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

Vasileios Koutavas[1], Yu-Yang Lin[1](✉), and Nikos Tzevelekos[2]

[1] Trinity College Dublin, Dublin, Ireland {Vasileios.Koutavas,linhouy}@tcd.ie
[2] Queen Mary University of London, London, UK nikos.tzevelekos@qmul.ac.uk

Abstract. We present a bounded equivalence verification technique for higher-order programs with local state. This technique combines fully abstract *symbolic environmental bisimulations* similar to symbolic game semantics, novel *up-to techniques*, and lightweight *state invariant annotations*. This yields an equivalence verification technique with no false positives or negatives. The technique is bounded-complete, in that all inequivalences are automatically detected given large enough bounds. Moreover, several hard equivalences are proved automatically or after being annotated with state invariants. We realise the technique in a tool prototype called Hobbit and benchmark it with an extensive set of new and existing examples. Hobbit can prove many classical equivalences including all Meyer and Sieber examples.

Keywords: Contextual equivalence · bounded model checking · symbolic bisimulation · up-to techniques · operational game semantics.

1 Introduction

Contextual equivalence is a relation over program expressions which guarantees that related expressions are interchangeable in any program context. It encompasses verification properties like safety and termination. It has attracted considerable attention from the semantics community (cf. the 2017 Alonzo Church Award), and has found its main applications in the verification of cryptographic protocols [4], compiler correctness [26] and regression verification [10,11,9,17].

In its full generality, contextual equivalence is hard as it requires reasoning about the behaviour of all program contexts, and becomes even more difficult in languages with higher-order features (e.g. callbacks) and local state. Advances in bisimulations [16,29,3], logical relations [1,13,15] and game semantics [18,25,8,20] have offered powerful theoretical techniques for hand-written proofs of contextual equivalence in higher-order languages with state. However, these advancements have yet to be fully integrated in verification tools for contextual equivalence in programming languages, especially in the case of bisimulation techniques. Existing tools [12,24,14] only tackle carefully delineated language fragments.

* This publication has emanated from research supported in part by a grant from Science Foundation Ireland under Grant number 13/RC/2094_2.

In this paper we aim to push the frontier further by proposing a bounded model checking technique for contextual equivalence for the entirety of a higher-order language with local state (Sec. 3). This technique, realised in a tool called Hobbit,[3] automatically detects inequivalent program expressions given sufficient bounds, and proves hard equivalences automatically or semi-automatically.

Our technique uses a labelled transition system (LTS) for open expressions in order to express equivalence as a bisimulation. The LTS is symbolic both for higher-order arguments (Sec. 4), similarly to symbolic game models [8,20] and derived proof techniques [3,15], and first-order ones (Sec. 6), adopting established techniques (e.g. [6]) and tools such as Z3 [23]. This enables the definition of a fully abstract *symbolic environmental bisimulation*, the bounded exploration of which is the task of the Hobbit tool. Full abstraction guarantees that our tool finds all inequivalences given sufficient bounds, and only reports true inequivalences. As is corroborated by our experiments, this makes Hobbit a practical inequivalence detector, similar to traditional bounded model checking [2] which has been proved an effective bug detection technique in industrial-scale C code [6,7,30].

However, while proficient in bug finding, bounded model checking can rarely prove the absence of errors, and in our setting prove an equivalence: a bound is usually reached before all—potentially infinite—program runs are explored. Inspired by hand-written equivalence proofs, we address this challenge by proposing two key technologies: new *bisimulation up-to techniques*, and lightweight user guidance in the form of *state invariant annotations*. Hence we increase significantly the number of equivalences proven by Hobbit, including for example all classical equivalences due to Meyer and Sieber [21].

Up-to techniques [28] are specific to bisimulation and concern the reduction of the size of bisimulation relations, oftentimes turning infinite transition systems into finite ones by focusing on a core part of the relation. Although extensively studied in the theory of bisimulation, up-to techniques have not been used in practice in an equivalence checker. We specifically propose three novel up-to techniques: *up to separation* and *up to re-entry* (Sec. 5), dealing with infinity in the LTS due to the higher-order nature of the language, and *up to state invariants* (Sec. 7), dealing with infinity due to state updates. Up to separation allows us to reduce the knowledge of the context the examined program expressions are running in, similar to a frame rule in separation logic. Up to re-entry removes the need of exploring unbounded nestings of higher-order function calls under specific conditions. Up to state invariants allows us to abstract parts of the state and make finite the number of explored configurations by introducing state invariant predicates in configurations.

State invariants are common in equivalence proofs of stateful programs, both in handwritten (e.g. [16]) and tool-based proofs. In the latter they are expressed manually in annotations (e.g. [9]) or automatically inferred (e.g. [14]). In Hobbit we follow the manual approach, leaving heuristics for automatic invariant inference for future work. An important feature of our annotations is the ability to express relations between the states of the two compared terms, enabled by the up to

[3] Higher Order Bounded BIsimulation Tool (Hobbit), https://github.com/LaifsV1/Hobbit.

state invariants technique. This leads to finite bisimulation transition systems in examples where concrete value semantics are infinite state.

The above technologies, combined with standard up-to techniques, transform Hobbit from a bounded checker into an equivalence prover able to reason about infinite behaviour in a finite manner in a range of examples, including classical example equivalences (e.g. all in [21]) and some that previous work on up-to techniques cannot algorithmically decide [3] (cf. Ex. 22). We have benchmarked Hobbit on examples from the literature and newly designed ones (Sec. 8). Due to the undecidable nature of contextual equivalence, up-to techniques are not exhaustive: no set of up-to techniques is guaranteed to finitise all examples. Indeed there are a number of examples where the bisimulation transition system is still infinite and Hobbit reaches the exploration bound. For instance, Hobbit is not able to prove examples with inner recursion and well-bracketing properties, which we leave to future work. Nevertheless, our approach provides a contextual equivalence tool for a higher-order language with state that can prove many equivalences and inequivalences which previous work could not handle due to syntactic restrictions and other limitations (Sec. 9).

Related work Our paper marries techniques from environmental bisimulations up-to [16,29,28,3] with the work on fully abstract game models for higher-order languages with state [18,8,20]. The closest to our technique is that of Biernacki et al. [3], which introduces up-to techniques for a similar symbolic LTS to ours, albeit with symbolic values restricted to higher-order types, resulting in infinite LTSs in examples such as Ex. 21, and with inequivalence decided outside the bisimulation by (non-)termination, precluding the use up-to techniques in examples such as Ex. 22. Close in spirit is the line of research on logical relations [1,13,15] which provides a powerful tool for hand-written proofs of contextual equivalence. Also related are the tools Hector [12] and Coneqct [24], and SyTeCi [14], based on game semantics and step-indexed logical relations respectively (cf. Sec. 9).

2 High-Level Intuitions

Contextual equivalence requires that two program expressions lead to the same observable result *in any program context* these may be fed in. Instead of working directly with this definition, we can translate programs into a semantic model that is *fully abstract*, reducing contextual equivalence to semantic equality.

The semantic model we use is that of Game Semantics [18]. We model programs as formal interactions between two *players*: a *Proponent* (corresponding to the program) and an *Opponent* (standing for any program context). Concretely, these interactions are sets of traces produced from a Labelled Transition System (LTS), the nodes and labels of which are called *configurations* and *moves* respectively. The LTS captures the interaction of the program with its environment, which is realised via function applications and returns: moves can be *questions* (i.e. function applications) or *answers* (returns), and belong to proponent or opponent. E.g. a program calling an external function will issue a proponent question, while the return of the external function will be an opponent answer. In the examples that follow, moves that correspond to the opponent shall be underlined.

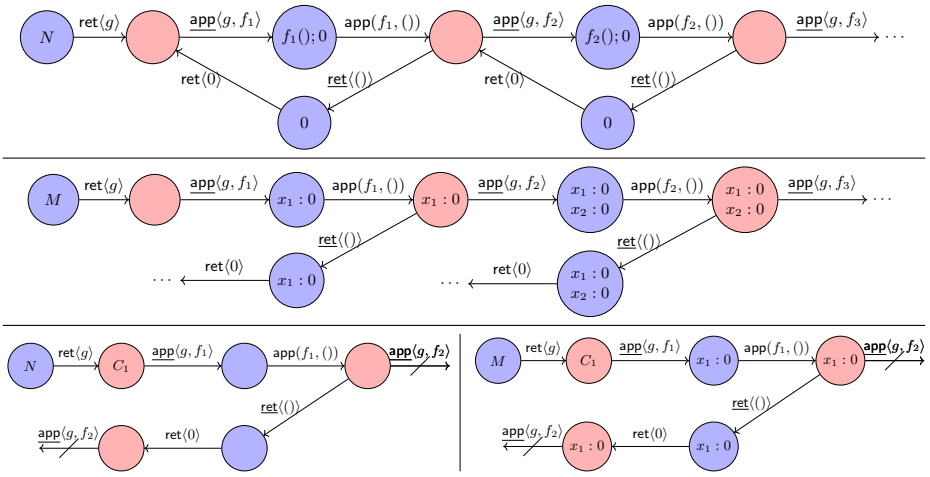


Fig. 1. Sample LTS's modelling expressions in Section 2.

Example 1. Consider the expression $N =$ `(fun f -> f (); 0)` of type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{int}$. Evaluating N leads to a function g being returned (i.e. g is $\lambda f.f();0$). When g is called with some input f_1, it will always return 0 but in the process it may call the external function f_1. The call to f_1 may immediately return or it may call g again (i.e. reenter), and so on. The LTS for N is as in Fig. 1 (top).

Given two expressions M, N, checking their equivalence will amount to checking bisimulation equivalence of their (generally infinite) LTS's. Our checking routine performs a bounded analysis that aims to either find a finite counterexample and thus prove inequivalence, or build a bisimulation relation that shows the equivalence of the expressions. The former case is easier as it is relatively rapid to explore a bisimulation graph up to a given depth. The latter one is harder, as the target bisimulation can be infinite. To tackle part of this infinity, we use three novel *up-to techniques* for environmental bisimulation.

Up-to techniques roughly assert that if a core set of configurations in the bisimulation graph explored can be proven to be part of a relation satisfying a definition that is more permissive than standard bisimulation, then a superset of configurations forms a proper bisimulation relation. This has the implication that a bounded analysis can be used to explore a finite part of the bisimulation graph to verify potentially infinitely many configurations. As there can be no complete set of up-to techniques, the pertaining question is how useful they are in practice. In the remainder of this section we present the first of our up-to techniques, called *up to separation*, via an example equivalence. The intuition behind this technique comes from Separation Logic and amounts to saying that functions that access separate regions of the state can be explored independently. As a corollary, a function that manipulates only its own local references may be explored independently of itself, i.e. it suffices to call it once.

$$
\begin{array}{lll}
\text{Loc:} & l, k & \text{Var:} x, y, z \qquad \text{Const:} c \\
\text{Ty:} & T ::= \mathsf{bool} \mid \mathsf{int} \mid \mathsf{unit} \mid T \to T \mid T_1 * \ldots * T_n & \\
\text{Ex:} & e, M, N ::= v \mid (\vec{e}) \mid op(\vec{e}) \mid e\, e \mid \mathsf{if}\, e\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{ref}\, l = v\, \mathsf{in}\, e \mid !l \mid l := e \mid \mathsf{let}(\vec{x}) = e\, \mathsf{in}\, e & \\
\text{Val:} & u, v ::= c \mid x \mid \mathsf{fix} f(x).e \mid (\vec{v}) & \\
\text{EC:} & E ::= [\cdot]_T \mid (\vec{v}, E, \vec{e}) \mid op(\vec{v}, E, \vec{e}) \mid E\, e \mid v\, E \mid l := E \mid \mathsf{if}\, E\, \mathsf{then}\, e\, \mathsf{else}\, e \mid \mathsf{let}(\vec{x}) = E\, \mathsf{in}\, e & \\
\text{Cxt:} & D ::= [\cdot]_{i,T} \mid e \mid (\vec{D}) \mid op(\vec{D}) \mid D\, D \mid l := D \mid \mathsf{if}\, D\, \mathsf{then}\, D\, \mathsf{else}\, D \mid \mathsf{fix} f(x).D & \\
 & \quad \mid \mathsf{ref}\, l = D\, \mathsf{in}\, D \mid \mathsf{let}(\vec{x}) = D\, \mathsf{in}\, D & \\
\text{St:} & s, t \in \mathsf{Loc} \xrightarrow{fin} \mathsf{Val} &
\end{array}
$$

$$
\begin{array}{lll}
\langle s\,;\, op(\vec{c})\rangle & \hookrightarrow \langle s\,;\, w\rangle & \text{if } op^{\mathsf{arith}}(\vec{c}) = w \\
\langle s\,;(\mathsf{fix} f(x).e)\, v\rangle & \hookrightarrow \langle s\,;\, e[v/x][\mathsf{fix} f(x).e/f]\rangle & \\
\langle s\,;\, \mathsf{let}(\vec{x}) = (\vec{v})\, \mathsf{in}\, e\rangle & \hookrightarrow \langle s\,;\, e[\vec{v}/\vec{x}]\rangle & \\
\langle s\,;\, \mathsf{ref}\, l = v\, \mathsf{in}\, e\rangle & \hookrightarrow \langle s[l \mapsto v]\,;\, e\rangle & \text{if } l \notin \mathsf{dom}(s) \\
\langle s\,;\, !l\rangle & \hookrightarrow \langle s\,;\, v\rangle & \text{if } s(l) = v \\
\langle s\,;\, l := v\rangle & \hookrightarrow \langle s[l \mapsto v]\,;()\rangle & \\
\langle s\,;\, \mathsf{if}\, c\, \mathsf{then}\, e_1\, \mathsf{else}\, e_2\rangle & \hookrightarrow \langle s\,;\, e_i\rangle & \text{if } (c, i) \in \{(\mathsf{tt}, 1), (\mathsf{ff}, 2)\} \\
\langle s\,;\, E[e]\rangle & \to \langle s'\,;\, E[e']\rangle & \text{if } \langle s\,;\, e\rangle \hookrightarrow \langle s'\,;\, e'\rangle
\end{array}
$$

Fig. 2. Syntax and reduction semantics of the language λ^{imp}.

Example 2. Consider M = `(fun f -> ref x = 0 in f (); !x)` and N from Ex. 1. The LTS corresponding to M and N are shown in Fig. 1 (middle and top). Regarding M, we can see that opponent is always allowed to reenter the proponent function g, which creates a new reference x_n each time. This makes each configuration unique, which prevents us from finding cycles and thus finitise the bisimulation graph. Moreover, both the LTS for M and N are infinite because of the stack discipline they need to adhere to when O issues reentrant calls.

With separation, however, we could prune the two LTS's as in Fig. 1 (bottom). We denote the configurations after the first opponent call as C_1. Any opponent call after C_1 leads to a configuration which differs from C_1 either by a state component that is not accessible anymore and can thus be separated, or by a stack component that can be similarly separated. Hence, the LTS's that we need to consider are finite and thus the expressions are proven equivalent.

3 Language and Semantics

We develop our technique for the language λ^{imp}, a simply typed lambda calculus with local state whose syntax and reduction semantics are shown in Fig. 2. Expressions (Ex) include the standard lambda expressions with recursive functions ($\mathsf{fix} f(x).e$), together with location creation ($\mathsf{ref}\, l = v\, \mathsf{in}\, e$), dereferencing ($!l$), and assignment ($l := e$), as well as standard base type constants (c) and operations ($op(\vec{e})$). Locations are mapped to values, including function values, in a store (St). We write $\cdot$ for the empty store and let $\mathsf{fl}(\chi)$ denote the set of free locations in χ.

The language λ^{imp} is simply-typed with typing judgements of the form $\Delta; \Sigma \vdash e : T$, where Δ is a type environment (omitted when empty), Σ a store typing and T a value type (Ty); Σ_s is the typing of store s. The rules of the type system are standard and omitted here. Values consist of boolean, integer, and unit constants,

functions and arbitrary length tuples of values. To keep the presentation of our technique simple we do not include reference types as value types, effectively keeping all locations local. Exchange of locations between expressions can be encoded using get and set functions. In Ex. 22 we show the encoding of a classic equivalence with location exchange between expressions and their context. Future work extensions to our technique to handle location types can be informed from previous work [18,14].

The reduction semantics is by small-step transitions between configurations containing a store and an expression, $\langle s\,;e\rangle \to \langle s'\,;e'\rangle$, defined using single-hole evaluation contexts (EC) over a base relation $\hookrightarrow$. Holes $[\cdot]_T$ are annotated with the type T of closed values they accept, which we may omit to lighten notation. Beta substitution of x with v in e is written as $e[v/x]$. We write $\langle s\,;e\rangle \Downarrow$ to denote $\langle s\,;e\rangle \to^* \langle t\,;v\rangle$ for some t, v. We write $\vec{\chi}$ to mean a syntactic sequence, and assume standard syntactic sugar from the lambda calculus. In our examples we assume an ML-like syntax and implementation of the type system, which is also the concrete syntax of Hobbit.

We consider environments $\Gamma \in \mathbb{N} \xrightarrow{\mathsf{fin}} \mathsf{Val}$ which map natural numbers to closed values. The concatenation of two such environments Γ_1 and Γ_2, written Γ_1, Γ_2 is defined when $\mathsf{dom}(\Gamma_1) \cap \mathsf{dom}(\Gamma_2) = \emptyset$. We write $({}^{i_1}v_1, \ldots, {}^{i_n}v_n)$ for a concrete environment mapping $i_1, \ldots, i_n$ to $v_1, \ldots, v_n$, respectively. When indices are unimportant we omit them and treat Γ environments as lists.

General contexts D contain multiple, non-uniquely indexed holes $[\cdot]_{i,T}$, where T is the type of value that can replace the hole. Notation $D[\Gamma]$ denotes the context D with each hole $[\cdot]_{i,T}$ replaced with $\Gamma(i)$, provided that $i \in \mathsf{dom}(\Gamma)$ and $\Sigma \vdash \Gamma(i) : T$, for some Σ. We omit hole types where possible and indices when all holes in D are annotated with the same i. In the latter case we write $D[v]$ instead of $D[({}^{i}v)]$ and allow to replace all holes of D with a closed expression e, written $D[e]$. We assume the Barendregt convention for locations, thus replacing context holes avoids location capture. Standard contextual equivalence [22] follows.

Definition 3 (Contextual Equivalence). *Expressions* $\vdash e_1 : T$ *and* $\vdash e_2 : T$ *are* contextually equivalent*, written as* $e_1 \equiv e_2$*, when for all contexts* D *such that* $\vdash D[e_1] : \mathsf{unit}$ *and* $\vdash D[e_2] : \mathsf{unit}$ *we have* $\langle \cdot\,; D[e_1]\rangle \Downarrow$ *iff* $\langle \cdot\,; D[e_2]\rangle \Downarrow$.

4 LTS with Symbolic Higher-Order Transitions

Our Labelled Transition System (LTS) has symbolic transitions for both higher-order and first-order transitions. For simplicity we first present our LTS with symbolic higher-order and concrete first-order transitions. We develop our theory and most up-to techniques on this simpler LTS. We then show its extension with symbolic first-order transitions and develop up to state invariants which relies on this extension. We extend the syntax with abstract function names α:

$$\mathsf{Val}: \quad u, v, w ::= c \mid \mathsf{fix} f(x).e \mid (\vec{v}) \mid \alpha_T$$

Abstract function names α_T are annotated with the type T of function they represent, omitted where possible; $\mathsf{an}(\chi)$ is the set of abstract names in χ.

$$\textsc{PropApp} : \langle A\,;\Gamma\,;K\,;s\,;E[\alpha\,v]\rangle \xrightarrow{\mathsf{app}(\alpha,D)} \langle A\,;\Gamma,\Gamma'\,;E[\cdot],K\,;s\,;\cdot\rangle \quad \text{if } (D,\Gamma') \in \mathsf{ulpatt}(v)$$

$$\textsc{PropRet} : \langle A\,;\Gamma\,;K\,;s\,;v\rangle \xrightarrow{\mathsf{ret}(D)} \langle A\,;\Gamma,\Gamma'\,;K\,;s\,;\cdot\rangle \quad \text{if } (D,\Gamma') \in \mathsf{ulpatt}(v)$$

$$\textsc{OpApp} : \langle A\,;\Gamma\,;K\,;s\,;\cdot\rangle \xrightarrow{\mathsf{app}(i,D[\vec{\alpha}])} \langle A \uplus \vec{\alpha}\,;\Gamma\,;K\,;s\,;e\rangle \quad \text{if } \Sigma_s \vdash \Gamma(i) : T \to T' \text{ and } (D,\vec{\alpha}) \in \mathsf{ulpatt}(T) \text{ and } \Gamma(i)\,D[\vec{\alpha}] \succ e$$

$$\textsc{OpRet} : \langle A\,;\Gamma\,;E[\cdot]_T,K\,;s\,;\cdot\rangle \xrightarrow{\mathsf{ret}(D[\vec{\alpha}])} \langle A \uplus \vec{\alpha}\,;\Gamma\,;K\,;s'\,;E[D[\vec{\alpha}]]\rangle \quad \text{if } (D,\vec{\alpha}) \in \mathsf{ulpatt}(T)$$

$$\textsc{Tau} : \langle A\,;\Gamma\,;K\,;s\,;e\rangle \xrightarrow{\tau} \langle A\,;\Gamma\,;K\,;s\,;e'\rangle \quad \text{if } \langle s\,;e\rangle \to \langle s'\,;e'\rangle$$

$$\textsc{Response} : C \xrightarrow{\eta} \langle\bot\rangle \quad \text{if } \eta \neq \downarrow$$

$$\textsc{Term} : \langle A\,;\Gamma\,;\cdot\,;s\,;\cdot\rangle \xrightarrow{\downarrow} \langle\bot\rangle$$

Fig. 3. The Labelled Transition System.

We define our LTS (shown in Fig. 3) by opponent and proponent call and return transitions, based on Game Semantics [18]. Proponent transitions are the moves of an expression interacting with its context. Opponent transitions are the moves of the context surrounding this expression. These transitions are over proponent and opponent configurations $\langle A\,;\Gamma\,;K\,;s\,;e\rangle$ and $\langle A\,;\Gamma\,;K\,;s\,;\cdot\rangle$, respectively. In these configurations:

- A is a set of abstract function names been used so far in the interaction;
- Γ is an environment indexing proponent functions known to opponent;[4]
- K is a stack of proponent continuations, created by nested proponent calls;
- s is the store containing proponent locations;
- e is the expression reduced in proponent configurations; $\hat{e}$ denotes e or $\cdot$.

In addition, we introduce a special configuration $\langle\bot\rangle$ which is used in order to represent expressions that cannot perform given transitions (cf. Remark 6). We let a *trace* be a sequence of app and ret moves (i.e. labels), as defined in Fig. 3.

For the LTS to provide a fully abstract model of the language, it is necessary that functions which are passed as arguments or return values from proponent to opponent be abstracted away, as the actual syntax of functions is not directly observable in λ^{imp}. This is achieved by deconstructing such values v to:

- an *ultimate pattern* D (cf. [19]), which is a context obtained from v by replacing each function in v with a distinct numbered hole; together with
- an environment Γ mapping indices of these holes to values, and $D[\Gamma] = v$.

We let $\mathsf{ulpatt}(v)$ contain all such pairs (D,Γ) for v; e.g.: $\mathsf{ulpatt}((\lambda x.e_1, 5)) = \{(([\cdot]_i, 5), [^i\lambda x.e_1]) \mid \text{for any } i\}$. We extend ulpatt to types through the use of symbolic function names: $\mathsf{ulpatt}(T)$ is the largest set of pairs (D,Γ) such that $\vdash D[\Gamma] : T$, where $rng(\Gamma) = \vec{\alpha}_{\vec{T}}$, and D does not contain functions.

[4] thus, Γ is encoding the environment of Environmental Bisimulations (e.g. [16])

In Fig. 3, proponent application and return transitions (PropApp, PropRet) use ultimate pattern matching for values and accumulate the functions generated by the proponent in the Γ environment of the configuration, leaving only their indices on the label of the transition itself. Opponent application and return transitions (OpApp, OpRet) use ultimate pattern matching for types to generate opponent-generated values which can only contain abstract functions. This eliminates the need for quantifying over all functions in opponent transitions but still includes infinite quantification over all base values. Symbolic first-order values in Sec. 6 will obviate the latter.

At opponent application the following preorder performs a beta reduction when opponent applies a concrete function. This technicality is needed for soundness.

Definition 4 ($\succ$). *For application $v\,u$ we write $v\,u \succ e$ to mean $e = \alpha\,u$, when $v = \alpha$; and $e = e'[u/x][\mathsf{fix} f(x).e'/f]$, when $v = \mathsf{fix} f(x).e'$.*

In our LTS, C ranges over configurations and η over transition labels; $\overset{\eta}{\Longrightarrow}$ means $\overset{\tau}{\longrightarrow}{}^*$, when $\eta = \tau$, and $\overset{\tau}{\Longrightarrow}\overset{\eta}{\longrightarrow}\overset{\tau}{\Longrightarrow}$ otherwise. Standard weak (bi-)simulation follows.

Definition 5 (Weak Bisimulation). *Binary relation $\mathcal{R}$ is a* weak simulation *when for all $C_1\ \mathcal{R}\ C_2$ and $C_1 \overset{\eta}{\longrightarrow} C_1'$, there exists C_2' such that $C_2 \overset{\eta}{\Longrightarrow} C_2'$ and $C_1'\ \mathcal{R}\ C_2'$. If $\mathcal{R}$, $\mathcal{R}^{-1}$ are weak simulations then $\mathcal{R}$ is a* weak bisimulation. *Similarity ($\underset{\approx}{\sqsubset}$) and bisimilarity ($\approx$) are the largest weak simulation and bisimulation, respectively.*

Remark 6. Any proponent configuration that cannot match a standard bisimulation transition challenge can trivially respond to the challenge by transitioning into $\langle\bot\rangle$ by the Response rule in Fig. 3. By the same rule, this configuration can trivially perform all transitions except a special termination transition, labelled with $\downarrow$. However, regular configurations that have no pending proponent calls ($K = \cdot$), can perform the special termination transition (Term rule), signalling the end of a *complete trace*, i.e. a completed computation. This mechanism allows us to encode complete trace equivalence, which coincides with contextual equivalence [18], as bisimulation equivalence. In a bisimulation proof, if a proponent configuration is unable to match a bisimulation transition with a regular transition, it can still transition to $\langle\bot\rangle$ where it can simulate every transition of the other expression, apart from $\overset{\downarrow}{\longrightarrow}$ leading to a complete trace.

Our mechanism for treating unmatched transitions has the benefit of enabling us to use the standard definition of bisimulation over our LTS. This is in contrast to previous work [3,15], where termination/non-termination needed to be proven independently or baked in the simulation conditions. More importantly, our approach allows us to use bisimulation up-to techniques even when one of the related configurations diverges, which is not possible in previous symbolic LTSs [18,15,3], and is necessary in examples such as Ex. 22.

Definition 7 (Bisimilar Expressions). *Expressions $\vdash e_1 : T$ and $\vdash e_2 : T$ are bisimilar, written $e_1 \approx e_2$, when $\langle \cdot ; \cdot ; \cdot ; \cdot ; e_1 \rangle \approx \langle \cdot ; \cdot ; \cdot ; \cdot ; e_2 \rangle$.*

Theorem 8 (Soundness and Completeness). *$e_1 \approx e_2$ iff $e_1 \equiv e_2$.*

As a final remark, the LTS presented in this section is finite state only for a small number of trivial equivalence examples. The following section addresses sources of infinity in the transition systems through bisimulation up-to techniques.

5 Up-to Techniques

We start by the definition of a sound up-to technique.

Definition 9 (Weak Bisimulation up to f). *$\mathcal{R}$ is a* weak simulation up to f *when for all $C_1\ \mathcal{R}\ C_2$ and $C_1 \xrightarrow{\eta} C_1'$, there is C_2' with $C_2 \stackrel{\eta}{\Longrightarrow} C_2'$ and $C_1'\ f(\mathcal{R})\ C_2'$. If $\mathcal{R}$, $\mathcal{R}^{-1}$ are weak simulations up to f then $\mathcal{R}$ is a* weak bisimulation up to f.

Definition 10 (Sound up-to technique). *A function f is a* sound up-to technique *when for any $\mathcal{R}$ which is a simulation up to f we have $R \subseteq (\lessapprox)$.*

HOBBIT employs the standard techniques: up to identity, up to garbage collection, up to beta reductions and up to name permutations. Here we present two novel up-to techniques: up to separation and up to reentry.

Up to Separation Our experience with HOBBIT has shown that one of the most effective up-to techniques for finitising bisimulation transition systems is the novel *up to separation* which we propose here. The intuition of this technique is that if different functions operate on disjoint parts of the store, they can be explored in disjoint parts of the bisimulation transition system. Taken to the extreme, a function that does not contain free locations can be applied only once in a bisimulation test as two copies of the function will not interfere with each other, even if they allocate new locations after application. To define up to separation we need to define a separating conjunction for configurations.

Definition 11 (Stack Interleaving). *Let K_1, K_2 be lists of evaluation contexts from* EC *(Fig. 2); we define the interleaving operation $K_1 \#_{\vec{k}} K_2$ inductively, and write $K_1 \# K_2$ to mean $K_1 \#_{\vec{k}} K_2$ for unspecified $\vec{k}$. We let $\cdot \#_{\cdot} \cdot = \cdot$ and:*

$$E_1, K_1 \#_{(1,\vec{k})} K_2 = E_1, (K_1 \#_{\vec{k}} K_2) \qquad K_1 \#_{(2,\vec{k})} E_2, K_2 = E_2, (K_1 \#_{\vec{k}} K_2)\,.$$

Definition 12 (Separating Conjuction). *Let $C_1 = \langle A_1 ; \Gamma_1 ; K_1 ; s_1 ; \hat{e}_1 \rangle$ and $C_2 = \langle A_2 ; \Gamma_2 ; K_2 ; s_2 ; \hat{e}_2 \rangle$ be well-formed configurations. We define:*

- $C_1 \oplus^1_{\vec{k}} C_2 \stackrel{def}{=} \langle A_1 \cup A_2 ; \Gamma_1, \Gamma_2 ; K_1 \#_{\vec{k}} K_2 ; s_1, s_2 ; \hat{e}_1 \rangle$ *when* $\hat{e}_2 = \cdot$
- $C_1 \oplus^2_{\vec{k}} C_2 \stackrel{def}{=} \langle A_1 \cup A_2 ; \Gamma_1, \Gamma_2 ; K_1 \#_{\vec{k}} K_2 ; s_1, s_2 ; \hat{e}_2 \rangle$ *when* $\hat{e}_1 = \cdot$

provided $\mathsf{dom}(s_1) \cap \mathsf{dom}(s_2) = \emptyset$. *We let $C_1 \oplus C_2$ denote $\exists i, \vec{k}.\ C_1 \oplus^i_{\vec{k}} C_2$.*

The function sep provides the up to separation technique; it is defined as:

$$\textsc{UpTo}\oplus\ \frac{C_1\ \mathcal{R}\ C_2 \qquad C_3\ \mathcal{R}\ C_4}{C_1 \oplus^i_{\vec{k}} C_3\ \mathsf{sep}(\mathcal{R})\ C_2 \oplus^i_{\vec{k}} C_4} \qquad \textsc{UpTo}\oplus\bot_L\ \frac{C_1\ \mathcal{R}\ \langle\bot\rangle \qquad C_3\ \mathcal{R}\ C_4}{C_1 \oplus C_3\ \mathsf{sep}(\mathcal{R})\ \langle\bot\rangle} \qquad \textsc{UpTo}\oplus\bot_R\ \frac{C_1\ \mathcal{R}\ C_2 \qquad C_3\ \mathcal{R}\ \langle\bot\rangle}{C_1 \oplus C_3\ \mathsf{sep}(\mathcal{R})\ \langle\bot\rangle}$$

Soundness follows by extending [28,27] with a weaker, sufficient proof obligation.

Lemma 13. *Function* sep *is a sound up-to technique.*

Many example equivalences have a finite transition system when using up to separation in conjunction with the simple techniques of the preceding section.

Example 14. The following is a classic example equivalence from Meyer and Sieber [21]. The following expressions are equivalent at type $(\mathsf{unit} \to \mathsf{unit}) \to \mathsf{unit}$.

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

For both functions, after initial application of the function by the opponent, the proponent calls `f`, growing the stack K in the two configurations. At that point the opponent can apply the same functions again. The LTS of both M and N is thus infinite because K can grow indefinitely, and so is a bisimulation proving this equivalence. It is additionally infinite because the opponent can keep applying the initial function applications even after these return. However, if we apply the up-to separation technique immediately after the first opponent application, the Γ environments become empty, and thus no second application of the same functions can happen. The LTS thus becomes trivially small. Note that no other up to technique is needed here. HOBBIT applies up-to separation after every opponent application transition and explores the configuration containing the application expression and the smallest possible Γ; this does not lead to false-negative (or false-positive) results.

Example 15. This example is due to Bohr and Birkedal [5] and includes a non-synchronised divergence.

$M =$
```
fun f ->
  ref l1 = false in ref l2 = false in
  f (fun () -> if !l1 then _bot_ else l2 := true);
  if !l2 then _bot_ else l1 := true
```

$N =$ `fun f -> f (fun () -> _bot_)`

Note that `_bot_` is a diverging computation. This is a hard example to prove using environmental bisimulation even with up to techniques, requiring quantification over contexts within the proof. However, with up-to separation after the opponent applies the initial functions, the Γ environments are emptied, thus leaving only one application of M and N that needs to be explored by the bisimulation. Applications of the inner function provided as argument to `f` only leads to a small number of reachable configurations. HOBBIT can indeed prove this equivalence.

Up to Proponent Function Re-entry The higher-order nature of λ^{imp} and its LTS allows infinite nesting of opponent and proponent calls. Although up to separation avoids those in a number of examples, here we present a second novel up-to technique, which we call *up to proponent function re-entry* (or simply, up to re-entry). This technique has connections to the induction hypothesis in the definition of environmental bisimulations in [16]. However up to re-entry is specifically aimed at avoiding nested calls to proponent functions, and it is designed to work with our symbolic LTS. In combination with other techniques this eliminates the need to consider configurations with unbounded stacks K in many classical equivalences, including those in [21].

$$\textsc{UpToReentry}\quad \frac{\begin{array}{l} C_1 = \langle A\,;\Gamma_1\,;K_1\,;s_1\,;\cdot\rangle\ \mathcal{R}\ \langle A\,;\Gamma_2\,;K_2\,;s_2\,;\cdot\rangle = C_2 \\ \forall \vec{\eta}, C, A', \Gamma_1', \Gamma_2', s_1', s_2'.\ \big[(\underline{\mathsf{app}}(i,\_) \notin \{\vec{\eta}\} \text{ and} \\ \qquad \langle A\,;\Gamma_1\,;\cdot\,;s_1\,;\cdot\rangle \xrightarrow{\underline{\mathsf{app}}(i,C)} \xrightarrow{\vec{\eta}} \asymp \langle A'\,;\Gamma_1'\,;\cdot\,;s_1'\,;\cdot\rangle \text{ and} \\ \qquad \langle A\,;\Gamma_2\,;\cdot\,;s_2\,;\cdot\rangle \xrightarrow{\underline{\mathsf{app}}(i,C)} \xrightarrow{\vec{\eta}} \asymp \langle A'\,;\Gamma_2'\,;\cdot\,;s_2'\,;\cdot\rangle \\ \qquad \text{implies } \Gamma_1' = \Gamma_1 \text{ and } \Gamma_2' = \Gamma_2 \text{ and } s_1 = s_1' \text{ and } s_2 = s_2'\big] \\ C_1 \xrightarrow{\underline{\mathsf{app}}(i,C)} \xrightarrow{\vec{\eta}'} \xrightarrow{\underline{\mathsf{app}}(i,C')} \langle A'\,;\Gamma_1\,;K_1'\,,K_1\,;s_1\,;e_1'\rangle \\ C_2 \xrightarrow{\underline{\mathsf{app}}(i,C)} \xrightarrow{\vec{\eta}'} \xrightarrow{\underline{\mathsf{app}}(i,C')} \langle A'\,;\Gamma_2\,;K_2'\,,K_2\,;s_2\,;e_2'\rangle \end{array}}{\langle A'\,;\Gamma_1\,;K_1'\,,K_1\,;s_1\,;e_1'\rangle\ \mathsf{reent}(\mathcal{R})\ \langle A'\,;\Gamma_2\,;K_2'\,,K_2\,;s_2\,;e_2'\rangle}$$

Fig. 4. Up to Proponent Function Re-entry (omitting rules for $\bot$-configurations).

Up to re-entry is realised by function reent in Fig. 4. The intuition of this up-to technique is that if the application of related functions at i in the Γ environments has no potential to change the local stores (up to garbage collection, encoded by $(\asymp)$) or increase the Γ environments, then there are no additional observations to be made by nested calls to the i-functions, thus configurations reached by such nested calls are added to the relation by this up-to technique. Soundness follows similarly to up-to separation.

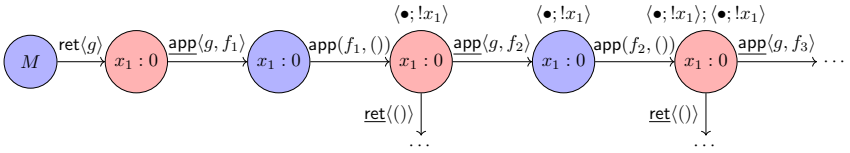
In Hobbit we require the user to flag the functions to be considered for the up to re-entry technique. This annotation is later combined with state invariant annotations, as they are often used together. Inequivalences found while using the up to re-entry and state invariant annotations could be false-negatives due to incorrect user annotations. Hobbit ensures that no such false-negatives are reported by re-running discovered inequivalences with these two techniques off.

Below is an example where the state invariant needed is trivial and up to separation together with up to re-entry are sufficient to prove the equivalence.

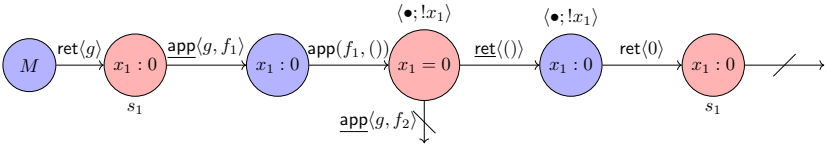
Example 16.

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

This is like Ex. 2 except the reference in M is created outside of the function body. The LTS for this is as follows. Labels $\langle \bullet; !x_1\rangle$ are continuations.



Again, the opponent is allowed to reenter g as before. With up-to reentry, however, the opponent skips nested calls to g as these do not modify the state.



N mirrors the above LTS without the x_1 reference and with continuation $\langle \bullet; 0 \rangle$.

6 Symbolic First-Order Transitions

We extend λ^{imp} constants (Const) with a countable set of symbolic constants ranged over by κ. We define symbolic environments $\sigma ::= \cdot \mid (\kappa \frown e), \sigma$, where $\frown$ is either $=$ or $\neq$, and e is an arithmetic expression over constants, and interpret them as conjunctions of (in-)equalities, with the empty set interpreted as $\top$.

Definition 17 (Satisfiability). *Symbolic environment σ is* satisfiable *if there exists an* assignment *δ, mapping the symbolic constants of σ to actual constants, such that $\delta\sigma$ is a tautology; we then write $\delta \vDash \sigma$.*

We extend reduction configurations with a symbolic environment σ, written as $\sigma \vdash \langle s \,;\, e \rangle$. These constants are implicitly annotated with their type. We modify the reduction semantics from Fig. 2 to consider symbolic constants:

$$\begin{array}{lll}
\sigma \vdash \langle s \,;\, op(\vec{c}) \rangle & \hookrightarrow \sigma \wedge (\kappa = op(\vec{c})) \vdash \langle s \,;\, \kappa \rangle & \text{if } \kappa \text{ fresh} \\
\sigma \vdash \langle s \,;\, \mathsf{if}\ \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rangle & \hookrightarrow \sigma \wedge (\kappa = \mathtt{tt}) \vdash \langle s \,;\, e_1 \rangle & \text{if } \sigma \wedge (\kappa = \mathtt{tt}) \text{ is sat.} \\
\sigma \vdash \langle s \,;\, \mathsf{if}\ \kappa\ \mathsf{then}\ e_1\ \mathsf{else}\ e_2 \rangle & \hookrightarrow \sigma \wedge (\kappa = \mathtt{ff}) \vdash \langle s \,;\, e_2 \rangle & \text{if } \sigma \wedge (\kappa = \mathtt{ff}) \text{ is sat.}
\end{array}$$

All other reduction semantics rules carry the σ. The LTS from Sec. 4 is modified to operate over configurations of the form $\sigma \vdash C$ or $\cdot \vdash \langle \bot \rangle$. We let $\widetilde{C}$ range over both forms of configurations. All LTS rules for proponent transitions simply carry the σ; rule Tau may increase σ due to the inner reduction. Opponent transitions generate fresh symbolic constants, instead of actual constants: labels $\underline{\mathsf{app}}(i, D[\vec{\alpha}])$ and $\underline{\mathsf{ret}}(D[\vec{\alpha}])$ in rules OpApp and OpRet of Fig. 3, respectively, contain D with symbolic, instead of concrete constants. We adapt (bi-)simulation as follows.

Definition 18. *Binary relation $\mathcal{R}$ on symbolic configurations is a* weak simulation *when for all $\widetilde{C}_1\ \mathcal{R}\ \widetilde{C}_2$ and $\widetilde{C}_1 \xrightarrow{\eta_1} \widetilde{C}_1'$, $\exists \widetilde{C}_2'$ such that $\widetilde{C}_2 \overset{\eta_2}{\Longrightarrow} \widetilde{C}_2'$ and*

$$\widetilde{C}_1'\ \mathcal{R}\ \widetilde{C}_2' \qquad (\widetilde{C}_1'.\sigma, \widetilde{C}_2'.\sigma) \text{ is sat.} \qquad \forall \delta.\ \delta \vDash (\widetilde{C}_1'.\sigma, \widetilde{C}_2'.\sigma) \implies \delta\eta_1 = \delta\eta_2$$

Lemma 19. $(\sigma_1 \vdash C_1) \mathrel{\sqsubseteq\!\!\!\approx} (\sigma_2 \vdash C_2)$ *iff for all $\delta \vDash \sigma_1, \sigma_2$ we have $\delta C_1 \mathrel{\sqsubseteq\!\!\!\approx} \delta C_2$.*

Corollary 20 (Soundness, Completeness). $(\cdot \vdash C_1) \mathrel{\sqsubseteq\!\!\!\approx} (\cdot \vdash C_2)$ *iff* $C_1 \mathrel{\sqsubseteq\!\!\!\approx} C_2$.

The up-to techniques we have developed in previous sections apply unmodified to the extended LTS as the techniques do not involve symbolic constants, with the exception of up to beta which requires adapting the definition of a beta move to consider all possible δ. The introduction of symbolic first-order transitions allows us to prove many interesting first-order examples, such as the equivalence of

bubble sort and insertion sort, an example borrowed from Hector [12] (omitted here, see the Hobbit distribution). Below is a simpler example showing the equivalence of two integer swap functions which, by leveraging Z3 [23], Hobbit is able to prove.

Example 21.

$M =$
```
let swap xy =
  let (x,y) = xy
  in (y, x)
in swap
```

$N =$
```
fun xy -> let (x,y) = xy in
  ref x = x in ref y = y in
  x := !x - !y; y := !x + !y;
  x := !y - !x; (!x, !y)
```

7 Up to State Invariants

The addition of symbolic constants into λ^{imp} and the LTS not only allows us to consider all possible opponent-generated constants simultaneously in a symbolic execution of proponent expressions, but also allows us to define an additional powerful up-to technique: *up to state invariants*. We define this technique in two parts: *up to abstraction* and *up to tautology* realised by abs and taut.[5]

$$\textsc{UpTo}\mathsf{abs}\ \frac{(\sigma_1 \vdash C_1)\ \mathcal{R}\ (\sigma_2 \vdash C_2)}{(\sigma_1 \vdash C_1)[\vec{c}/\vec{\kappa}]\ \mathsf{abs}(\mathcal{R})\ (\sigma_2 \vdash C_2)[\vec{c}/\vec{\kappa}]} \qquad \textsc{UpTo}\mathsf{taut}\ \frac{(\sigma_1, \sigma_1' \vdash C_1)\ \mathcal{R}\ (\sigma_2, \sigma_2' \vdash C_2) \quad \sigma_1, \sigma_2, \sigma_1', \sigma_2' \text{ is sat.} \quad \sigma_1, \sigma_2 \wedge \neg(\sigma_1', \sigma_2') \text{ is not sat.}}{(\sigma_1 \vdash C_1)\ \mathsf{taut}(\mathcal{R})\ (\sigma_2 \vdash C_2)}$$

The first function abs allows us to derive the equivalence of configurations by abstracting constants with fresh symbolic constants (of the same type) and instead prove equivalent the more abstract configurations. The second function taut allows us to introduce tautologies into the symbolic environments. These are predicates which are valid; i.e., they hold for all instantiations of the abstract variables. Combining the two functions we can introduce a tautology $I(\vec{c})$ into the symbolic environments, and then abstract constants $\vec{c}$ from the predicate but also from the configurations with symbolic ones, obtaining $I(\vec{\kappa})$, which encodes an invariant that always holds.

Currently in Hobbit, up to abstraction and tautology are combined and applied in a principled way. Functions can be annotated with the following syntax:

F `= fun x {`$\vec{\kappa}$ `|` l_1 `as` $C_1[\vec{\kappa}]$`, ..,` l_n `as` $C_n[\vec{\kappa}]$ `|` ϕ`} -> e`

The annotation instructs Hobbit to use the two techniques when opponent applies related functions where at least one of them has such an annotation. If both functions contain annotations, then they are combined and the same $\vec{\kappa}$ are used in both annotations. The techniques are used again when proponent returns from the functions, and proponent calls opponent from within the functions.[6] As discussed in Sec. 5, the same annotation enables up to reentry in Hobbit.

When Hobbit uses the above two up-to techniques it *1)* pattern-matches the values currently in each location l_i with the value context C_i where fresh

[5] Hobbit also implements an *up to σ-normalisation and garbage collection* technique.
[6] Finer-grain control of application of these up-to techniques is left to future work.

symbolic constants $\vec{\kappa}$ are in its holes, obtaining a substitution $[\vec{c}/\vec{\kappa}]$; *2)* the up to tautology technique is applied for the formula $\phi[\vec{c}/\vec{\kappa}]$; and *3)* the up to abstraction technique is applied by replacing $\phi[\vec{c}/\vec{\kappa}]$ in the symbolic environment with ϕ, and the contents of locations l_i with $C_i[\vec{\kappa}]$.

Example 22. Following is an example by Meyer and Sieber [21] featuring location passing, adapted to λ^{imp} where locations are local.

```
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_
```

In this example we simulate general references as a pair of read-write functions. Function `loc_eq` implements a standard location equality test. The two higher-order expressions are equivalent because the opponent can only increase the contents of `x` through the function `almostadd_2`. As the number of times the opponent can call this function is unbounded, the LTS is infinite. However, the annotation of function `almostadd_2` applies the up to state invariants technique when the function is called (and, less crucially, when it returns), replacing the concrete value of `x` with a symbolic integer constant w satisfying the invariant `w mod 2 == 0`. This makes the LTS finite, up to permutations of symbolic constants. Moreover, up to separation removes the outer functions from the Γ environments, thus preventing re-entrant calls to these functions. Note the up to techniques are applied even though one of the configurations is diverging (`_bot_`). This would not be possible with the LTS and bisimulation of [3].

8 Implementation and Evaluation

We implemented the LTS and up-to techniques for λ^{imp} in a tool prototype called Hobbit, which we ran on a test-suite of 105 equivalences and 68 inequivalences—3338 and 2263 lines of code for equivalences and inequivalences respectively.

Hobbit is bounded in the total number of function calls it explores per path. We ran Hobbit with a default bound of 6 calls except where a larger bound was found to prove or disprove equivalence—46 examples required a larger bound, and the largest bound used was 348. To illustrate the impact of up-to techniques, we checked all files (pairs of expressions to be checked for equivalence) in five configurations: default (all up-to techniques on), up to separation off, annotations (up to state invariants and re-entry) off, up to re-entry off, and everything off. The tool stops at the first trace that disproves equivalence, after enumerating all traces up to the bound, or after timing out at 150 seconds. Time taken and exit status (equivalent, inequivalent, inconclusive) were recorded for each file; an overview of the experiment can be seen in the following table. All experiments ran on an Ubuntu 18.04 machine with 32GB RAM, Intel Core i7 1.90GHz CPU, with intermediate calls to Z3 4.8.10 to prune invalid internal symbolic branching

and decide symbolic bisimulation conditions. All constraints passed to Z3 are of propositional satisfiability in conjunctive normal form (CNF).

	default	**sep. off**	**annot. off**	**ree. off**	**all off**
eq.	72 | 0 [5.6s]	32 | 0 [1622.9s]	47 | 0 [178.3s]	57 | 0 [177.6s]	3 | 0 [2098.5s]
ineq.	0 | 68 [20.0s]	0 | 66 [312.8s]	0 | 68 [19.6s]	0 | 68 [20.1s]	0 | 65 [515.7s]
$a \mid b$ [c] for a (out of 105) equivalences and b (out of 68) inequivalences reported taking c seconds in total.					

We can observe that Hobbit was sound and bounded-complete for our examples; no false reports and all inequivalences were identified. Up-to techniques also had a significant impact on proving equivalence. With all techniques on, it proved 68.6% of our equivalences; a dramatic improvement over 2.9% proven with none on. The most significant technique was up-to separation—necessary for 55.6% of equivalences proven and reducing time taken by 99.99%—which was useful when functions could be independently explored by the context. Following was annotations—necessary for 34.7% of equivalences and decreasing time by 96.9%—and up-to re-entry—20.8% of files and decreased time by 96.8%. Although the latter two required manual annotation, they enabled equivalences where our language was able to capture the proof conditions. Note that, since turning off invariant annotations also turns off re-entry, only 10 files needed up-to re-entry on top of invariant annotations. In contrast, inequivalences did not benefit as much. This was expected as without up-to techniques Hobbit is still based on bounded model checking, which is theoretically sound and complete for inequivalences, and finds the shortest counterexample traces using breadth-first search. Nonetheless, with up-to techniques turned off, inequivalences were discovered in 515.7s (vs. 20s with techniques on) and three files timed out, due to the techniques reducing the size and branching factor of configurations. This suggests that the reduction in state space is still relevant when searching for counterexamples.

9 Comparison with Existing Tools

There are two main classes of tools for contextual equivalence checking. The first one includes semantics-driven tools that tackle higher-order languages with state like ours. In this class belong game-based tools Hector [12] and Coneqct [24], which can only address carefully crafted fragments of the language, delineated by type restrictions and bounded data types. The most advanced tool in this class is SyTeCi [14], which is based on logical relations and removes a good part of the language restrictions needed in the previous tools. The second class concerns tools that focus on first-order languages, typically variants of C, with main tools including Rêve [9], SymDiff [17] and RVT [11]. These are highly optimised for handling *internal loops*, a problem orthogonal to handling the interactions between higher-order functions and their environment, addressed by Hobbit and related tools. We believe the techniques used in these tools may be useful when adapted to Hobbit, which we leave for future work.

In the higher-order contextual equivalence setting, the most relevant tool to compare with Hobbit is SyTeCi. This is because SyTeCi supersedes previous tools by proving examples with fewer syntactical limitations. We ran the tools on

examples from both SyTeCi's and our own benchmarks—7 and 15 equivalences, and 2 and 7 inequivalences from SyTeCi and Hobbit respectively—with a timeout of 150s and using Z3. Unfortunately, due to differences in parsing and SyTeCi's syntactical restrictions, the input languages were not entirely compatible and only few manually translated programs were chosen.

	SyTeCi	**Hobbit**
SyTeCi eq. examples	3 | 0 | 4 (0.03s)	1 | 0 | 6 (<0.01s)
Hobbit eq. examples	8 | 0 | 7 (0.4s)	15 | 0 | 0 (<0.01s)
SyTeCi ineq. examples	0 | 2 | 0 (0.06s)	0 | 2 | 0 (0.02s)
Hobbit ineq. examples	2 | 3 | 2 (0.52s)	0 | 7 | 0 (0.45s)
$a \mid b \mid c$ (d) for a eq's, b ineq's and c inconclusive's reported taking d sec in total		

We were unable to translate many of our examples because of restrictions in the input syntax supported by SyTeCi. Some of these restrictions were inessential (e.g. absence of tuples) while others were substantial: the tool does not support programs where references are allocated both inside and outside functions (e.g. Ex. 15), or with non-synchroniseable recursive calls. Moreover, SyTeCi relies on Constrained Horn Clause satisfiability which is undecidable. In our testing SyTeCi sometimes timed out on examples; in private correspondence with its creator this was attributed to Z3's ability to solve Constrained Horn Clauses. Finally, SyTeCi was sound for equivalences, but not always for inequivalences as can be seen in the table above; the reason is unclear and may be due to bugs. On the other hand, SyTeCi was able to solve equivalences we are not able to handle; e.g. synchronisable recursive calls and examples with well-bracketing properties.

10 Conclusion

Our experience with Hobbit suggests that our technique provides a significant contribution to verification of contextual equivalence. In the higher-order case, Hobbit does not impose language restrictions as present in other tools. Our tool is able to solve several examples that can not be solved by SyTeCi, which is the most advanced tool in this family. In the first-order case, the problem of contextual equivalence differs significantly as the interactions that a first-order expression can have with its context are limited; e.g. equivalence analyses do not need to consider callbacks or re-entrant calls. Moreover, the distinction between global and local state is only meaningful in higher-order languages where a program phrase can invoke different calls of the same function, each with its own state. Therefore, tools for first-order languages focus on what in our setting are internal transitions and the complexities arising from e.g. unbounded datatypes and recursion, whereas we focus on external interactions with the context.

As for limitations, Hobbit does not handle synchronised internal recursion and well-bracketed state, which SyTeCi can often solve. More generally, Hobbit is not optimised for internal recursion as first-order tools are. In this work we have also disallowed reference types in λ^{imp} to simplify the technical development; location exchange is encoded via function exchange (cf. Ex. 22). We intend to address these limitations in future work and explore applications of Hobbit to real-world examples.

References

1. Ahmed, A., Dreyer, D., Rossberg, A.: State-dependent representation independence. In: POPL. Association for Computing Machinery (2009)
2. Biere, A., Cimatti, A., Clarke, E., Zhu, Y.: Symbolic model checking without BDDs. In: TACAS. Springer Berlin Heidelberg (1999)
3. Biernacki, D., Lenglet, S., Polesiuk, P.: A complete normal-form bisimilarity for state. In: FOSSACS 2019, ETAPS 2019, Prague, Czech Republic. Springer (2019)
4. Blanchet, B.: A computationally sound mechanized prover for security protocols. In: IEEE Symposium on Security and Privacy (2006)
5. Bohr, N., Birkedal, L.: Relational reasoning for recursive types and references. In: Kobayashi, N. (ed.) APLAS. LNCS, vol. 4279, pp. 79–96. Springer (2006)
6. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ANSI-C programs. In: TACAS. Springer Berlin Heidelberg (2004)
7. Cordeiro, L., Kroening, D., Schrammel, P.: JBMC: Bounded model checking for Java Bytecode. In: TACAS. Springer (2019)
8. Dimovski, A.: Program verification using symbolic game semantics. TCS **560** (2014)
9. Felsing, D., Grebing, S., Klebanov, V., Rümmer, P., Ulbrich, M.: Automating regression verification. In: ACM/IEEE ASE '14. ACM (2014)
10. Godlin, B., Strichman, O.: Inference rules for proving the equivalence of recursive procedures. Acta Informatica **45**(6) (2008)
11. Godlin, B., Strichman, O.: Regression verification. In: DAC. ACM (2009)
12. Hopkins, D., Murawski, A.S., Ong, C.L.: Hector: An equivalence checker for a higher-order fragment of ML. In: CAV. LNCS, Springer (2012)
13. Hur, C.K., Dreyer, D., Neis, G., Vafeiadis, V.: The marriage of bisimulations and Kripke logical relations. SIGPLAN Not. (2012)
14. Jaber, G.: SyTeCi: Automating contextual equivalence for higher-order programs with references. Proc. ACM Program. Lang. (POPL) (2020)
15. Jaber, G., Tabareau, N.: Kripke open bisimulation - A marriage of game semantics and operational techniques. In: APLAS. Springer (2015)
16. Koutavas, V., Wand, M.: Small bisimulations for reasoning about higher-order imperative programs. In: POPL. ACM (2006)
17. Lahiri, S.K., Hawblitzel, C., Kawaguchi, M., Rebêlo, H.: SYMDIFF: A language-agnostic semantic diff tool for imperative programs. In: CAV. Springer (2012)
18. Laird, J.: A fully abstract trace semantics for general references. In: ICALP, Wroclaw, Poland. LNCS, Springer (2007)
19. Lassen, S.B., Levy, P.B.: Typed normal form bisimulation. In: Computer Science Logic. Springer Berlin Heidelberg (2007)
20. Lin, Y., Tzevelekos, N.: Symbolic execution game semantics. In: FSCD. Schloss Dagstuhl - Leibniz-Zentrum für Informatik (2020)
21. Meyer, A.R., Sieber, K.: Towards fully abstract semantics for local variables. In: POPL. Association for Computing Machinery (1988)
22. Morris, Jr., J.H.: Lambda Calculus Models of Programming Languages. Ph.D. thesis, MIT, Cambridge, MA (1968)
23. de Moura, L., Bjørner, N.: Z3: An efficient smt solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. Springer Berlin Heidelberg, Berlin, Heidelberg (2008)
24. Murawski, A.S., Ramsay, S.J., Tzevelekos, N.: A contextual equivalence checker for IMJ*. In: ATVA. Springer (2015)
25. Murawski, A.S., Tzevelekos, N.: Nominal game semantics. FTPL **2**(4) (2016)

26. Patterson, D., Ahmed, A.: The next 700 compiler correctness theorems (functional pearl). Proc. ACM Program. Lang. (ICFP) (2019)
27. Pous, D.: Coinduction all the way up. In: ACM/IEEE LICS. ACM (2016)
28. Pous, D., Sangiorgi, D.: Enhancements of the bisimulation proof method. In: Advanced Topics in Bisimulation and Coinduction. CUP (2012)
29. Sangiorgi, D., Kobayashi, N., Sumii, E.: Environmental bisimulations for higher-order languages. In: LICS. IEEE Computer Society (2007)
30. Schrammel, P., Kroening, D., Brain, M., Martins, R., Teige, T., Bienmüller, T.: Successful use of incremental BMC in the automotive industry. In: FMICS (2015)