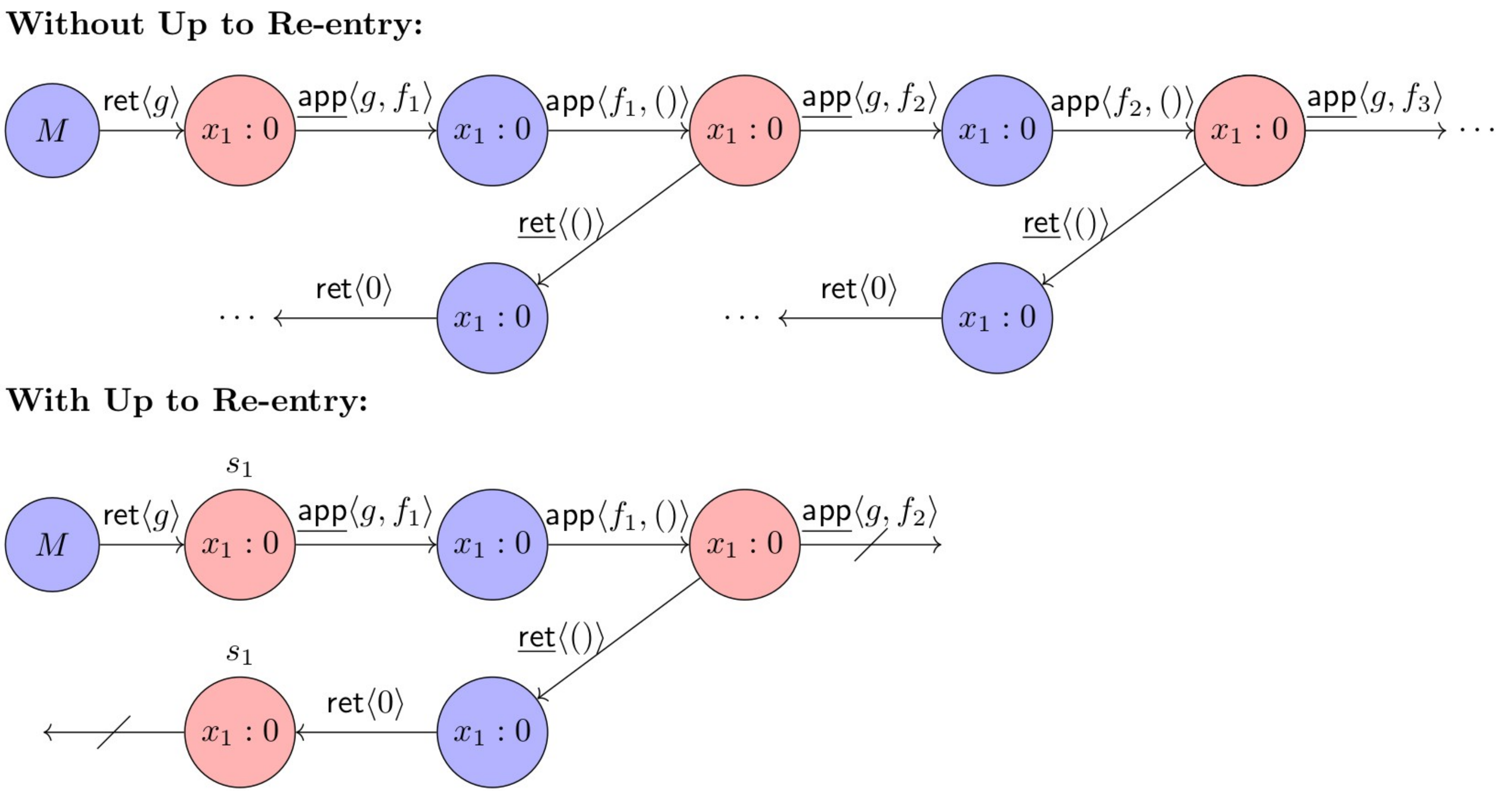
With Separation:



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, we can skip the nested call.

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



Up to State Invariants

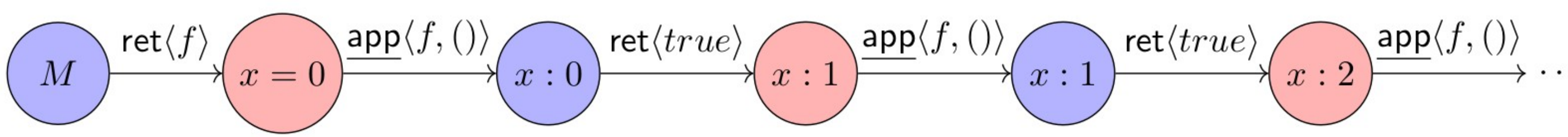
Intuition: values stored in references can be abstracted using invariants

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

e.g.

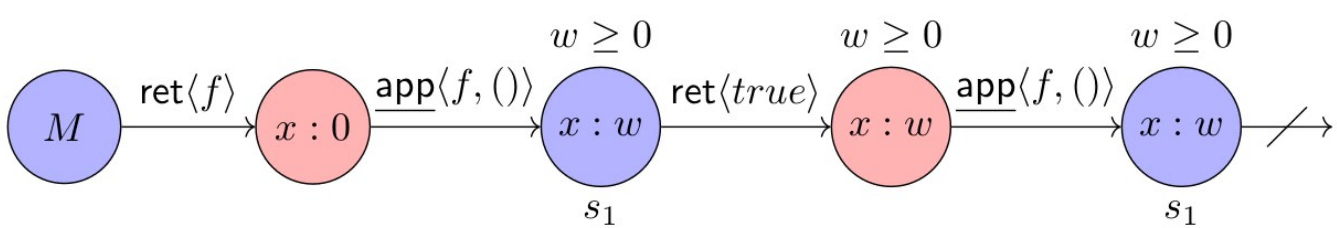
Without Abstraction:

$M =$ `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`



Demo: Equivalence

Example from Meyer and Sieber:

- adapted to Hobbit's local references by encapsulating references
- full example in the Technical Report

$M =$
```
let loc_eq loc1loc2 = [...] in
fun q -> ref x = 0 in
         let locx = (fun () -> !x) , (fun v -> x := v) in
         let almostadd_2 locz {w | x as w | w mod 2 == 0} =
           if loc_eq (locx,locz) then x := 1 else x := !x + 2
         in q almostadd_2; if !x mod 2 = 0 then _bot_ else ()
```

$N =$
```
fun q -> _bot_
```

Demo: Inequivalence

Example from the Ethereum blockchain:

- simplified withdraw function adapted from The DAO

$M =$
```
ref funds = 100 in
let withdraw1 =
 (fun send1 -> (if not(!funds < 1)
                then (send1 (); funds := !funds - 1)
                else ());
                !funds)
in withdraw1
```

$N =$
```
ref funds = 100 in
let withdraw1 =
 (fun send1 -> (if not(!funds < 1)
                then (funds := !funds - 1; send1 ())
                else ());
                !funds)
in withdraw1
```

Performance of Up-to Techniques

Equivalences: 105 examples (~3900 lines of code)

- **up-to techniques off:** 2098.5s, 3 equivalences found
- **up-to techniques on:** 5.6s, 72 equivalences found

Inequivalences: 68 examples (~2300 lines of code)

- **up-to techniques off:** 515.7s, 65 inequivalences found
- **up-to techniques on:** 20.0s, 68 inequivalences found

Theoretically sound: Hobbit reported no false positives/negatives

Hobbit: https://github.com/LaifsV1/Hobbit

Technical Report: https://arxiv.org/abs/2105.02541

Related Work

SyTeCi: state of the art in contextual equivalence for stateful higher-order terms
Based on: logical relations, push-down systems and heuristics

- Proves equivalences that Hobbit cannot
  e.g. well-bracketed state
- Guarantees soundness only on the language subset it handles,
  e.g. no references in function bodies, no general recursion
- Generally slower than Hobbit and produces Horn clauses that are also harder to solve than Hobbit's SAT CNF formulas

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

- First-order contextual equivalence with global state is a simpler setting
  e.g. no callbacks or re-entrant calls
- Better at complexities arising from internal term transitions
  e.g. unbounded datatypes, recursion

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