SyTeCi: Automating Contextual Equivalence for Higher-Order Programs with References

Guilhem Jaber

SyTeCi: Automating Contextual Equivalence for Higher-Order Programs with References

GUILHEM JABER, Université de Nantes, LS2N CNRS, Inria, France

We propose a framework to study contextual equivalence of programs written in a call-by-value functional language with local integer references. It reduces the problem of contextual equivalence to the problem of non-reachability in a transition system of memory configurations. This reduction is complete for recursion-free programs. Restricting to programs that do not allocate references inside the body of functions, we encode this non-reachability problem as a set of constrained Horn clause that can then be checked for satisfiability automatically. Restricting furthermore to a language with finite data-types, we also get a new decidability result for contextual equivalence at any type.

CCS Concepts: • **Theory of computation → Functional constructs**; **Program semantics**; **Program reasoning**.

Additional Key Words and Phrases: Contextual Equivalence, Higher-Order Stateful Programs, Operational Game Semantics

ACM Reference Format:
Guilhem Jaber. 2020. SyTeCi: Automating Contextual Equivalence for Higher-Order Programs with References. *Proc. ACM Program. Lang.* 4, POPL, Article 59 (January 2020), 28 pages. https://doi.org/10.1145/3371127

1 INTRODUCTION

Proving that two programs are equivalent is crucial to ensure that an optimization of a program is correct, or to ensure that a modification does not introduce any regression. Many recent works have introduced techniques to check automatically if two programs are equivalent [Ciobâcă et al. 2016; Felsing et al. 2014; Godlin and Strichman 2009; Lahiri et al. 2012]. Considering a notion of input/output equivalence, they use standard techniques from static analysis to check it. An important idea in this setting is to use similarities of the two programs. This is particularly useful when reasoning on recursive programs, where we can perform circular reasoning by supposing that the recursive calls are related to prove that the body of the programs are related.

However, the notions of equivalence considered in these works are not suitable for open or higher-order programs that manipulate private memory cells, a.k.a. references or local states. In this paper, we study such a language: a call-by-value higher-order language with references, i.e. a fragment of ML. For this language, ***contextual (a.k.a. observational) equivalence*** is accepted as the right notion of equivalence of programs, being the gold standard in the programming language semantics community.

It consists in seeing programs as black-boxes, checking that they interact in an observationally equal way w.r.t. any context (i.e. a program with a hole). This means that no context can distinguish them, not being able to observe any difference between them. Due to the universal quantification over any contexts in its definition, it can be hard to prove that two programs are contextually

Author's address: Guilhem Jaber, Université de Nantes, LS2N CNRS, Inria, France, guilhem.jaber@univ-nantes.fr.

2475-1421/2020/1-ART59
https://doi.org/10.1145/3371127

equivalent. Indeed, for a fragment of ML with finite data-types, integer references but no recursion, it has been proven that contextual equivalence is undecidable [Murawski 2005]. Techniques like Kripke Logical Relations [Ahmed et al. 2009; Dreyer et al. 2010a; Pitts 1996] or various forms of bisimulations [Hur et al. 2012; Koutavas and Wand 2006; Lassen and Levy 2007; Sangiorgi et al. 2007; Sumii 2009; Sumii and Pierce 2005] have been designed to prove contextual equivalence of programs written in fragments of ML. However none of these techniques have been used to effectively automate reasoning on equivalence of programs.

In a different setting, *algorithmic game semantics* has been developed to obtain a typed-based classification of decidability of contextual equivalence for fragments of ML [Cotton-Barratt et al. 2015, 2017; Hopkins et al. 2011; Murawski 2005; Murawski and Tzevelekos 2011a, 2012]. Designed to yield decidability results, algorithmic game semantics provides an automatic technique to prove contextual equivalence only for programs with finite data-types, and "low-order" types. It uses an automata representation of the denotation of programs, in order to test them for equality

In this paper, we are rather interested in the following question:

Is it possible to algorithmically reduce the problem of contextual equivalence of two programs to the verification of a single "program-like" object that only manipulate first-order values (i.e. integers and locations) ?

Working in the setting of RefML, a fragment of ML with higher-order functions, Int and Bool datatypes, recursion, and higher-order references (but no reference disclosure), this paper provides a first answer to this question, by introducing the framework SyTeCi. It reduces the problem of contextual equivalence of two programs to the problem of ***non-reachability*** of "failed" configurations in a transition system of memory configurations, called a Structured-Memory Transition Machine (SMTM). These transition systems are represented symbolically and can be effectively constructed. They manipulate only ground values like integers and memory addresses. Paths in these SMTSs represent an abstraction of an interaction between a context and each of the two programs considered. Reaching a failed configuration means that there exists a context that can discriminate these two programs.

While we consider contexts in RefML, for technical reasons explained later, we will only consider programs in a fragment SimpleML of RefML, where references can only store integers. In RefML, there is no control operators like call/cc, so that contexts can discriminate less programs. Indeed, in this setting the sequence of calls and returns must follow a "well-bracketed" discipline, that will necessitate to have a stack in the configurations of the SMTMs that we will consider.

Importantly, our method is ***complete*** for recursion-free programs, as soon as we allow contexts to use higher-order references (that is references that can store functions). This means that given two recursion-free programs, the failed configurations of the automatically constructed SMTM are non-reachable if and only if the two programs are contextually equivalent. For programs with recursion, we may fail to build the SMTM if the two programs are too "different", so that we cannot find synchronization points between them.

Compared to algorithmic game semantics, this reduction works with unbounded data-types, by treating them symbolically, and at any type order. Notice that as soon as we have unbounded integer with basic arithmetic opertions, contextual equivalence is undecidable, even without recursion.

Restricting ourselves to programs that do not create references in the body of functions, so that the interaction w.r.t. any context always generates bounded heaps, we can encode this non-reachability problem as satisfaction of a set of constrained Horn clauses [Bjørner et al. 2015, 2013]. If furthermore programs only handle finite datatypes and are recursion-free, we prove that contextual equivalence is decidable.

A prototype implementing this work is available at https://github.com/g-jaber/SyTeCi.

2 SYTECI AT WORK

To prove contextual equivalence of higher-order programs, one has to represent the ***interaction*** between a program and a context. Game semantics provides a general framework to do so, representing interactions as traces formed by actions, which can be of four kinds:

- Player Answer: the program returns a value (boolean, integer, function);
- Player Question: the program calls a function provided by the context (i.e. a callback);
- Opponent Answer: the context returns a value to terminate a callback;
- Opponent Question: the context calls a function provided by the program.

The denotation of a program is then formed by the set of traces corresponding to the interaction of the program with all the possible contexts. A key point of game semantics is that it provides a fully abstract model – without any need of quotienting – for RefML, a typed higher-order language with higher-order references [Laird 2007; Murawski and Tzevelekos 2011b]. This means that two programs are contextually equivalent if and only if their denotations are equal.

However, building effectively the denotation of programs and comparing them can be impossible when working with programs of arbitrary types, with infinite data-types, or with recursion.

To avoid to have to consider all the possible interactions of the programs in one go, we will rather try to ***synchronize the interaction points*** of the two considered programs on the fly. Synchronizing interaction points means:

- to check they are of the same kind (Player answers or Player questions),
- for Player questions, to check that they interrogate the same function provided by the context, and that the ground values (integer or booleans) provided as arguments to this function in these questions are equal,
- for Player answer carrying ground values, to check that these values are equal.

We use an abstract representation (using free variables) for the higher-order values provided by the context to the program, similar to what is done with normal form (a.k.a. open) bisimulations [Jagadeesan et al. 2009; Lassen and Levy 2007]. One of the novelty of this work is to do the same for the ground values too, allowing to perform symbolic reasoning on the behaviour of programs. This is done using a ***symbolic evaluation***, that collects arithmetic constraints on these variables representing ground values like integers.

So types are important to determine how to represent values provided by Opponent, being boolean, integer or functional. But it is interesting to notice that, working in an untyped setting with only functional values, it would be straightforward to adapt this framework.

Since we consider programs that use references, the symbolic evaluation has to handle the heap too. It does so by using a symbolic heap and by collecting, during the evaluation, arithmetic constraints on the values stored in it. So this symbolic evaluation is used to find all the possible executions of each program until we reach an interaction point. A key point is that after each interaction point, the symbolic evaluation cannot use the knowledge it has acquired before on the heap. This is due to the fact that at an interaction point, the context could have performed some (possibly reentrant) calls to the functions that the programs have supplied, whose execution could have modified the heap.

These various synchronization points, together with the arithmetic constraints collected by the symbolic evaluation, can then be organized as a transition system representing the evolution of the memory configurations (the two heaps, a stack and the ground values provided by the context) of the two programs. We call these transitions systems Structured-Memory Transition Machines (SMTM). In these SMTMs, we represent failure of synchronization with special configurations, called ***failed configurations.***. We then have to check such failed configurations are not reachable in order to prove contextual equivalence of the two programs.

The idea of using transition systems of memory configurations to prove contextual equivalence of ML programs was advocated by Dreyer, Neis & Birkedal in a landmark paper [Dreyer et al. 2010a] on Kripke Logical Relations. This idea was later refined in [Jaber and Tabareau 2015], where such transition systems were shown to be an abstraction of the labelled transition systems used in operational game semantics [Jaber 2015; Laird 2007] to represent the denotation of programs. In [Jaber and Tabareau 2015], transition systems of memory configurations needed to prove contextual equivalence where shown to always exists. However, no effective way to build a finite representation of them, suited for automatic reasoning, was given. Our paper can thus be seen as the conclusive step in this direction, by showing that these transition systems is the only needed object to prove contextual equivalence, and by providing an algorithmic way to build them.

We now present on two examples (using the syntax of OCaml) the main ideas of this framework.

2.1 Representation Independence Example

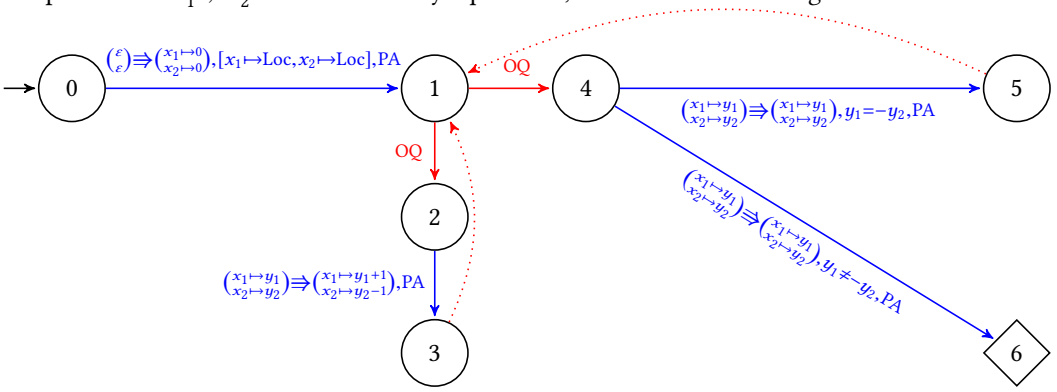
Our first example, consists in two programs $M_1^{\mathsf{ri}}, M_2^{\mathsf{ri}}$ that both provide two functions to handle a counter: an increment function `inc` and a getter function `get`. Their code, where the blue and red labels correspond to the states of the SMTM presented later, are:

M_1^{ri} :
$_0$`let c1 = ref 0 in`
`let inc () =`$_2$ `c1 :=` $\underbrace{\texttt{!c1} + 1}_{3}$ `in`
`let get () =`$_4$ $\underbrace{\texttt{!c1}}_{5,6}$ `in` $\langle \texttt{inc}, \texttt{get} \rangle_1$

M_2^{ri} :
$_0$`let c2 = ref 0 in`
`let inc () =`$_2$ `c2 :=` $\underbrace{\texttt{!c2} - 1}_{3}$ `in`
`let get () =`$_4$ $\underbrace{-\texttt{!c2}}_{5,6}$ `in` $\langle \texttt{inc}, \texttt{get} \rangle_1$

While M_1^{ri} indeed implements these functions as expected, M_2^{ri} rather chooses to decrement the counter in its `inc` function, but returns the opposite of the value of the counter in its get function. Since the references $\texttt{c}_1, \texttt{c}_2$ are not directly accessible by the context, they are indeed contextually equivalent. Even if this is not an higher-order example, contextual equivalence here does not correspond to the kind of input/output equivalences that are considered in regression verification. This is due to the fact that both programs uses a private reference that continue to exists after each call of `inc` and `get`. So each call of `inc` and `get` depends on the previous calls of these functions. Here contextual equivalence rather correspond to class equivalence [Banerjee and Naumann 2005] that we may find in object-oriented programming. It could also be reformulated as an equivalence checking problem of database-driven applications, as studied in [Wang et al. 2018].

To prove that $M_1^{\mathsf{ri}}, M_2^{\mathsf{ri}}$ are contextually equivalent, we build the following SMTM:



Let us now describe how to build and understand this SMTM system. The state 0 is the initial one. The transition from 0 to 1 corresponds to the evaluation of $M_1^{\mathsf{ri}}, M_2^{\mathsf{ri}}$ till they both reach an interaction point. In both cases, the evaluation does not need anything about the heap, and return in both cases a heap where a reference x_i has been allocated, and stores 0. This is represented by

the label $\binom{\varepsilon}{\varepsilon} \Rrightarrow \binom{x_1 \mapsto 0}{x_2 \mapsto 0}$ on the transition, where ε represent the empty heap. The typing context $[x_1 \mapsto \text{Loc}, x_2 \mapsto \text{Loc}]$ in this transition indicates that this transition creates two fresh variables x_1, x_2 of type Loc. The evaluation of $M_1^{\text{ri}}, M_2^{\text{ri}}$ returns in both cases a pair of values $\langle \texttt{inc}, \texttt{get} \rangle$, corresponding to a Player Answer, so that we can synchronize these interaction points.

In state 1, the context (i.e. Opponent) chooses what to do. It has two possibilities:

- either interrogate `inc` in both cases, which corresponds to the transition from 1 to 2;
- or interrogate `get` in both cases, which corresponds to the transition from 1 to 4.

In the first case, the evaluation of `inc` necessitates in both cases that the domain of the heap contains respectively x_1 and x_2. In the first program, the value stored in x_1 will be incremented, while in the second program, it will be decremented. This is represented by the label $\binom{x_1 \mapsto y_1}{x_2 \mapsto y_2} \Rrightarrow \binom{x_1 \mapsto y_1 + 1}{x_2 \mapsto y_2 - 1}$ on the transition. Since these evaluations return () (i.e. the unit value) in both cases, corresponding to a Player Answer, we can indeed synchronize these interaction points. Then, Opponent has the control, and can go back to the state 1 via an Oϵ-transition to continue the interaction.

In the second case the evaluation of `get` necessitates again in both cases that the domain of the heap contains respectively x_1 and x_2. Each program returns the integer stored in x_i, written y_i, corresponding again to a Player Answer. However, in order to synchronize these interactions, y_1 must be equal to $-y_2$. The state 5 corresponds to this case. When it is not the case, the synchronization fails, which correspond to the diamond state 6. Since the heap is not modified in both cases, these transitions are thus labelled $\binom{x_1 \mapsto y_1}{x_2 \mapsto y_2} \Rrightarrow \binom{x_1 \mapsto y_1}{x_2 \mapsto y_2}$. When it has been possible to synchronize the interactions, corresponding to state 5, Opponent can continue the interaction by going back to the state 1 via an Oϵ-transition.

Runs of such SMTMs systems work on configurations, that contains, in addition to a state s, two heaps h_1, h_2 whose evolution is dictated by the labels on the Player transitions. On Opponent transitions, these heaps are left unchanged. Note that the variables x_1, x_2 are common to the labels of all the Player transitions. To handle them, configurations also come with an environment η that maps these variables to their values.

So an example of a run of this SMTM, corresponding to the interaction with the context $C = \texttt{let}\langle \texttt{f}, \texttt{g} \rangle = \bullet \texttt{ in f}(); \texttt{g}()$ that calls first `inc` then `get`, is given by:

$$\begin{aligned}(0, \varepsilon, \varepsilon, \varepsilon) \quad &\xrightarrow{\text{PA}} (1, \eta, [\ell_1 \mapsto 0], [\ell_2 \mapsto 0]) \xrightarrow{\text{OQ}} (2, \eta, [\ell_1 \mapsto 0], [\ell_2 \mapsto 0]) \xrightarrow{\text{PA}} (3, \eta, h_1, h_2) \\ &\xrightarrow{\text{O}\epsilon} (1, \eta, h_1, h_2) \xrightarrow{\text{OQ}} (4, \eta, h_1, h_2) \xrightarrow{\text{PA}} (5, \eta, h_1, h_2)\end{aligned}$$

with $\eta = [x_1 \mapsto \ell_1, x_2 \mapsto \ell_2]$ with ℓ_1, ℓ_2 two locations, $h_1 = [\ell_1 \mapsto 1]$ and $h_2 = [\ell_2 \mapsto -1]$.

Notice that the last transition $(4, \eta, h_1, h_2) \xrightarrow{\text{PA}} (5, \eta, h_1, h_2)$ is possible because $h_1(\ell_1) = -h_2(\ell_2)$, otherwise we would have reached the failed configuration $(6, \eta, h_1, h_2)$

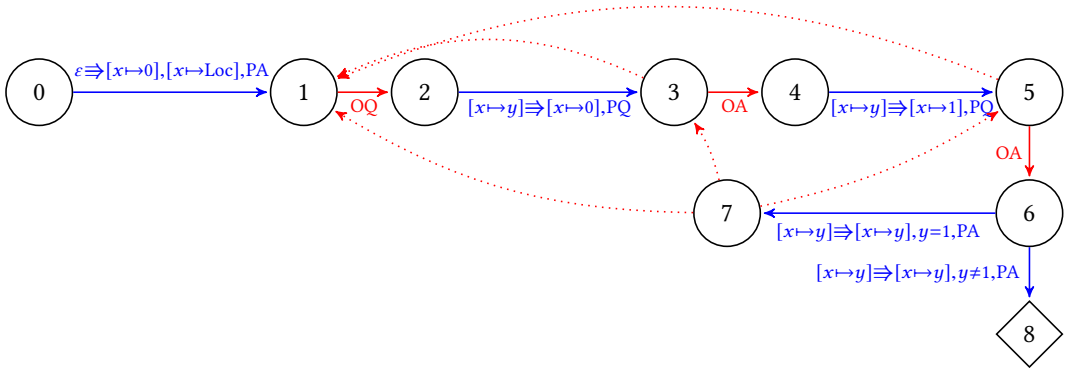
To prove that $M_1^{\text{ri}}, M_2^{\text{ri}}$ are contextually equivalent, we then have to prove that there is no run starting in a configuration whose state is 0 and ending in a configuration whose state is 6. To do so, we can simply prove that the invariant $\binom{x_1 \mapsto y_1}{x_2 \mapsto y_2}, y_1 = -y_2$ is true in all the configurations of any run of the transition system after the initial configuration starting in 0, so that the a configuration whose state is 4 cannot go to a configuration whose state is 6.

2.2 Well-Bracketed State Change Example

We now consider an example of higher-order programs that perform callbacks. It is called the ***well-bracketed state change example***, since it relies crucially on the fact that contexts must deal with callbacks in a well-bracketed way, since they cannot use control operators.

$$M_1^{\text{wbsc}} : \begin{array}{l} {}_0\texttt{let x = ref0 in} \\ {}_1\texttt{fun f} \to_2 \texttt{x} := \texttt{0};_3\texttt{f}()_4; \texttt{x} := \texttt{1};_5\texttt{f}()_6; !\texttt{x}_{7,8} \end{array} \quad \Big| \quad M_2^{\text{wbsc}} : \quad {}_{0\,1}\texttt{fun f} \to_{2\,3} \texttt{f}()_4;_5\texttt{f}()_{6,7}; \texttt{1}$$

In order to prove that $M_1^{\mathtt{wbsc}}, M_2^{\mathtt{wbsc}}$ are equivalent, one must show that after performing the second callback to `f` in $M_1^{\mathtt{wbsc}}$, the value stored in the reference x is equal to 1. The difficulty to do so is that when the program performs a callback to `f`, the context has the control and can decide to call reentrantly the function it got as a result of the evaluation of $M_1^{\mathtt{wbsc}}$. In such a case, the reference x will be set temporarily back to 0. The SMTM we build to prove this example is the following:



On Player transitions, we only give the evolution of the heap for the first program, since for the second one the heap is always empty. The state 3 corresponds to the point where the two programs have performed the first callback to `f`. At this point, Opponent can either answer back directly, by taking the transition from 3 to 4, or go back to the state 1 via an Oϵ-transition, where he can perform a reentrant call to the function. The same situation occurs in state 5, that corresponds to the second callback to `f` for the two programs.

We want that for any run of this SMTM, there is indeed a context that produce the interaction represented by this run. However, there exists a run going through the states:

$$0 \xrightarrow{\text{PA}} 1 \xrightarrow{\text{OQ}} 2 \xrightarrow{\text{PQ}} 3 \xrightarrow{\text{O}\epsilon} 1 \xrightarrow{\text{OQ}} 2 \xrightarrow{\text{PQ}} 3 \xrightarrow{\text{OA}} 4 \xrightarrow{\text{PQ}} 5 \xrightarrow{\text{OA}} 6 \xrightarrow{\text{PA}} 7 \xrightarrow{\text{O}\epsilon} 5 \xrightarrow{\text{OA}} 6 \rightarrow \dots$$

The Player question $2 \xrightarrow{\text{PQ}} 3$ is then answered by the transition $5 \xrightarrow{\text{OA}} 6$, which operationally corresponds to the fact that after performing the first callback to `f`, the context perform a reentrant call, which once terminates return to the control point of the second callback. But the context cannot do that, since it does not have access to control operator like call/cc.

To forbid such runs, we equip configurations c of the SMTM with a stack σ of states. Player questions $c \xrightarrow{\text{PQ}} c'$ pushes on top of the stack of c' the state associated to c', and an Opponent answer $c \xrightarrow{\text{OA}} c'$ can be taken only if the top of the stack of c is equal to the state associated to c, in which case we pop the first element of the stack. The stack is also used to store the current environment η before a callback and to restore it after.

So a possible run (where we do not indicate in configurations the always empty second heap) is:

$$\begin{aligned}(0, \varepsilon, [], \varepsilon) &\xrightarrow{\text{PA}} (1, \eta, [], h_0) \xrightarrow{\text{OQ}} (2, \eta, [], h_0) \xrightarrow{\text{PQ}} (3, \eta, \sigma, h_0) \xrightarrow{\text{O}\epsilon} (1, \eta, \sigma, h_0) \xrightarrow{\text{OQ}} (2, \eta, \sigma, h_0)\\ &\xrightarrow{\text{PQ}} (3, \eta, (3, \eta) :: \sigma, h_0) \xrightarrow{\text{OA}} (4, \eta, \sigma, h_0) \xrightarrow{\text{PQ}} (5, \eta, (5, \eta) :: \sigma, h_1) \xrightarrow{\text{OA}} (6, \eta, \sigma, h_1) \xrightarrow{\text{PA}} (7, \eta, \sigma, h_1)\end{aligned}$$

with $\eta = [x \mapsto \ell], h_0 = [\ell \mapsto 0], h_1 = [\ell \mapsto 1]$ and $\sigma = [(3, \eta)]$.

Then from $(7, \eta, \sigma, h_1)$ we could take the Oϵ-transition to go to the configuration $(5, \eta, \sigma, h_1)$, but we would get stuck here since we cannot go to $(6, \eta, [], h_1)$, because $\sigma = [(3, \eta)]$. But we could go to the configuration $(3, \eta, \sigma, h_1)$ and then to $(4, \eta, [], h_1)$, which correspond to a well-bracketed run. Notice that for this example, since the environment η is always the same, the fact that we save it in the stack after a Player Question and restore it after an Opponent Answer is not visible.

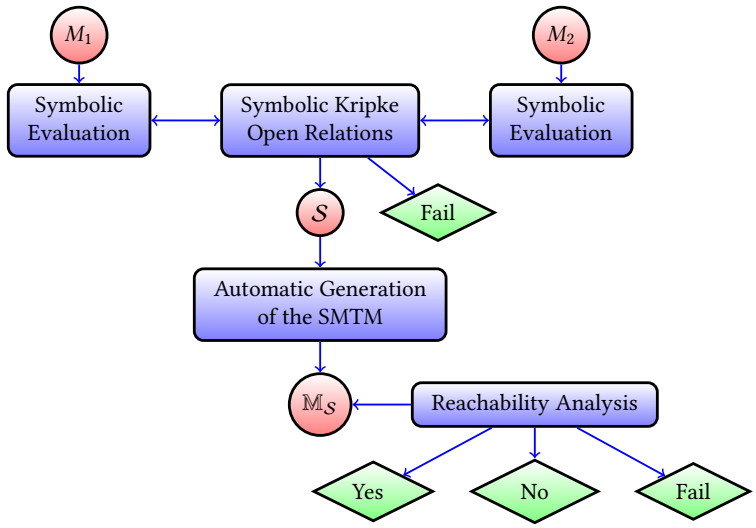


Fig. 1. Organization of SyTeCi

3 THE ARCHITECTURE OF SYTECI

To build automatically the SMTM $\mathbb{M}$, our framework is organized in Figure 1 The first step is ***Symbolic Kripke Open Relations*** (SKORs), introduced in Section 6. They are relational predicates $\mathbb{E}[\![\tau]\!]$ on programs of type τ, defined via a set of inference rules. Given two programs M_1, M_2, starting from $\mathbb{E}[\![\tau]\!](M_1, M_2)$, we build a derivation tree $\mathcal{S}$ using these rules. It enforces the synchronization of the interaction points between each program M_1, M_2 and any context.

We then present in Section 7 how to transform this derivation tree $\mathcal{S}$ into a ***SMTM*** $\mathbb{M}_{\mathcal{S}}$. Contextual equivalence is then reduced to non-reachability of failed configurations in $\mathbb{M}_{\mathcal{S}}$.

In Section 8, we extend our framework to handle recursion. For programs with recursion, the symbolic evaluation could diverge, or be arbitrary long. To avoid this problem, the symbolic evaluation gets stuck on recursive calls. Then, SKORs try to synchronize recursive calls between the two programs. To do so, we generalize the inference system to allow back-edges, thus getting derivation graphs. These back-edges correspond to ***circular reasoning***, allowing us to terminate the structure once we reach a formula already seen before.

In Section 9, we present how to encode the reachability of failed configurations as a set of constrained Horn clauses, which are unsatisfiable when such failed configurations are unreachable. This encoding is only possible when the heaps we get in the runs of $\mathbb{M}_{\mathcal{S}}$ are bounded. We enforce this condition by considering programs that never create references inside the body of functions. We also deduce a decidability result for contextual equivalence when we restrict furthermore to recursion-free closed programs that manipulate finite datatypes.

Finally, in Section 10, we sketch the proof of soundness, and completeness for recursion-free programs, of our framework.

Note that we could have tried to give algorithm to build $\mathbb{M}_{\mathcal{S}}$ directly from M_1, M_2, without the detour through $\mathcal{S}$. But two reasons prevented us from doing this. First, it would not have been possible to perform circular reasoning in this setting. Secondly, our soundness and completeness proofs relies on the detour through $\mathcal{S}$.

A prototype implementing this work, that generates the derivation graph $\mathcal{S}$, the SMTM $\mathbb{M}_{\mathcal{S}}$, and the constrained Horn clauses associated to the reachability problem of the failed configurations of $\mathbb{M}_{\mathcal{S}}$, is available at https://github.com/g-jaber/SyTeCi.

$$
\begin{array}{lll}
\sigma, \tau & \triangleq & \mathrm{Unit} \mid \mathrm{Int} \mid \mathrm{Bool} \mid \mathrm{ref}\ \tau \mid \tau \times \sigma \mid \tau \to \sigma \\
u, v & \triangleq & () \mid \mathbf{true} \mid \mathbf{false} \mid \widehat{n} \mid x \mid \ell \mid \mathbf{fun}\ x \to M \mid \mathbf{fix}\ y(x) \to M \mid \langle u, v \rangle \\
M, N & \triangleq & v \mid MN \mid M \oplus N \mid M \boxdot N \mid \mathrm{ref}\ M \mid\ !M \mid\ \mid M := N \mid \mathrm{if}\ M_1\ M_2\ M_3 \mid \langle M, N \rangle \mid \pi_1 M \mid \pi_2 M \\
K & \triangleq & \bullet \mid vK \mid KM \mid \langle v, K \rangle \mid \langle K, M \rangle \mid \mathrm{ref}\ K \mid\ !K \mid v := K \mid K := M \mid v \oplus K \mid \ \ldots \\
C & \triangleq & \bullet \mid \mathbf{fun}\ x \to C \mid \mathbf{fix}\ y(x) \to C \mid MC \mid CM \mid \langle M, C \rangle \mid \langle C, M \rangle \mid \mathrm{ref}\ C \mid\ !C \mid M := C \mid \ \ldots
\end{array}
$$

Syntactic Sugar:
$\mathtt{let}\ f\ x = M\ \mathtt{in}\ N \triangleq (\mathbf{fun}\ f \to N)(\mathbf{fun}\ x \to M)$
$\mathtt{let\ rec}\ f\ x = M\ \mathtt{in}\ N \triangleq (\mathbf{fun}\ f \to N)(\mathbf{fix}\ f(x) \to M)$
$M; N \triangleq (\mathbf{fun}\ x \to N)M \qquad (\text{with } x \notin \mathrm{FV}(N))$

Fig. 2. Definition of RefML (with $n \in \mathbb{Z}$, $\oplus \in \{+, -, *\}$, $\boxdot \in \{=, <\}$, $x, y \in \mathrm{Var}$ and $\ell \in \mathbf{Loc}$)

4 NOTATIONS

In the following, we make heavy use of partial maps with finite domain $f : A \rightharpoonup B$ between two sets A, B. Such partial maps can be seen as functional relations between A and B, and we use indifferently the two representations. We write ε for the empty partial map (corresponding to the empty set $\varnothing$), and $\mathrm{dom}(f)$, $\mathrm{codom}(f)$ for the domain and the codomain of f. We define:

- the extension $f \cdot [a \mapsto b]$ as $f \cup \{(a, b)\}$, provided that $a \notin \mathrm{dom}(f)$;
- the modification $f[a \mapsto b]$ as $\{(x, f(x)) \mid x \in \mathrm{dom}(f) \backslash \{x\}\} \cup \{(a, b)\}$;
- the restriction $f_{|A'}$, when $A' \subseteq A$, as $\{(x, f(x)) \mid x \in \mathrm{dom}(f) \cap A'\}$.

Taking two relations $R_1 : A \times B$ and $R_2 : B \times C$, we define their composition $R_1 \circ R_2 : A \times C$ as $\{(a, c) \mid \exists b \in B, (a, b) \in R_1 \wedge (b, c) \in R_2\}$. For a relation $R \subseteq A \times A$ and $i \in \mathbb{N}$, we write $R^=$, R^+ and R^* respectively for the reflexive, the transitive and the reflexive-transitive closure of R.

We often manipulate large tuples, so we use an informal record-like notation $\{(a : A, b : B, \ldots) \mid \ldots\}$ to define them, whose meaning is $\{(a, b, \ldots) \in A \times B \times \ldots \mid \ldots\}$. Then, for such a tuple x, we write $x.a, x.b, \ldots$ for the various components (i.e. the fields) of x.

5 THE LANGUAGES: REFML & SIMPLEML

5.1 First Definitions

RefML is a typed call-by-value λ-calculus with full references (both ground and higher-order) and explicit fixed points definitions. The grammars of its types, values, terms, evaluation contexts and contexts is given in Figure 2. A term is said to be ***recursion-free*** if it does not have any subterms of the form $\mathbf{fix}\ y(x) \to M$. In the following, we consider various forms of finite-domain partial maps m from variables to terms, called substitution, valuations or environments depending on the constraints put on the codomain of m. We then write $M\{m\}$ for the term $M\{x_1 := m(x_1)\} \cdots \{x_n := m(x_n)\}$, where $\mathrm{dom}(m) = \{x_1, \ldots, x_n\}$. It is uniquely defined (i.e. the order on the variables in $\mathrm{dom}(m)$ is irrelevant in the definition of $M\{m\}$) as soon as $\mathrm{FV}(\mathrm{codom}(m)) \cap \mathrm{dom}(m) = \varnothing$.

Typing judgements are of the form $\Sigma; \Gamma \vdash M : \tau$, where Σ is a location context and Γ a variable context, i.e. finite partial function respectively from locations and variables to types. They are standard and are given in Appendix A.

We then consider a fragment of RefML, called SimpleML, that corresponds to terms that ***only create and manipulate private integer references***.

Definition 5.1. SimpleML terms are terms M of RefML s.t. there exists a typing derivation $\varepsilon; \Gamma \vdash M : \tau$, where:

- The typing rule for ref N in the typing derivation of M is restricted to ref Int;
- There is no occurrence of ref σ (even for $\sigma = \mathrm{Int}$) in τ or in $\mathrm{codom}(\Gamma)$.

This last condition corresponds to the fact that the term cannot disclose any location to the context, or get locations provided by the context.

$$
\begin{array}{l|l}
(K[(\textbf{fun}\ x \to M)v], h) \xrightarrow{r} (K[M[x := v]], h) & (K[\pi_i\langle u_1, u_2\rangle], h) \xrightarrow{r} (K[u_i], h) \\
(K[\textbf{if}\ \textbf{true}\ M_1\ M_2]], h) \xrightarrow{r} (K[M_1], h) & (K[!\ell], h) \xrightarrow{r} (K[h(\ell)], h) \\
(K[\textbf{if}\ \textbf{false}\ M_1\ M_2]], h) \xrightarrow{r} (K[M_2], h) & (K[\text{ref}\, v], h) \xrightarrow{r} (K[\ell], h \cdot [\ell \mapsto v]) \\
(K[\widehat{n} \oplus \widehat{m}], h) \xrightarrow{r} (K[\widehat{n \oplus m}], h) & (K[\ell := v], h) \xrightarrow{r} (K[()], h[\ell \mapsto v]) \\
(K[\hat{n} \boxdot \hat{m}], h) \xrightarrow{r} (K[b], h) & (K[\ell = \ell'], h) \xrightarrow{r} (K[b], h) \\
\text{with } b = \textbf{true} \text{ if } n \boxdot m, \text{otherwise } b = \textbf{false} & \text{with } b = \textbf{true} \text{ if } \ell = \ell', \text{otherwise } b = \textbf{false}
\end{array}
$$

$$
\frac{(M, h) \xrightarrow{r} (M', h')}{(M, h) \to (M', h')} \qquad (K[\overbrace{(\textbf{fix}\ y(x) \to M)}^{u} v], h) \to (K[M\{x := v, y := u\}], h)
$$

Fig. 3. Operational reduction for RefML

5.2 Operational Semantics and Contextual Equivalence

The operation semantics of RefML manipulates pairs (M, h) of a term M and a ***heap*** h, which is a finite partial map $\textbf{Loc} \rightharpoonup \text{Val}$ from locations to values. The operational reduction is defined in Figure 3 via two reductions $\xrightarrow{r}$ and $\to$. The reduction $\xrightarrow{r}$ does not reduce fixed-points, while $\to$ is the standard reduction. Using this full reduction, we define ***contextual equivalence*** for terms of SimpleML, with contexts in RefML.

Definition 5.2. Taking two SimpleML terms M_1, M_2 s.t. $\cdot; \Gamma \vdash M_1, M_2 : \tau$, we say that M_1 and M_2 are ***contextually equivalent***, written $\Gamma \vdash M_1 \simeq_{\text{ctx}} M_2 : \tau$, when for all RefML contexts C s.t. $\cdot; x : \tau \vdash C[x] : \text{Unit}$ with $x \notin \text{dom}(\Gamma)$, $(C[M_1], \varepsilon) \to^* ((), h_1)$ iff $(C[M_2], \varepsilon) \to^* ((), h_2)$.

6 SYMBOLIC KRIPKE OPEN RELATIONS

We now introduce a symbolic framework to reason about contextual equivalence of programs. It is based on a set of inference rules to reason about *Symbolic Kripke Open Relations* (SKORs). The main idea is to find the interaction points (Player answer or questions) of the two programs considered, in order to synchronize them. Symbolic evaluation is used to find such interaction points.

We introduce the symbolic values in Section 6.1, and the ground logic used to represent arithmetic constraints and to reason about symbolic heaps in Section 6.2. Then we present the symbolic evaluation of open terms in Section 6.3. In Section 6.4, we present the basic rules defining SKORs. Finally, in Section 6.6 we show how to build automatically a derivation tree relating two terms.

6.1 Symbolic Values

Symbolic evaluation will handle terms with free variables. However, to make it simple, we only allow free variable of type Int, ref Int or a functional type $\tau \to \sigma$. ***Ground variable contexts*** $\Delta \in \text{GContext}$ are then used for variables of ***ground types*** Int and ref Int (so neither Bool nor Int×Int are considered as ground!). We write Δ_{Loc} for $\Delta^{-1}(\text{ref Int})$ and Δ_{Int} for $\Delta^{-1}(\text{Int})$. ***Functional variable contexts*** Ξ are used for variables of functional types.

We introduce ***symbolic values*** to represent symbolically (i.e. using variables) integers, locations and functional values, and concretely pairs, booleans and unit. They are defined via the set $\textbf{SVal}(\tau)$ formed by triples (u, Ξ, Δ) of a value, a functional variable context and a ground variable context:

$$
\begin{array}{ll}
\textbf{SVal}(\text{Unit}) \triangleq \{((), \varepsilon, \varepsilon)\} & \textbf{SVal}(\text{Bool}) \triangleq \{(\textbf{true}, \varepsilon, \varepsilon), (\textbf{false}, \varepsilon, \varepsilon)\} \\
\textbf{SVal}(\text{Int}) \triangleq \{(x, \varepsilon, [x \mapsto \text{Int}]) \mid x \in \text{Var}\} & \\
\textbf{SVal}(\tau \to \sigma) \triangleq \{(x, [x \mapsto (\tau \to \sigma)], \varepsilon) \mid x \in \text{Var}\} & \\
\textbf{SVal}(\sigma_1 \times \sigma_2) \triangleq \{(\langle u_1, u_2\rangle, \Xi_1 \cup \Xi_2, \Delta_1 \cup \Delta_2) \mid \forall i \in \{1, 2\}, (u_i, \Xi_i, \Delta_i) \in \textbf{SVal}(\sigma_i) & \\
\qquad \wedge \text{dom}(\Delta_1) \cap \text{dom}(\Delta_2) = \varnothing \wedge \text{dom}(\Xi_1) \cap \text{dom}(\Xi_2) = \varnothing\} &
\end{array}
$$

So taking $(u, \Xi, \Delta) \in \textbf{SVal}(\tau)$, all the (necessarily free) variables of u appear only once in u. Because we do not allow boolean variables, we have to represent them concretely. Since symbolic values

$$
\begin{array}{lll}
(M, \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (M', \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \text{when } (M, \varepsilon) \xrightarrow{r} (M', \varepsilon) \\
(K[t \boxdot t'], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (K[\mathbf{true}], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C} \cup \{t \boxdot t'\}) & \\
(K[t \boxdot t'], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (K[\mathbf{false}], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C} \cup \{\neg(t \boxdot t')\}) & \\
(K[t \oplus t'], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (K[x], \Delta \cdot [x \mapsto \text{Int}], \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C} \cup \{x = t \oplus t'\}) & \\
(K[!x], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (K[\mathbb{H}^{\text{post}}(x)], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \text{if } x \in \text{dom}(\mathbb{H}^{\text{post}}) \\
(K[!x], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (K[y], \Delta \cdot [y \mapsto \text{Int}], \mathbb{H}^{\text{pre}} \cdot [x \mapsto y], \mathbb{H}^{\text{post}} \cdot [x \mapsto y], \mathbb{C}) & \text{otherwise} \\
(K[x := u], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (K[()], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}[x \mapsto u], \mathbb{C}) & \text{if } x \in \text{dom}(\mathbb{H}^{\text{post}}) \\
(K[x := u], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (K[()], \Delta \cdot [y \mapsto \text{Int}], \mathbb{H}^{\text{pre}} \cdot [x \mapsto y], \mathbb{H}^{\text{post}} \cdot [x \mapsto u], \mathbb{C}) & \text{otherwise} \\
(K[\text{ref } u], \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C}) & \mapsto_s (K[x], \Delta \cdot [x \mapsto \text{ref Int}], \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}} \cdot [x \mapsto u], \mathbb{C}) &
\end{array}
$$

Fig. 4. Definition of Symbolic Reduction

will be used to represent values that the programs and the context exchange, we do not need to define symbolic values for reference types, since types of programs of SimpleMLcannot contain occurences of reference types.

We then introduce (typed) ***symbolic substitutions*** $(\mu, \Xi, \Delta) \in \text{SSubst}(\Gamma)$, which are used to transform variables of type Unit, Bool and $\tau \times \sigma$ from Γ into variables of ground or functional types. They are defined as $\text{SSubst}(\varepsilon) \triangleq \{(\varepsilon, \varepsilon, \varepsilon)\}$ and

$\text{SSubst}(x : \theta, \Gamma) \triangleq \{(\mu \cdot [x \mapsto u], \Xi_1 \cdot \Xi_2, \Delta_1 \cdot \Delta_2) \mid (u, \Xi_1, \Delta_1) \in \mathbf{SVal}(\theta), (\mu, \Xi_2, \Delta_2) \in \text{SSubst}(\Gamma)\}$

6.2 Ground Logic

We now introduce a logic to reason symbolically on variables of ground type. It is based on arithmetic formula $F \in \text{AForms}$ that manipulates ***ground terms*** $t \in \text{GTerms}$, which are terms of ground type whose free variables are also of ground type. Their grammars are:

$$
\begin{array}{lll}
t, t' & \triangleq & x \mid \widehat{n} \mid t \oplus t' \text{ with } x \in \text{Var}, n \in \mathbb{Z} \text{ and } \oplus \in \{+, -, *\} \\
F & \triangleq & \text{True} \mid \text{False} \mid t \boxdot t' \mid \neg F \text{ with } \boxdot \in \{=, <\}
\end{array}
$$

We consider ***ground valuations*** $\eta \in \text{GVal}(\Delta)$, which are maps from $\text{dom}(\Delta)$ to values s.t. for all $x \in \Delta_{\text{Int}}, \eta(x) \in \text{Int}$ an for all $x \in \Delta_{\text{Loc}}, \eta(x) \in \mathbf{Loc}$. Using a ground valuation η, we define the semantics of ground terms $[\![t]\!]_\eta$ and the validity of a formula $\models_\eta F$:

$$
[\![x]\!]_\eta \triangleq \eta(x) \qquad [\![t_1 \oplus t_2]\!]_\eta \triangleq [\![t_1]\!]_\eta \oplus [\![t_2]\!]_\eta \qquad [\![n]\!]_\eta \triangleq \widehat{n}
$$

$$
\models \text{True} \triangleq \text{True} \qquad \models \text{False} \triangleq \text{False} \qquad \models_\eta t_1 \boxdot t_2 \triangleq [\![t_1]\!]_\eta \boxdot [\![t_2]\!]_\eta \qquad \models_\eta \neg F \triangleq \neg(\models_\eta F)
$$

In the following, we consider sets of arithmetic formulas $\mathbb{C}$. We say that $\mathbb{C}$ is ***satisfiable*** (w.r.t. a ground variable context $\Delta \supseteq \text{FV}(\mathbb{C})$) when there exists $\eta \in \text{GVal}(\Delta)$ s.t. $\models_\eta F$ for all $F \in \mathbb{C}$. We write $\text{GVal}(\Delta, \mathbb{C})$ for the set $\{\eta \in \text{GVal}(\Delta) \mid \models_\eta \mathbb{C}\}$.

To represent heaps, we introduce ***symbolic heaps*** $\mathbb{H}$, which are finite partial maps $\text{Var} \rightharpoonup \text{GTerms}$. From a symbolic heaps $\mathbb{H}$ and a ground typing context Δ s.t. $\text{dom}(\mathbb{H}) \subseteq \Delta_{\text{Loc}}$ and $\text{FV}(\text{codom}(\mathbb{H})) \subseteq \Delta_{\text{Int}}$, taking a ground valuation $\eta \in \text{GVal}(\Delta)$, we may want to transform $\mathbb{H}$ into a (concrete) heap using η. But applying such substitutions do not always produce a partial map, since η may alias two different variables x, y to the same location ℓ. To avoid this problem, we need η to be ***injective*** on Δ_{Loc}. In such a case we define $\uparrow^\eta(\mathbb{H})$ as the heap $h \triangleq \{(\eta(x), [\![\mathbb{H}(x)]\!]_\eta) \mid x \in \text{dom}(\mathbb{H})\}$.

6.3 Symbolic Evaluation

We now introduce in Figure 4 a way to compute symbolically the normal forms of a recursion-free term M. This symbolic evaluation $\mapsto_s$ works on configuration $(M, \Delta, \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C})$ where M is a term whose free variables are either of ground type or of functional type; Δ is a ground variable context; $\mathbb{H}^{\text{pre}}$ and $\mathbb{H}^{\text{post}}$ are two symbolic heaps; and $\mathbb{C}$ is a set of arithmetic formulas. We write $\Downarrow_s(M, \Delta)$ for the set of all the normal forms (w.r.t. $\mapsto_s^*$) of the symbolic configurations $(M, \Delta, \varepsilon, \varepsilon, \varnothing)$.

This reduction is non-deterministic and generates all possible terms and symbolic heaps that can be obtained from M. Since M do not contain free variables of type Bool, we do not need to incorporate specific rules for the reduction of the if construction (it will be reduced directly using $\xrightarrow{r}$ via the first rule of Figure 4). Moreover, the symbolic reduction performs a small footprint analysis, by computing the smallest symbolic heap $\mathbb{H}^{\text{pre}}$ that is necessary to reduce the term.
Considering the term $M \triangleq \mathsf{c} :=!\mathsf{c} + 1$, one has:

$$\begin{aligned}(M, [c \mapsto \text{ref Int}], \varepsilon, \varepsilon, \varnothing) &\mapsto_s (c := x + 1, [c \mapsto \text{ref Int}] \cdot [x \mapsto \text{Int}], [c \mapsto x], [c \mapsto x], \varnothing)\\ &\mapsto_s (c := y, [c \mapsto \text{ref Int}] \cdot [x, y \mapsto \text{Int}], [c \mapsto x], [c \mapsto x], \{y = x + 1\})\\ &\mapsto_s ((), [c \mapsto \text{ref Int}] \cdot [x, y \mapsto \text{Int}], [c \mapsto x], [c \mapsto y], \{y = x + 1\})\end{aligned}$$

The following theorem, proven in Appendix C, relates the symbolic reduction to the operational reduction $\xrightarrow{r}$.

Theorem 6.1. *Given:*

- *a term $\widetilde{M}$ s.t. $\Delta \cdot \Xi \vdash \widetilde{M} : \tau$ where Δ (resp. Ξ) is a ground (resp. functional) variable context;*
- *a ground valuation $\eta \in \text{GVal}(\Delta)$ injective on Δ_{Loc} and a heap h s.t. $\text{dom}(h) = \Delta_{\text{Loc}}$.*

Writing M for $\widetilde{M}\{\eta\}$ we have $(M, h) \xrightarrow{r}{}^{*} (M', h')$ if and only if $(\widetilde{M}, \Delta, \varepsilon, \varepsilon, \varnothing) \mapsto_s^* (\widetilde{M'}, \Delta', \mathbb{H}^{\text{pre}}, \mathbb{H}^{\text{post}}, \mathbb{C})$ and there exists a ground valuation $\eta' \in \text{GVal}(\Delta', \mathbb{C})$ s.t. η' contains η and is injective on Δ'_{Loc}; $M' = \widetilde{M'}\{\eta'\}$; and there exists a heap h^f such that $h = \uparrow^{\eta'}(\mathbb{H}^{\text{pre}}) \cdot h^f$ and $h' = \uparrow^{\eta'}(\mathbb{H}^{\text{post}}) \cdot h^f$.

6.4 Derivation Trees

We now introduce in Figure 5 the inference rules that define the Symbolic Kripke Open Relations. They manipulate ***sequents*** of the form $\Delta_\alpha; \mathbb{C} \vdash \phi$ where:

- Δ is a ground variable context, and $\mathbb{C}$ is a set of arithmetic formulas;
- $\alpha : \Delta_{\text{Loc}} \to \{1, 2\}$ is a map that keep track of the owner (the left or the right term) of each location variable, and is used to enforce non-aliasing;
- ϕ is a ***Symbolic Kripke Open Relation***, which can be:
  - $\mathbb{E}[\![\tau]\!]_\Xi(M_1, M_2)$, $\mathbb{V}[\![\tau]\!]_\Xi(v_1, v_2)$, or $\mathbb{K}[\![\sigma_2, \tau]\!]_\Xi(K_1, K_2)$, which are predicates respectively on pairs of terms, values and evaluation contexts;
  - $\mathbb{S}^{\mathbf{x}}[\![\tau]\!]_\Xi(M_1, M_2)$ for $\mathbf{x} \in \{\text{PQ}, \text{PA}, \lightning\}$, which are auxiliary relations on terms used to reason about the various possible synchronization points, with $\mathbb{S}^{\lightning}[\![\tau]\!]_\Xi(M_1, M_2)$ corresponding to a failed synchronization.

From these inference rules, we build a ***derivation tree*** $\mathcal{S}$, which is an oriented pointed tree $(\mathcal{N}, \text{r}, \text{Sequent}, \text{Edge}, \text{Annot})$ where:

- $\mathcal{N}$ is a finite set of ***nodes*** and $\text{r} \in \mathcal{N}$ is the root of the tree;
- Sequent is the function mapping nodes to sequents;
- $\text{Edge} : \mathcal{N} \to \mathcal{P}(\mathcal{N}) \times \text{Rules}$ is the ***edge function*** mapping a node n to a pair $(\{\text{n}_1, \ldots, \text{n}_n\}, \text{L})$ formed by the set of its sons and a label L.
  $\dfrac{\text{Sequent}(\text{n}_1) \ldots \text{Sequent}(\text{n}_k)}{\text{Sequent}(\text{n})}$ must correspond to the rule of the given label L in Figure 5.
- Annot maps nodes to the extra annotations that appears in rules E.

We now detail the rules of Figure 5:

- Unsat terminates the tree once $\mathbb{C}$ is unsatisfiable.
- V_ι terminates the tree once we have reached an equivalence of ground typed value. The verification that $v_1 = v_2$ will then be done at a later stage, when reasoning on the runs of the SMTM generated from the derivation tree. More precisely, when building this SMTM, a failed state with a transition containing the constraint $v_1 \neq v_2$ will be generated, so that checking that no configuration associated to this failed state is reachable will amount to check that this constraint $v_1 = v_2$ is indeed always verified.

$$\mathrm{V}_\iota\ \frac{\iota \in \{\mathrm{Unit}, \mathrm{Bool}, \mathrm{Int}\}}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{V}[\![\iota]\!]_\Xi(v_1, v_2)} \qquad \frac{\mathbb{C}\ \text{unsat.}}{\Delta_\alpha; \mathbb{C} \vdash \phi}\ \textsc{Unsat} \qquad \frac{}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{S}^{\lightning}[\![\tau]\!]_\Xi(M_1, M_2)}\ \mathrm{S}^{\lightning}$$

$$\mathrm{V}_\times\ \frac{\Delta_\alpha; \mathbb{C} \vdash \mathbb{V}[\![\sigma_1]\!]_\Xi(v_1, v_1') \qquad \Delta_\alpha; \mathbb{C} \vdash \mathbb{V}[\![\sigma_2]\!]_\Xi(v_2, v_2')}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{V}[\![\sigma_1 \times \sigma_2]\!]_\Xi(\langle v_1, v_2\rangle, \langle v_1', v_2'\rangle)} \qquad \frac{\Delta_\alpha; \mathbb{C} \vdash \mathbb{V}[\![\tau]\!]_\Xi(v_1, v_2)}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{S}^{\mathrm{PA}}[\![\tau]\!]_\Xi(v_1, v_2)}\ \mathrm{S}^{\mathrm{PA}}$$

$$\frac{\Delta_\alpha; \mathbb{C} \vdash \mathbb{K}[\![\sigma_2, \tau]\!]_\Xi(K_1, K_2) \qquad \Delta_\alpha; \mathbb{C} \vdash \mathbb{V}[\![\sigma_1]\!]_\Xi(v_1, v_2) \qquad \Xi(f) = \sigma_1 \to \sigma_2}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{S}^{\mathrm{PQ}}[\![\tau]\!]_\Xi(K_1[f\ v_1], K_2[f\ v_2])}\ \mathrm{S}^{\mathrm{PQ}}$$

$$\mathrm{V}_\to\ \frac{\begin{array}{c}\text{for all } (u, \Xi', \Delta') \in \mathbf{SVal}(\sigma) \\ (\Delta \cdot \Delta')_\alpha; \mathbb{C} \vdash \mathbb{E}[\![\tau]\!]_{\Xi\cdot\Xi'}(v_1 u, v_2 u)\end{array}}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{V}[\![\sigma \to \tau]\!]_\Xi(v_1, v_2)} \qquad \frac{\begin{array}{c}\text{for all } (u, \Xi', \Delta') \in \mathbf{SVal}(\sigma) \\ (\Delta \cdot \Delta')_\alpha; \mathbb{C} \vdash \mathbb{E}[\![\tau]\!]_{\Xi\cdot\Xi'}(K_1[u], K_2[u])\end{array}}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{K}[\![\sigma, \tau]\!]_\Xi(K_1, K_2)}\ \mathrm{K}$$

$$\mathrm{E}\ \frac{\begin{array}{l}\text{for all } (M_1', \Delta_1, \mathbb{H}_1^{\mathrm{pre}}, \mathbb{H}_1^{\mathrm{post}}, \mathbb{C}_1) \in \Downarrow_{\mathrm{s}}(M_1, \Delta), \\ \text{for all } (M_2', \Delta_2, \mathbb{H}_2^{\mathrm{pre}}, \mathbb{H}_2^{\mathrm{post}}, \mathbb{C}_2) \in \Downarrow_{\mathrm{s}}(M_2, \Delta), \\ (\Delta_1 \cup \Delta_2)_{\alpha'}; \mathbb{C} \cup \mathbb{C}_1 \cup \mathbb{C}_2 \vdash \mathbb{S}^{\mathbf{x}}[\![\tau]\!]_\Xi(M_1', M_2')\end{array}}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{E}[\![\tau]\!]_\Xi(M_1, M_2)} \qquad \frac{\begin{array}{c}\text{for all } (\mu, \Delta, \Xi) \in \mathrm{SSubst}(\Gamma) \\ \Delta; \varnothing \vdash \mathbb{E}[\![\tau]\!]_\Xi(M_1\{\mu\}, M_2\{\mu\})\end{array}}{\varepsilon; \varnothing \vdash M_1 \simeq^{\Gamma}_{\mathrm{skor}} M_2 : \tau}\ \textsc{Init}$$

when $\Delta_1 \cap \Delta_2 = \Delta$, with $\mathbf{x} = \mathrm{select}_\Xi(M_1', M_2')$ and $\alpha' = \alpha \cdot \{(y, i) \mid i \in \{1, 2\} \wedge y \in \mathrm{dom}(\mathbb{H}_i^{\mathrm{post}}) \backslash \mathrm{dom}(\mathbb{H}_i^{\mathrm{pre}})\}$.

$\mathrm{select}_\Xi(K_1[f_1 v_1], K_2[f_2 v_2]) \triangleq \mathrm{PQ}$ when $f_1 = f_2 \in \mathrm{dom}(\Xi)$
$\mathrm{select}_\Xi(v_1, v_2) \triangleq \mathrm{PA}$ $\quad$ $\mathrm{select}_\Xi(M_1, M_2) \triangleq \lightning$ otherwise.

Fig. 5. Definition of the inference rules for reasoning on SKORs

- $\mathrm{V}_\to$ is used to synchronize two functional values, by providing them a symbolic value coming from the context, then trying to relate the produced terms. K follows the same pattern, but for the return point of callbacks.
- E tries to relate any pair (M_1', M_2') formed by the result of the symbolic execution of respectively M_1 and M_2. For any of these pairs, it uses the function select to decide which synchronization point should be applied, using the symbol $\lightning$ for a failed synchronization. If there are m_1 elements in $\Downarrow_{\mathrm{s}}(M_1, \Delta_1)$ and m_2 in $\Downarrow_{\mathrm{s}}(M_2, \Delta_2)$ (up to the renaming of variables), we then have to deal with the $m_1 \times m_2$ combinations. For each of these premises (represented by a node n_i, for $i \in \{1, \ldots, m_1 \times m_2\}$), we then put in $\mathrm{Annot}(\mathrm{n}_i)$ the information associated to the given symbolic reduction, i.e. the result $\mathbf{x}_i$ of select and the four symbolic heaps $\mathbf{H} = \mathbb{H}_1^{\mathrm{pre}}, \mathbb{H}_2^{\mathrm{pre}}, \mathbb{H}_1^{\mathrm{post}}, \mathbb{H}_2^{\mathrm{post}}$.
- Init is the root of the derivation tree and initiates the reasoning by picking a symbolic substitution μ to transform the typing context Γ into a ground typing context Δ and a functional typing context Ξ, while eliminating variables of type Unit, Bool and $\tau \times \sigma$ by applying the substitution μ on the given terms.

If a derivation tree does not use the rule $\mathrm{S}^{\lightning}$, and in all the uses of the rule V_ι whose sequent is $\Delta_\alpha; \mathbb{C} \vdash \mathbb{V}[\![\iota]\!]_\Xi(v_1, v_2)$, for all $\eta \in \mathrm{GVal}(\Delta, \mathbb{C})$, one has $\models_\eta v_1 = v_2$, then this derivation tree can indeed be seen as a proof of contextual equivalence. However, in general the two terms initially considered can be contextually equivalent even if the built derivation tree does not satisfy this condition. Indeed, v_1 or v_2 could be variables corresponding to what is stored in the heap at a given location. But since in the rule E, we consider all the possible executions of the two terms, without any hypothesis on what is stored in the heaps when starting their execution, some of the branch of the derivation tree will not correspond to real interaction between the two terms and a context.

$$
\textsc{Init}\ \dfrac{\mathrm{E}\ \dfrac{\mathrm{PA}(\mathbf{H}_1) \qquad \Delta_1;\varnothing \vdash \mathbb{V}\llbracket\tau\rrbracket(\langle v_1^{\mathtt{i}}, v_1^{\mathtt{g}}\rangle, \langle v_2^{\mathtt{i}}, v_2^{\mathtt{g}}\rangle)}{\varepsilon;\varnothing \vdash \mathbb{E}\llbracket\tau\rrbracket(M_1^{\mathtt{ri}}, M_2^{\mathtt{ri}})}}{\varepsilon;\varnothing \vdash M_1^{\mathtt{ri}} \simeq_{\mathrm{skor}} M_2^{\mathtt{ri}} : \tau}
$$

where the premise $\Delta_1;\varnothing \vdash \mathbb{V}\llbracket\tau\rrbracket(\langle v_1^{\mathtt{i}}, v_1^{\mathtt{g}}\rangle, \langle v_2^{\mathtt{i}}, v_2^{\mathtt{g}}\rangle)$ is obtained by

$$
V_\times\ \dfrac{V_\to\ \dfrac{\mathrm{E}\ \dfrac{\mathrm{PA}(\mathbf{H}_2) \qquad V_\iota\ \dfrac{}{\Delta_2;\mathbb{C} \vdash \mathbb{V}\llbracket\mathrm{Unit}\rrbracket((),())}}{\Delta_2;\varnothing \vdash \mathbb{E}\llbracket\mathrm{Unit}\rrbracket(v_1^{\mathtt{i}}(), v_2^{\mathtt{i}}())}}{\Delta_1;\varnothing \vdash \mathbb{V}\llbracket\mathrm{Unit}\to\mathrm{Unit}\rrbracket(v_1^{\mathtt{i}}, v_2^{\mathtt{i}})} \qquad \dfrac{\dfrac{\mathrm{PA}(\mathbf{H}_3) \qquad \dfrac{}{\Delta_3; z = -y_2 \vdash \mathbb{V}\llbracket\mathrm{Int}\rrbracket(y_1, z)}\ V_\iota}{\Delta_1;\varnothing \vdash \mathbb{E}\llbracket\mathrm{Int}\rrbracket(v_1^{\mathtt{g}}(), v_2^{\mathtt{g}}())}\ \mathrm{E}}{\Delta_1;\varnothing \vdash \mathbb{V}\llbracket\mathrm{Unit}\to\mathrm{Int}\rrbracket(v_1^{\mathtt{g}}, v_2^{\mathtt{g}})}\ V_\to}{\Delta_1;\varnothing \vdash \mathbb{V}\llbracket\tau\rrbracket(\langle v_1^{\mathtt{i}}, v_1^{\mathtt{g}}\rangle, \langle v_2^{\mathtt{i}}, v_2^{\mathtt{g}}\rangle)}
$$

$$
\begin{array}{lll}
 & \tau = (\mathrm{Unit}\to\mathrm{Unit})\times(\mathrm{Unit}\to\mathrm{Int}) & \mathbb{C} = \{z_1 = y_1 + 1, z_2 = y_2 - 1\} \\
 & \multicolumn{2}{l}{v_1^{\mathtt{i}} = \lambda\_.\mathtt{c}_1 := !\mathtt{c}_1 + 1,\ v_2^{\mathtt{i}} = \lambda\_.\mathtt{c}_1 := !\mathtt{c}_2 - 1,\ v_1^{\mathtt{g}} = \lambda\_.!\mathtt{c}_1,\ v_2^{\mathtt{g}} = \lambda\_.-!\mathtt{c}_2} \\
\text{with} & \Delta_1 = [c_1, c_2 \mapsto \mathrm{ref\ Int}] & \mathbf{H}_1 = (\varepsilon, \varepsilon, [c_1 \mapsto 0], [c_2 \mapsto 0]) \\
 & \Delta_2 = \Delta_1 \cdot [y_1, y_2, z_1, z_2 \mapsto \mathrm{Int}] & \mathbf{H}_2 = ([c_1 \mapsto y_1], [c_2 \mapsto y_2], [c_1 \mapsto z_1], [c_1 \mapsto z_2]) \\
 & \Delta_3 = \Delta_1 \cdot [y_1, y_2, z \mapsto \mathrm{Int}] & \mathbf{H}_3 = ([c_1 \mapsto y_1], [c_2 \mapsto y_2], [c_1 \mapsto y_1], [c_1 \mapsto y_2])
\end{array}
$$

Fig. 6. Representation Independence Example

One could imagine using in the rule E the heaps we obtain in the previous use of the E. However, doing this we would miss many possible interactions, so that this would not be sound.

6.5 Some Examples

We now give in Figure 6 the derivation tree for the Representation Independence example of Section 2.1, and in Figure 7 the derivation tree for the Well-Bracketed State Change (WBSC) example of Section 2.2. We have omitted the rules S^{PA}, using $\mathbb{S}^{\mathrm{PA}}\llbracket\tau\rrbracket$ as a shortcut for $\mathbb{V}\llbracket\tau\rrbracket$. For the WBSC example, we have also omitted in the premises of the rule S^{PQ} the tree corresponding to the equivalence $\mathbb{V}\llbracket\mathrm{Unit}\rrbracket_{\Xi}((),())$ (which is directly concluded using the rule V_ι):

In the derivation tree of Figure 6, in the rule V_ι that concludes the sequent $\Delta_3; z = -y_2 \vdash \mathbb{V}\llbracket\mathrm{Int}\rrbracket(y_1, z)$, one can see that we cannot deduce $y_1 = z$ from $z = -y_2$, so that the derivation tree cannot be seen as a full proof of a contextual equivalence of $M_1^{\mathtt{ri}}, M_2^{\mathtt{ri}}$. The same is true for proving $y_3 = 1$ in the rule V_ι of Figure 7.

6.6 Automatic Construction of Derivation Trees

To build a derivation tree, we have to use a SMT-solver to check if the logical context $\mathbb{C}$ is satisfiable, to decide if we can use the rule Unsat. Since our language has multiplication, satisfiability is in general undecidable. So we must allow the solver to fail to check if $\mathbb{C}$ is satisfiable, in which case we do not use the rule Unsat and continue the construction.

Doing so, we are always able to build a derivation tree whose root is $\vdash M_1 \simeq_{\mathrm{skor}}^{\Gamma} M_2 : \tau$, as soon as M_1, M_2 are of the same type. This construction is deterministic up-to the results returned by the SMT-solver we use.

7 STRUCTURED MEMORY TRANSITION MACHINES

We now introduce the central objects of our work, namely ***Structured Memory Transition Machines*** (SMTM). There definition is given in Section 7.1, and the ***execution relation*** $c \to_{\mathbb{M}} c'$ between ***memory configurations***, derived from such a SMTM $\mathbb{M}$, is given in Section 7.2. This execution relation is used to define the notion of ***runs*** of $\mathbb{M}$. Finally, in Section 7.3, we show how to build a SMTM $\mathbb{M}_{\mathcal{S}}$ from a derivation tree $\mathcal{S}$.

$\mathbf{H}_4 = ([x \mapsto y_3], \varepsilon, [x \mapsto y_3], \varepsilon)$
$\Delta_4 = \Delta_3 \cdot [y_3 \mapsto \text{Int}]$

$K_1' = \bullet; !\mathsf{x}$ and $K_2' = \bullet; 1$
$\mathbf{H}_3 = ([x \mapsto y_2], \varepsilon, [x \mapsto 1], \varepsilon)$
$\Delta_3 = \Delta_2 \cdot [y_2 \mapsto \text{Int}]$

$K_1 = \bullet; \mathsf{x} := 1; \mathsf{f}(); !\mathsf{x}$ and $K_2 = \bullet; \mathsf{f}(); 1$
$\mathbf{H}_2 = ([x \mapsto y_1], \varepsilon, [x \mapsto 0], \varepsilon)$
$\Delta_2 = \Delta_1 \cdot [y_1 \mapsto \text{Int}]$

$\mathbf{H}_1 = (\varepsilon, \varepsilon, [x \mapsto 0], \varepsilon)$
$\Delta_1 = [x \mapsto \text{ref Int}]$
$\Xi = [f \mapsto (\text{Unit} \to \text{Unit})]$
$v_1^{\mathsf{wbsc}} \triangleq \lambda f.x := 0; f(); x := 1; f(); !x$
$\tau = (\text{Unit} \to \text{Unit}) \to \text{Unit}$

$$\dfrac{\mathrm{PA}(\mathbf{H}_1) \quad \dfrac{\dfrac{\mathrm{PQ}(\mathbf{H}_2) \quad \dfrac{\dfrac{\mathrm{PQ}(\mathbf{H}_3) \quad \dfrac{\dfrac{\mathrm{PA}(\mathbf{H}_4) \quad \dfrac{}{\Delta_4; \varnothing \vdash \mathbb{V}[\![\text{Int}]\!]_\Xi(y_3, 1)}\,\mathrm{V}_\iota}{\Delta_3; \varnothing \vdash \mathbb{E}[\![\text{Int}]\!]_\Xi(K_1'[()], K_2'[()])}\,\mathrm{E}}{\Delta_3; \varnothing \vdash \mathbb{K}[\![\text{Unit}, \text{Int}]\!]_\Xi(K_1', K_2')}\,\mathrm{K}}{\Delta_3; \varnothing \vdash \mathbb{S}^{\mathrm{PQ}}[\![\text{Int}]\!]_\Xi(K_1'[f()], K_2'[f()])}\,\mathrm{S}^{\mathrm{PQ}}}{\Delta_2; \varnothing \vdash \mathbb{E}[\![\text{Int}]\!]_\Xi(K_1[()], K_2[()])}\,\mathrm{E}}{\Delta_2; \varnothing \vdash \mathbb{K}[\![\text{Unit}, \text{Int}]\!]_\Xi(K_1, K_2)}\,\mathrm{K}}{\Delta_2; \varnothing \vdash \mathbb{S}^{\mathrm{PQ}}[\![\text{Int}]\!]_\Xi(K_1[f()], K_2[f()])}\,\mathrm{S}^{\mathrm{PQ}}}{\Delta_1; \varnothing \vdash \mathbb{E}[\![\text{Int}]\!]_\Xi(v_1^{\mathsf{wbsc}} f, M_2^{\mathsf{wbsc}} f)}\,\mathrm{E}}{\Delta_1; \varnothing \vdash \mathbb{V}[\![\tau]\!]_\Xi(v_1^{\mathsf{wbsc}}, M_2^{\mathsf{wbsc}})}\,\mathrm{V}_\to}{\varepsilon; \varnothing \vdash \mathbb{E}[\![\tau]\!](M_1^{\mathsf{wbsc}}, M_2^{\mathsf{wbsc}})}\,\mathrm{E}}{\varepsilon; \varnothing \vdash M_1^{\mathsf{wbsc}} \simeq_{\mathrm{skor}} M_2^{\mathsf{wbsc}} : \tau}\,\textsc{Init}$$

Fig. 7. Well-Bracketed State Change Example

7.1 Definition of SMTM

A Structured Memory Transition Machines $\mathbb{M}$ is defined as a tuple $(A, a_0, A^{\lightning}, d_P, d_O, d_{\mathrm{O}\epsilon})$, where:

- A is a set of ***atomic states***, and $a_0 \in A$ is the initial atomic state;
- $A^{\lightning} \subseteq A$ is the set of ***failed atomic states***.
- $d_P : A \times A \rightharpoonup (\mathrm{SHeap}^4 \times \mathrm{GContext}^2 \times \mathcal{P}(\mathrm{AForms}) \times \{\mathrm{PA}, \mathrm{PQ}\})$ is the Player transition function. It maps pairs of atomic states to a tuple $(\mathbb{H}_1^{\mathrm{pre}}, \mathbb{H}_2^{\mathrm{pre}}, \mathbb{H}_1^{\mathrm{post}}, \mathbb{H}_2^{\mathrm{post}}, \mathbb{C}, \Delta_L, \Delta_S, \mathbf{x})$ formed by:
  - A quadruple of symbolic heaps $\mathbb{H}_1^{\mathrm{pre}}, \mathbb{H}_2^{\mathrm{pre}}, \mathbb{H}_1^{\mathrm{post}}, \mathbb{H}_2^{\mathrm{post}}$ and a set of arithmetic formulas $\mathbb{C}$;
  - Two ground typing contexts $\Delta_L, \Delta_S \in \mathrm{GContext}$ to distinguish between the variables that are used only locally to enforce constraints on the heaps, and the variables that can "leak" and be used in other transitions.
  - $\mathbf{x} \in \{\mathrm{PA}, \mathrm{PQ}\}$.
- $d_O : A \times A \rightharpoonup \mathrm{GContext} \times \{\mathrm{OA}, \mathrm{OQ}\}$ is the Opponent transition function. It maps pairs of atomic states to a pair $(\Delta, \mathbf{x})$, where Δ is a ground typing context corresponding to the fresh variables introduced by Opponent to represent the symbolic values it provides to Player.
- $d_{\mathrm{O}\epsilon} : A \times A$ is the Oϵ-transition function.

The SMTM that we will build are bipartite, with ***Player states*** being the sources of Player transitions and the targets of Opponent transitions, and ***Opponent states*** being the sources of Opponent transitions and the targets of Player transitions.

7.2 Runs of SMTM

We now describe how to deduce from a SMTM $\mathbb{M}$ a relation $c \to_{\mathbb{M}} c'$ between ***memory configurations*** c, c', that are used to represent the runs of SMTM. A memory configuration c is a tuple $(a, \eta, \sigma, \kappa, h_1, h_2)$, where:

- a is an atomic state of $\mathbb{M}$;
- $\eta \in \mathrm{GVal}$ is a ground valuation, which is used to interpret variables appearing in the symbolic representation of transitions of $\mathbb{M}$;

- σ is a stack (represented as a list) of pairs (a', η') of atomic states and ground valuation. It is used to enforce the well-bracketed properties of the interaction, and to restore the right valuation after a nested call;
- κ is the ***valuation knowledge***, a partial function from Opponent states to sets of ground valuations. It represents the current knowledge that Opponent has about Player closures.
- h_1, h_2 are two heaps.

The ***initial configuration*** c_0 of a SMTM $\mathbb{M}$ is the tuple $(a_0, \varepsilon, [], \{a_0, \varepsilon\})$ with a_0 the initial atomic state of $\mathbb{M}$. A configuration whose atomic state is in $A^{\lightning}$ is also said to be ***failed***.

In order to define $c \longrightarrow c'$ from Player transitions in d_P, one first need to transform the symbolic constraints such transitions specifies into a predicate over tuples $(h_1, h_2, h_1', h_2', \eta)$ formed by four heaps and a ground valuation. So we define $(h_1, h_2, h_1', h_2', \eta) \models (\mathbb{H}_1^{\text{pre}}, \mathbb{H}_2^{\text{pre}}, \mathbb{H}_1^{\text{post}}, \mathbb{H}_2^{\text{post}}, \Delta_L, \mathbb{C})$ as

$$\exists \eta_L \in \text{GVal}(\Delta_L), \models_{(\eta \cdot \eta_L)} \mathbb{C} \wedge \forall i \in \{1,2\}, \exists h_i^f, (\uparrow^{(\eta \cdot \eta_L)}(\mathbb{H}_i^{\text{pre}}) \cdot h_i^f) = h_i \wedge (\uparrow^{(\eta \cdot \eta_L)}(\mathbb{H}_i^{\text{post}}) \cdot h_i^f) = h_i'$$

Notice that the symbolic heaps given by d_P does not constraint the whole heaps that the configuration carries, but only the relevant part for this transition. To deal with this problem, we enforce a framing property, by picking two heaps h_1^f, h_2^f, that have to stay the same between the source and the target configuration of the execution relation of configurations.

Definition 7.1. Taking a SMTM $\mathbb{M}$ and $\mathbf{x} \in \{\text{PA}, \text{PQ}, \text{OQ}, \text{OA}, \text{O}\epsilon\}$, we define the ***execution relation*** $\xrightarrow{\mathbf{x}}_{\mathbb{M}}$ on configurations in the following way:

- If $d_P(a, a') = (\mathbb{H}_1^{\text{pre}}, \mathbb{H}_2^{\text{pre}}, \mathbb{H}_1^{\text{post}}, \mathbb{H}_2^{\text{post}}, \Delta_S, \Delta_L, \mathbb{C}, \mathbf{x})$ with $\mathbf{x} \in \{\text{PA}, \text{PQ}\}$, then one has:
  - $(a, \eta, \sigma, \kappa, h_1, h_2) \xrightarrow{\text{PA}}_{\mathbb{M}} (a', \eta \cdot \eta', \sigma, \kappa \cup \{(a', \eta'), h_1', h_2'\})$ if $\mathbf{x} = \text{PA}$;
  - $(a, \eta, \sigma, \kappa, h_1, h_2) \xrightarrow{\text{PQ}}_{\mathbb{M}} (a', \eta \cdot \eta', (a', \eta \cdot \eta') :: \sigma, \kappa, h_1', h_2')$ if $\mathbf{x} = \text{PQ}$

  if in both cases $\eta' \in \text{GVal}(\Delta_S)$ and $(h_1, h_2, h_1', h_2', \eta \cdot \eta') \models (\mathbb{H}_1^{\text{pre}}, \mathbb{H}_2^{\text{pre}}, \mathbb{H}_1^{\text{post}}, \mathbb{H}_2^{\text{post}}, \Delta_L, \mathbb{C})$.
- If $(a, a') \in d_{\text{O}\epsilon}$ then one has $(a, \eta, \sigma, \kappa, h_1, h_2) \xrightarrow{\text{O}\epsilon}_{\mathbb{M}} (a', \eta, \sigma, \kappa, h_1, h_2)$.
- If $d_O(a, a') = (\Delta_S, \mathbf{x})$ with $\mathbf{x} \in \{\text{OA}, \text{OQ}\}$, then one has:
  - $(a, \eta, \sigma, \kappa, h_1, h_2) \xrightarrow{\text{OQ}}_{\mathbb{M}} (a', \eta' \cdot \eta'', \sigma, \kappa, h_1, h_2)$ if $\mathbf{x} = \text{OA}$, $\eta' \in \kappa(a)$ and $\eta'' \in \text{GVal}(\Delta)$.
  - $(a, \eta, (a', \eta') :: \sigma, \kappa), h_1, h_2) \xrightarrow{\text{OA}}_{\mathbb{M}} (a', \eta', \sigma, \kappa), h_1, h_2)$ if $\mathbf{x} = \text{OA}$.

A ***run*** of a $\mathbb{M}$ is then a sequence of executions $c_1 \xrightarrow{\mathbf{x}_1}_{\mathbb{M}} c_2 \ldots \xrightarrow{\mathbf{x}_n}_{\mathbb{M}} c_{n+1}$. $\mathbb{M}$ is said to be safe when there is no run starting from the initial configuration that reaches an inconsistant configuration.

When it is clear about which SMTM $\mathbb{M}$ we are talking about, we write $\xrightarrow{\mathbf{x}}$ for $\xrightarrow{\mathbf{x}}_{\mathbb{M}}$.

To define $\xrightarrow{\text{PA}}$ and $\xrightarrow{\text{PQ}}$ from d_P, one need to take a new valuation η' for the new variables in Δ_S that will be used in the other transitions (i.e. for the leaked variables). The variables in Δ_L, that are local to this transition, are handled by the definition of $(h_1, h_2, h_1', h_2', \eta) \models (\mathbb{H}_1^{\text{pre}}, \mathbb{H}_2^{\text{pre}}, \mathbb{H}_1^{\text{post}}, \mathbb{H}_2^{\text{post}}, \Delta_L, \mathbb{C})$, so that the ground valuation corresponding to them does not have to be added to the new environment of the resulting configuration.

The evolution of the stack σ and the valuation knowledge κ of a configuration is dictated by the polarity of the transition:

PQ: we push on top of the stack the valuation and atomic state of the target configuration;
PA: we add the valuation and atomic state of the target configuration to κ;
OQ: we pick a valuation in κ, which becomes the new valuation of the target configuration;
OA: we pop from the top of the stack a valuation and an atomic state that become the ones of the target configuration.

7.3 Construction of the SMTM

The algorithm that builds a SMTM $\mathbb{M}_{\mathcal{S}}$ from a derivation tree $\mathcal{S}$ works in two steps. The first step transforms $\mathcal{S}$ into a transition system, by merging some nodes of $\mathcal{S}$ into one state, and by extracting arithmetic and heap constraints from the rules $\mathrm{E}, \mathrm{V}_\iota$. This transition system will thus also be a tree, its runs corresponding to interactions with simple contexts that can interrogate Player higher-order values only once, immediately when these values are provided.

The second step performs a closure by adding Oϵ-transitions to complete the behaviour of Opponent, allowing contexts to interrogate Player higher-order values multiple times, possibly reentrantly.

7.3.1 From Derivation Trees to Skeletal Representations. The algorithm manipulates ***skeletal representation*** $(\mathbb{M}, \mathbb{C}^f, \mathcal{F})$, where:

- $\mathbb{M}$ is a SMTM with $d_{\mathrm{O}\epsilon}$ left empty;
- $\mathbb{C}^f$ is a set of arithmetic formulas, that will have to be satisfied in the future, and whose negation will be used to generate transitions to failed states;
- $\mathcal{F} : \mathbb{M}.S \to \mathcal{P}(\mathcal{S}.\mathcal{N})$ is a function mapping each atomic state of $\mathbb{M}$ to a set of nodes of $\mathcal{S}$. This function partitions $\mathcal{S}$ into subtrees.

The skeletal representation $\mathrm{Skel}(\mathcal{S}, \mathrm{n})$ associated to a derivation tree $\mathcal{S}$ is constructed by induction on the tree substructure of $\mathcal{S}$. We write:

- (N, L) for $\mathrm{Edge}(\mathrm{n})$, with $N = \{\mathrm{n}_1, \dots, \mathrm{n}_m\}$, and $\Delta_\alpha ; \mathbb{C} \vdash \phi$ for $\mathrm{Sequent}(\mathrm{n})$
- $(\Delta_i)_{\alpha_i} ; \mathbb{C}_i \vdash \phi_i$ for $\mathrm{Sequent}(\mathrm{n}_i)$, Δ'_i for $\Delta_i \backslash \Delta$ and $\mathbb{C}'_i$ for $\mathbb{C}_i \backslash \mathbb{C}$;
- $(\mathbb{M}_i, \mathbb{C}^f_i, \mathcal{F}_i)$ for $\mathrm{Skel}(\mathcal{S}, \mathrm{n}_i)$ with the initial atomic state of $\mathbb{M}_i$ written a^i_0, and we suppose that the set of atomic states of each of the $\mathbb{M}_i$ are disjoint.

Then, the construction is done by case analysis on L:

- If $\mathrm{L} = \textsc{Unsat}$ or $\mathrm{L} = \mathrm{V}_\iota$ with $\phi = \mathbb{V}[\![\iota]\!]_\Xi(u_1, u_2)$, taking a_0 a fresh atomic state, we return $(\mathbb{M}, \{P\}, [a_0 \mapsto \{\mathrm{n}\}])$, where $\mathbb{M}$ is the SMTM with no transition and a unique atomic state a_0 and
$$P \triangleq \begin{cases} \text{True when } \mathrm{L} = \mathrm{V}_\iota \text{ and } \iota \in \{\mathrm{Unit}, \mathrm{Bool}\} \text{ with } u_1 = u_2 \\ \text{False when } \mathrm{L} = \textsc{Unsat}, \text{ or } \mathrm{L} = \mathrm{V}_\iota \text{ and } \iota = \mathrm{Bool} \text{ with } u_1 \neq u_2 \\ u_1 = u_2 \text{ when } \mathrm{L} = \mathrm{V}_\iota \text{ and } \iota = \mathrm{Int} \end{cases}$$
- If $\mathrm{L} \in \{\mathrm{S}^{\mathrm{PQ}}, \mathrm{V}_\times\}$, we return the ***pointed union*** of $(\mathrm{Skel}(\mathcal{S}_i))_{i \in \{1,2\}}$, constructed by taking a fresh atomic state a_0, substituting the former initial atomic states a^i_0 by a_0 everywhere, then taking their union piecewise, and adding n to the set $\mathcal{F}(a_0)$.
- If $\mathrm{L} \in \{\textsc{Init}, \mathrm{V}_\to, \mathrm{K}\}$ Then we write $\mathbb{M}'$ and $\mathcal{F}'$ for the union respectively of the $\mathbb{M}_i$ and the $\mathcal{F}_i$, We write $\mathbb{M}'$ as $(S', \_, A'^{\natural}, d'_P, d'_O)$. Taking a_0 a fresh atomic state, we then return $(\mathbb{M}, \varnothing, \mathcal{F}' \cdot [a_0 \mapsto \{\mathrm{n}\}])$, where $\mathbb{M}$ is defined as $(S' \cup \{a_0\}, a_0, A'^{\natural}, d'_P, d_O)$ with:
$$d_O \triangleq \begin{cases} d'_O \cup \{((a_0, a^i_0), (\Delta'_i, \mathrm{OQ}) \mid i \in \{1, \dots, m\}\} \text{ if L is } \textsc{Init} \text{ or } \mathrm{V}_\to \\ d'_O \cup \{((a_0, a^i_0), (\Delta'_i, \mathrm{OA}) \mid i \in \{1, \dots, m\}\} \text{ if L is K} \end{cases}$$
- if L=E with $\mathcal{S}.\mathrm{Annot}(\mathrm{n}_i) = \mathbf{x}_i(\mathbf{H}_i)$. Then we write $\mathbb{M}'$ and $\mathcal{F}'$ for the union respectively of the $\mathbb{M}_i$ and the $\mathcal{F}_i$, We write $\mathbb{M}'$ as $(S', \_, A'^{\natural}, d'_P, d'_O)$. Taking a_0 a fresh atomic state, we then return $(\mathbb{M}, \varnothing, \mathcal{F}' \cdot [a_0 \mapsto \{\mathrm{n}\}])$, where $\mathbb{M}$ is defined as $(S, a_0, A^{\natural}, d_P, d'_O)$ with:
$$S \triangleq S' \cup \{a_0\} \cup \{a^{\natural}_i \mid i \in \{1, \dots, m\}\} \qquad A^{\natural} \triangleq A'^{\natural} \cup \{a^{\natural}_i \mid i \in \{1, \dots, m\}\}$$
$$d_P \triangleq d'_P \cup \{((a_0, a^i_0), (\mathbf{H}_i, \Delta_i, \varnothing, \mathbb{C}'_i \wedge \mathbb{C}^f_i, \mathbf{x}_i)) \mid i \in \{1, \dots, m\}\}$$
$$\cup \{((a_0, a^{\natural}_i), (\mathbf{H}_i, \mathbb{C}'_i \wedge \neg \mathbb{C}^f_i, \mathbf{x}_i)) \mid i \in \{1, \dots, m\}\}$$

We then perform a post-treatment on the Player-transitions, modifying the ground typing contexts Δ_L, Δ_S of each of these transitions so that Δ_L contains all the variables that are used only in the considered transition, and Δ_S the variables that can leak.

Player states correspond to rules E, while Opponent states correspond to rules K or $\mathrm{V}_{\rightarrow}$.

7.3.2 Closure of the SMTM. So far, after each interaction point, the SMTM we have built allows Opponent to only interrogate the higher-order values that has just been provided at this point. However, Opponent should also be allowed to interrogate higher-order values that have been provided before. To do so, we will add Oϵ-transitions to go back to states where these values have been provided by Player. But when the interaction points from where these new Oϵ-transitions come from are Player questions, we also have to add the corresponding Opponent answers necessary, again using Oϵ-transitions.

To do so, we introduce a notion of ***external*** and ***well-bracketed*** transition associated to a SMTM $\mathbb{M} = (A, a_0, A^{\natural}, d_P, d_O, d_{\mathrm{O}\epsilon})$, defined as:

- $d_{\mathbf{e}} \triangleq \mathrm{dom}(d_P) \circ \mathrm{dom}(d_O) \circ d^{=}_{\mathrm{O}\epsilon}$,
- $d_{\mathbf{wb}} \triangleq \mathrm{dom}(d_{\mathrm{PA}}) \circ (\mathrm{dom}(d_{\mathrm{OA}}) \circ d^{=}_{\mathrm{O}\epsilon} \circ \mathrm{dom}(d_{\mathrm{PQ}}))^* \circ \mathrm{dom}(d_{\mathrm{OQ}}) \circ d^{=}_{\mathrm{O}\epsilon}$ where we write $d_{\mathbf{x}}$ (with $\mathbf{x} \in \{\mathrm{PQ}, \mathrm{PA}, \mathrm{OQ}, \mathrm{OA}\}$) for the set of transitions labelled with $\mathbf{x}$.

We can now define the second step, defined as a closure of a skeletal representation.

Definition 7.2. The closure $\bar{\mathbb{M}}$ of a SMTM $\mathbb{M} = (A, a_0, A^{\natural}, d_P, d_O, d_{\mathrm{O}\epsilon})$ is defined as the SMTM $(A, a_0, A^{\natural}, d_P, d_O, \bar{d}_{\mathrm{O}\epsilon})$ with $\bar{d}_{\mathrm{O}\epsilon}$ the smallest relation on atomic states containing $d_{\mathrm{O}\epsilon}$ and s.t.

1. for all atomic states $a_1 \in \pi_1(\mathrm{dom}(d_{\mathrm{OQ}}))$, for all atomic states a_2 s.t. $(a_1, a_2) \in \mathrm{dom}(\bar{d}^{+}_{\mathbf{e}})$, we have $(a_2, a_1) \in \bar{d}_{\mathrm{O}\epsilon}(a_2)$.
2. for all atomic states $a_1 \in \pi_1(\mathrm{dom}(d_{\mathrm{OA}}))$, for all atomic states a_2 s.t. $(a_1, a_2) \in \mathrm{dom}(\bar{d}^{+}_{\mathbf{wb}})$, we have $(a_2, a_1) \in \bar{d}_{\mathrm{O}\epsilon}$.

This closure only adds new Oϵ-transitions to $\mathbb{M}$. We write $\mathbb{M}_{\mathcal{S}}$ for the closure of $\mathbb{M}$ where $(\mathbb{M}, \_, \_) = \mathrm{Skel}(\mathcal{S}, \mathcal{S}.\mathrm{r})$.

The first step of this closure allow Opponent to interrogate higher-order values previously provided as soon as it has the control (i.e. when we are in an Opponent state), while the second step allows Opponent to answer after such interrogations, to get well-bracketed interactions.

8 SYTECI FOR PROGRAMS WITH RECURSION

To prove contextual equivalence for programs with recursion, we introduce a technique to ***synchronize recursive calls***. To do so, we consider recursive calls as ***internal interaction points***, proper to the program. The first step is thus to capture these recursive calls in the operational semantics and in the symbolic evaluation (Section 8.1). For that purpose, we introduce ***extended terms*** (M, γ), such that M can have free functional variable that are defined via the functional environment γ. We then extend symbolic Kripke open relations to deal with such extended terms in Section 8.2. The key points are to introduce rules to allow the unfolding of recursive calls, and to generalize derivation trees into ***derivation graphs***, in order to perform ***circular reasoning***. We then give the derivation graph associated to the stateful factorial example (Section 8.3), and provide some heuristics to build automatically derivation graphs (Section 8.4). Finally, in Section 8.5, we extend the construction of the state transition systems from derivation trees to derivation graphs.

8.1 Refining Operational Semantics and Symbolic Evaluation

We refine the operational semantics given in Section 5.2, to control the unfolding of fixed-points. To do so, we define the operational reduction $\xrightarrow{c}$ that blocks on recursive calls. It reduces ***extended***

terms (M, γ), formed by a term M and a ***functional environments*** $\gamma : \mathrm{Var} \rightharpoonup \mathrm{Val}$ mapping variables to higher-order values:

$$(K[(\mathbf{fix}\, y(x) \to M)v], \gamma, h) \xrightarrow{c} (K[M\{x := v\}], \gamma \cdot [y \mapsto (\mathbf{fun}\, x \to M)], h) \qquad \frac{(M, h) \xrightarrow{r} (M', h')}{(M, \gamma, h) \xrightarrow{c} (M', \gamma, h')}$$

We then modify the symbolic evaluation to work on extended terms, by replacing the first reduction rule by: $(M, \gamma, \Delta, \mathbb{H}^{\mathrm{pre}}, \mathbb{H}^{\mathrm{post}}, \mathbb{C}) \mapsto_s (M', \gamma', \Delta, \mathbb{H}^{\mathrm{pre}}, \mathbb{H}^{\mathrm{post}}, \mathbb{C})$ when $(M, \gamma, \varepsilon) \xrightarrow{c} (M', \gamma', \varepsilon)$. The other rules, that leave the functional environment γ unchanged, are straightforwardly adapted, and can be found in the Appendix C.

8.2 Derivation Graph

To reason about recursive terms, we extend the Symbolic Kripke Open Relations to reason on extended terms. To do so, we provide new rules to unfold recursive calls. We thus add a new tag PI, corresponding to a Player Internal interaction point, together with an auxiliary SKOR $\mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((M_1, \gamma_1), (K_2[f_2\ v_2], \gamma_2))$, whose new rule is:

$$\textsc{Unf}\ \frac{\Delta_\alpha; \mathbb{C} \vdash \mathbb{E}[\![\tau]\!]_\Xi((K_1[\gamma_1(f_1)\ v_1], \gamma_1), (K_2[\gamma_2(f_2)\ v_2], \gamma_2))}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((K_1[f_1\ v_1], \gamma_1), (K_2[f_2\ v_2], \gamma_2))}$$

We also extend the select function used in the rule E to work on extended terms:

$$\begin{array}{l}
\mathrm{select}_\Xi((v_1, \gamma_1), (v_2, \gamma_2)) \triangleq \mathrm{PA} \\
\mathrm{select}_\Xi((K_1[f_1 v_1], \gamma_1), (M_2, \gamma_2)) \triangleq \mathrm{PI} \text{ when } f_1 \in \mathrm{dom}(\gamma_1) \\
\mathrm{select}_\Xi((M_1, \gamma_1), (K_2[f_2 v_2], \gamma_2)) \triangleq \mathrm{PI} \text{ when } f_2 \in \mathrm{dom}(\gamma_2) \\
\mathrm{select}_\Xi((K_1[f_1 v_1], \gamma_1), (K_2[f_2 v_2], \gamma_2)) \triangleq \mathrm{PQ} \text{ when } f_1 = f_2 \in \mathrm{dom}(\Xi) \\
\mathrm{select}_\Xi((M_1, \gamma_1), (M_2, \gamma_2)) \triangleq \lightning \text{ otherwise.}
\end{array}$$

When we reason on recursive terms, there is in general no way to build a derivation tree. To avoid this problem, we introduce a notion of circular reasoning, by generalizing derivation trees to derivation graphs. Derivation graphs can use ***back-edges***, represented by a rule labelled by Circ, and the following new rule to generalize the terms we reason on:

$$\frac{(\Delta \cdot \Delta')_\alpha; \mathbb{C} \vdash \mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((M_1, \gamma_1), (M_2, \gamma_2)) \quad \Delta \vdash \rho : \Delta'}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((M_1\{\rho\}, \gamma_1), (M_2\{\rho\}, \gamma_2))}\ \textsc{Gen}$$

The rules Gen and Circ use a ***ground substitution*** $\Delta \vdash \rho : \Delta'$, that is a map from $\mathrm{dom}(\Delta')$ to ground terms, s.t. $\mathrm{codom}(\Delta') \subseteq \mathrm{Int}$ and $\Delta \vdash \rho(x) : \mathrm{Int}$ for all $x \in \mathrm{dom}(\rho)$. For sake of simplicity, we allow back-edges (i.e. Circ rule) to only point to a Gen rule. The rule Gen is used to generalize the current SKOR we reason on, in order to reach a point in the structure where we reason on the same SKOR, so that the rule Circ can be used. This combination of generalization and back-edges follows ideas developed with cyclic proofs in [Brotherston and Simpson 2011].

In practice, we may reach a point where we have "almost" the same relation, up-to some abstract reasoning. (like commutation of addition between a ground term and a term). So this inference system is parametrized by a notion of reduction $\looparrowright$, that is used in the following rule Rewrite to reason abstractly on terms:

$$\frac{\Delta_\alpha; \mathbb{C} \vdash \mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((K_1[M_1'], \gamma_1), (K_2[M_2'], \gamma_2)) \quad \Delta \vdash M_1 \looparrowright M_1' \quad \Delta \vdash M_2 \looparrowright M_2'}{\Delta_\alpha; \mathbb{C} \vdash \mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((K_2[M_1], \gamma_1), (K_1[M_2], \gamma_2))}\ \textsc{Rewrite}$$

For the reasoning with SKORs to be sound, we have to enforce that $\looparrowright$ preserve equivalence of terms. This is detailed in Appendix F.

In the following, we suppose that $\looparrowright$ includes reasoning modulo associativity and commutativity of arithmetic operations $\oplus \in \{+, *\}$, represented by the rules:

- $\Delta \vdash K[M \oplus t] \looparrowright K[t \oplus M]$;
- $\Delta \vdash K[t_1 \oplus (t_2 \oplus M)] \looparrowright K[(t_1 \oplus t_2) \oplus M]$;

It is crucial that we do not transform $K[M \oplus M']$ into $K[M' \oplus M]$, since this does not preserve contextual equivalence. This is why we only commute ground terms t with M, since t cannot modify the heap.

Definition 8.1. a ***derivation graph*** $\mathcal{S}$ is an oriented pointed labelled graph $(\mathcal{N}, \mathrm{r}, \mathrm{Sequent}, \mathrm{Edge}, \mathrm{Annot})$ s.t.:

- When removing all the edges labelled with Circ, we get a tree, called the ***tree substructure*** associated to S. Taking two nodes $\mathrm{n}_1, \mathrm{n}_2$ in S, we write $\mathrm{n}_1 \prec_{\mathcal{S}} \mathrm{n}_2$ if there exists a non-empty path from n_1 to n_2 in this tree.
- For all nodes n labelled by Gen and Circ, Annot(n) contains the ground substitution ρ used by this rule;
- For all $\mathrm{n}_1, \mathrm{n}_2 \in \mathcal{N}$, if $\mathrm{Edge}(\mathrm{n}_1) = (\{\mathrm{n}_2\}, \textsc{Circ})$ then $\mathrm{n}_2 \prec_{\mathcal{S}} \mathrm{n}_1$ and there exists n_0 s.t. $\mathrm{Edge}(\mathrm{n}_0) = (\{\mathrm{n}_2\}, \textsc{Gen})$. Writing $(\Delta_i)_{\alpha_i}; \mathbb{C}_i \vdash \phi$ for $\mathrm{Sequent}(\mathrm{n}_i)$ (with $i \in \{1, 2\}$) and $\Delta_i \vdash \rho_i : \Delta_i'$ for $\mathrm{Annot}(\mathrm{n}_i)$ (with $i \in \{0, 1\}$), then $\Delta_0' = \Delta_1'$, $\phi_2\{\rho_2\} = \phi_1$. Moreover, the rule Unf is used on the path between n_2 and n_1

This last condition ensuring the use of a rule Unf before using a back-edge is crucial to ensure the soundness of our reasoning.

8.3 Stateful Factorial

We now build a derivation graph for two values $v_1^{\mathrm{f}}, v_2^{\mathrm{f}}$ of type $\mathrm{Int} \to \mathrm{Int}$ that compute the factorial function. The first one is a naive implementation, while the second one is an optimized version that perform tail recursion and use a reference to store the result:

```
let rec fact₁ n =                 |  let fact₂ n =
  if (n ≤ 1) then 1               |    let acc = ref 1 in
  else (n * fact₁ (n − 1))        |    let rec aux m =                    } v_aux
                                  |      if (m ≤ 1) then () else (acc := m*!acc; aux (m − 1))
                                  |    in aux n; !acc
```

It is worth noticing that the recursive calls of v_1^{f} and v_2^{f} are of different type (resp. $\mathrm{Int} \to \mathrm{Int}$ and $\mathrm{Int} \to \mathrm{Unit}$). The first part of the derivation graph is:

$$\mathrm{E}\ \dfrac{\mathbf{H}_1\ \overset{\mathrm{V}_\iota}{\Delta_2; \{n \le 1\} \vdash \mathbb{V}[\![\mathrm{Int}]\!]((1,\gamma_1),(1,\gamma_2))} \qquad \dfrac{\dfrac{\mathcal{S}_1}{\Delta_4; \mathbb{C}_1 \vdash \mathbb{S}^{\mathrm{PI}}[\![\mathrm{Int}]\!]((\mathtt{m}_0 * \mathtt{fact}_1\ \mathtt{m}_1, \gamma_1), (\mathtt{aux}\ \mathtt{m}_2; !\mathtt{acc}, \gamma_2))}\ \textsc{Gen},\mathrm{n}}{\Delta_3 \vdash \rho_1 : \Delta_1' \qquad \mathbf{H}_2 \quad \Delta_3; \mathbb{C}_1 \vdash \mathbb{S}^{\mathrm{PI}}[\![\mathrm{Int}]\!]((\mathtt{n} * \mathtt{fact}_1\ \mathtt{n}_1, \gamma_1), (\mathtt{aux}\ \mathtt{n}_2; !\mathtt{acc}, \gamma_2))}}{\dfrac{\dfrac{\Delta_1; \varnothing \vdash \mathbb{E}[\![\mathrm{Int}]\!](v_1^{\mathrm{f}}\ \mathtt{n}, v_2^{\mathrm{f}}\ \mathtt{n})}{\mathbf{H}_0 \quad \varepsilon; \varnothing \vdash \mathbb{V}[\![\mathrm{Int} \to \mathrm{Int}]\!](v_1^{\mathrm{f}}, v_2^{\mathrm{f}})}\ \mathrm{V}_\to}{\varepsilon; \varnothing \vdash \mathbb{E}[\![\mathrm{Int} \to \mathrm{Int}]\!](v_1^{\mathrm{f}}, v_2^{\mathrm{f}})}\ \mathrm{E}}{\varepsilon; \varnothing \vdash v_1^{\mathrm{f}} \simeq_{\mathrm{skor}} v_2^{\mathrm{f}} : \mathrm{Int} \to \mathrm{Int}}\ \textsc{Init}}$$

where two premises of E are not indicated, corresponding to an unsatisfiable context $\{n \le 1, n > 1\}$ (and terminated via the rule Unsat), and:

- $\Delta_1 = [n \mapsto \mathrm{Int}]$, $\Delta_2 = \Delta_1 \cdot [acc \mapsto \mathrm{ref\ Int}]$ and $\Delta_3 = \Delta_2 \cdot [y, y' \mapsto \mathrm{Int}]$
- $\mathbf{H}_0 = (\varepsilon, \varepsilon, \varepsilon, \varepsilon)$, $\mathbf{H}_1 = (\varepsilon, [acc \mapsto 1], \varepsilon, [acc \mapsto 1])$ and $\mathbf{H}_2 = (\varepsilon, [x \mapsto y], \varepsilon, [x \mapsto y'])$
- $\Delta_1' = [m_0, m_1, m_2 \mapsto \mathrm{Int}]$, $\Delta_4 = \Delta_3 \cdot \Delta_1'$ and $\rho = [m_0 \mapsto n] \cdot [m_1 \mapsto n_1] \cdot [m_2 \mapsto n_2]$;

- $\mathbb{C}_1 = \{n > 1, n_1 = n - 1, n_2 = n - 1, y' = n * y\}$.

$\mathcal{S}_1$ is defined as:

$$\dfrac{\dfrac{\mathrm{V}_\iota \dfrac{}{\mathbf{H}_3 \quad \Delta_5; \mathbb{C}_2 \vdash \mathbb{V}[\![\mathrm{Int}]\!]((m_1, \gamma_1), (y_2, \gamma_2))} \quad \dfrac{\dfrac{\textsc{Circ to n with } \rho_2 = [m_0 \mapsto (m_0 * m_1)] \cdot [m_1 \mapsto m_1'] \cdot [m_2 \mapsto m_2']}{\Delta_6; \mathbb{C}_3 \vdash \mathbb{S}^{\mathrm{PI}}[\![\mathrm{Int}]\!](((\mathtt{m_0} * \mathtt{m_1}) * \mathtt{fact_1\ m_1'}), \gamma_1), (\mathtt{aux\ m_2'}; \mathtt{!acc}, \gamma_2))}}{\mathbf{H}_4 \quad \Delta_6; \mathbb{C}_3 \vdash \mathbb{S}^{\mathrm{PI}}[\![\mathrm{Int}]\!]((\mathtt{m_0} * (\mathtt{m_1} * \mathtt{fact_1\ m_1'}), \gamma_1), (\mathtt{aux\ m_2'}; \mathtt{!acc}, \gamma_2))}\ \textsc{Rewrite}}{\mathrm{E}}}{\textsc{Unf}\ \dfrac{\Delta_4; \mathbb{C}_1 \vdash \mathbb{E}[\![\mathrm{Int}]\!]((\mathtt{m_0} * \gamma_1(\mathtt{fact_1})\ \mathtt{m_1}), \gamma_1), (\gamma_2(\mathtt{aux})\ \mathtt{m_2}; \mathtt{!acc}, \gamma_2))}{\Delta_4; \mathbb{C}_1 \vdash \mathbb{S}^{\mathrm{PI}}[\![\mathrm{Int}]\!]((\mathtt{m_0} * \mathtt{fact_1\ m_1}, \gamma_1), (\mathtt{aux\ m_2}; \mathtt{!acc}, \gamma_2))}}$$

where

$\Delta_5 = \Delta_4 \cdot [y_2 \mapsto \mathrm{Int}], \mathbb{C}_2 = \mathbb{C}_1 \cup \{m_1 \leq 1, m_2 \leq 1\}$ $\quad \mathbf{H}_3 = (\varepsilon, [acc \mapsto y_2], \varepsilon, [acc \mapsto y_2])$

$\Delta_6 = \Delta_4 \cdot [y_2, y_3, m_1', m_2' \mapsto \mathrm{Int}]$ $\quad \mathbf{H}_4 = (\varepsilon, [acc \mapsto y_2], \varepsilon, [acc \mapsto y_3])$

$\mathbb{C}_3 = \mathbb{C}_2 \cup \{m_1 > 1, m_2 > 1, m_1' = m_1 - 1, m_2' = m_2 - 1, y_3 = y_2 * m_2\}$

and where two of the premises of E are not indicated, corresponding to sequents whose contexts are $\{m_1 \leq 1, m_2 > 1\}$ and $\{m_1 > 1, m_2 \leq 1\}$ and whose SKOR is $\mathbb{S}^{\nmid}[\![\mathrm{Int}]\!]_\Xi$.

8.4 Automatic Construction of Derivation Graphs

We now present a way to build automatically a derivation graph, based on the automated construction of a derivation tree presented in Section 6.6. Suppose that we are trying to prove $\Delta_\alpha; \mathbb{C} \vdash \mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((M_1, \gamma_1), (M_2, \gamma_2))$. We look at the path below this current node, looking for sequents of the shape $\Delta'_\alpha; \mathbb{C}' \vdash \mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((M_1', \gamma_1), (M_2', \gamma_2))$, with $j < j'$.

Then, we try the following heuristic: we look for two terms M_1^0, M_2^0 and two substitutions ρ, ρ' s.t.: $M_i \curvearrowright M_i^0\{\rho\}$ and $M_i' \curvearrowright M_i^0\{\rho'\}$. if we can find such terms and substitutions, we then backtrack to the point where the sequent $\Delta'_\alpha; \mathbb{C}' \vdash \mathbb{S}^{\mathrm{PI}}[\![\tau]\!]_\Xi((M_1', \gamma_1), (M_2', \gamma_2))$ was introduced, and we apply the rule Rewrite and Gen (with ρ') there. Using this heuristic, we are able to build automatically the derivation graph for the factorial examples presented previously.

Compared to what we have done for derivation trees, this construction may fail, in which case the user could provide some help by indicating which rules should be applied.

8.5 Construction of the SMTM

The use of ground substitutions in the rules Gen and Circ has to be reflected in the symbolic transition system we build. To do so, we extend of the SMTM with Player τ-transitions $d_{\mathrm{P}\tau} : A \times A \rightharpoonup \mathrm{GSubst}$. To define the execution relation associated to such transitions, one has to define the ***closure*** $\mathbf{c}_\eta(\rho)$ of a ground substitution $\Delta \vdash \rho : \Delta'$ associated to a ground valuation $\eta \in \mathrm{GVal}(\Delta)$, which is the ***ground valuation*** $\{(x, t) \mid x \in \mathrm{dom}(\rho), t = [\![\rho(x)]\!]_\eta\}$.

From it, we define the execution relation $(a, \eta, \sigma, \kappa, h_1, h_2) \xrightarrow{\mathrm{P}\tau} (a', \eta', \sigma, \kappa, h_1, h_2)$ when $d_{\mathrm{P}\tau}(a, a') = \Delta \vdash \rho : \Delta'$ and $\eta' = \eta_{|\Delta} \cdot \mathbf{c}_\eta(\rho)$

We also allow Player transitions to be tagged as Player Internal (PI), following the fact that the rules E of the derivation graphs can themselves be annotated with this tag.

From a derivation graph $\mathcal{S}$, the construction of the skeletal representation $\mathrm{Skel}(\mathcal{S}, \mathrm{n})$ is defined by induction on the tree substructure of $\mathcal{S}$, extending the construction given in Section 7.3 to the new rules introduced previously. So writing (N, L) for $\mathrm{Edge}(\mathrm{n})$, with $N = \{\mathrm{n}_1, \ldots, \mathrm{n}_m\}$,

- If L=Circ, taking a_0 a fresh atomic state, we return $(\mathbb{M}, \varnothing, [a_0 \mapsto \{\mathrm{n}\}])$, where $\mathbb{M}$ is the SMTM with no transition and a unique atomic state a_0.
- If L=Gen, taking a_0 a fresh atomic state, we return $(\mathbb{M}, \mathbb{C}_1^f, \mathcal{F}_1 \cdot [a_0 \mapsto \{\mathrm{n}\}])$ where $\mathbb{M}$ is the SMTM constructed from $\mathrm{Skel}(\mathcal{S}, \mathrm{n}_1) = (\mathbb{M}_1, \mathbb{C}_1^f, \mathcal{F}_1)$ with a_0 as new initial atomic state, together with an Pτ-transition from a_0 to a_0^1, labelled with ρ, the annotation of n.
- If L $\in$ {Rewrite, Unf}, then we return $\mathrm{Skel}(\mathcal{S}, \mathrm{n}_1)$.

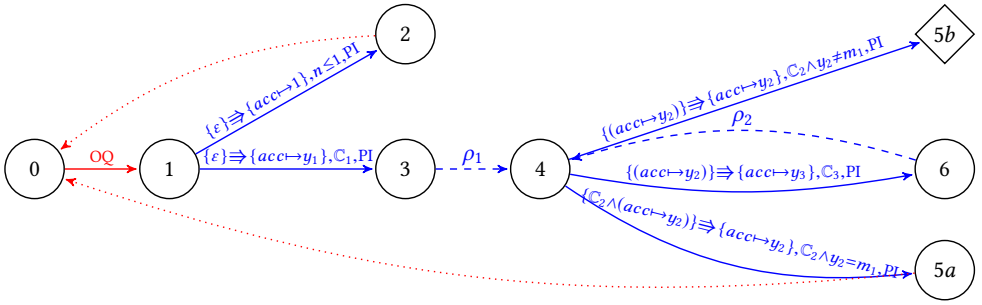


Fig. 8. SMTM for the Stateful Factorial Example

Because of the new PI-transitions, the SMTMs are not bipartite anymore, however by ignoring the internal atomic states we can recover this property. We thus introduce two transition functions $d_{\mathbf{e},\text{PA}}$ and $d_{\mathbf{e},\text{PQ}}$ that compose the succession of PI transition of d_P until a PA or a PQ transition is reached. It is formally defined as $d_{\mathbf{e},\mathbf{x}} \triangleq \text{dom}(d_{\mathbf{x}}) \circ (\text{dom}(d_{\text{PI}}) \circ \text{dom}(d^{=}_{\text{P}\tau}))^*$ (with $\mathbf{x} \in \{\text{PA}, \text{PQ}\}$).

We then redefined $d_{\mathbf{e}}$ and $d_{\mathbf{wb}}$ used to define the closure of a SMTM as:

- $d_{\mathbf{e}} \triangleq (d_{\mathbf{e},\text{PA}} \cup d_{\mathbf{e},\text{PQ}}) \circ \text{dom}(d_O) \circ d^{=}_{\text{O}\epsilon}$;
- $d_{\mathbf{wb}} \triangleq d_{\mathbf{e},\text{PA}} \circ (\text{dom}(d_{\text{OA}}) \circ d^{=}_{\text{O}\epsilon} \circ \text{dom}(d_{\mathbf{e},\text{PQ}}))^* \circ \text{dom}(d_{\text{OQ}}) \circ d^{=}_{\text{O}\epsilon}$

Finally, an extra closure operation has to be performed to take into account the Circ rules in $\mathcal{S}$. It corresponds to adding a Pτ-transition from atomic states corresponding to Circ rules to the corresponding Gen rules their backing edges are pointing to. Formally, it is defined as a new clause, that is added to first two given in Section 7.3.2:

3. for all atomic states a_1, a_2 s.t. there exists $\text{n}_1 \in \mathcal{F}(a_1)$ and $\text{n}_2 \in \mathcal{F}(a_2)$ with $\mathcal{S}.\text{Edge}(\text{n}_1) = (\text{Circ}, \{\text{n}_2\})$ and $\mathcal{S}.\text{Annot}(\text{n}_1) = \rho$, we have $\bar{d}_{\text{P}\tau}(a_1, a_2) = \rho$.

A simplified version of the SMTM generated for the stateful factorial example is given in Figure 8.

9 NON-REACHABILITY ANALYSIS OF STRUCTURED-MEMORY TRANSITION MACHINE

We now present some techniques to automatically check if there exists runs in SMTMs that can reach failed configurations. One of the main source of difficulty to do so is the fact that the size of the heaps that appear in the runs can be unbounded. Indeed, considering a program **fun** _ $\rightarrow$ (`let y = ref0 in` M), since the context can call an unbounded number of time this function, this will result in the creation of an unbounded number of references.

To avoid this problem, we will only consider for now programs that never create references inside a function body. In this setting, all the references are created in the first Player transition of the SMTM, that can be taken only one time.

This means that we can represent the heaps h carried in the configurations of the SMTM $\mathbb{M}$ as a sequence $y_1, \ldots, y_n$ of integer variables that represent the codomain of h.

The factorial example given in Section 8.3 does not satisfy this restriction, since the stateful version does allocate a reference in its body. However, it is still possible to handle this example by using the fact that these two programs are values rather than terms. So we can use the fact that $\Gamma \vdash (\textbf{fun}\ x \rightarrow M_1) \simeq_{\text{ctx}} (\textbf{fun}\ x \rightarrow M_2) : \tau \rightarrow \sigma$ iff $\Gamma, x : \tau \vdash M_1 \simeq_{\text{ctx}} M_2 : \sigma$.

We leave for future work the exploration of non-reachability analysis for programs that can create references inside a function body.

9.1 Decidability Result

Restricting ourselves to a language with finite datatypes, and considering recursion-free closed programs that never create references inside a function body, one get decidability of contextual

equivalence. Indeed, in this setting, the heaps h_1, h_2, the environment η and the knowledge function κ are all bounded. So we can represent all the runs of a SMTM as a non-deterministic pushdown system, and use the fact that reachability is then decidable. We finally rely on the completeness result of our framework for recursion-free programs, as proven in Section 10.

9.2 Translating Non-Reachability to Constrained Horn Clauses

We now present informally how to encode the reachability problem of failed configurations as a set of constrained Horn clauses, such that if this set is unsatisfiable, then no failed configuration is reachable.

A constrained Horn clause (CHC) [Bjørner et al. 2015, 2013] is a formula $\forall \vec{x}, \vec{y}. G_1 \wedge \ldots \wedge G_n \Rightarrow P(\vec{x})$, where P is an uninterpreted predicate of arity m, and G_1, G_n are atomic formulas written in language of arithmetic together with such uninterpreted predicates. Such a CHC is abbreviated as $P(\vec{x}) \Leftarrow G_1 \wedge \ldots \wedge G_n$. In the following, we allow disjunctions in the formulas $G_1, \ldots, G_n$, that can then be eliminated in order to get a CHC [Bjørner et al. 2015].

So we associate an uninterpreted predicate $P(b, y_1, \ldots, y_n, y'_1, \ldots, y'_n)$ to each sequence of transitions of a SMTM $\mathbb{M}$ that starts by an OQ-transition and ends by a PA-transition, with only PI, PQ and OA-transitions in between. These sequences correspond to well-bracketed computations. The integer variables $y_1, \ldots, y_n$ correspond to the codomain of the two heaps provided as input of these computations, while $y'_1, \ldots, y'_n$ correspond to the codomain of the two heaps we get as output of these computations. The variable b is used to indicate if a failed configuration is reached.

We then collect the arithmetic and heap constraints of such a sequence of transitions to define the body of the CHC. To deal with the Oϵ transitions and Pτ-transitions that we get from applying the closure defined in the second step of the construction of the SMTM $\mathbb{M}$, we perform non-deterministically recursive calls associated to each of these transitions.

Coming back to the SMTM for the well-bracketed state change example introduced in Section 2.2, the associated set of CHCs is:

$$\begin{array}{lll} P_0(b, y') & \Leftarrow & (y' = 0 \wedge b) \vee P_1(b, 0, y') \\ P_1(b, y, y') & \Leftarrow & y_1 = 0 \wedge (y_2 = y_1 \vee P_1(\text{True}, y_1, y_2)) \wedge y_3 = 1 \wedge (y_4 = y_3 \vee P_1(\text{True}, 1, y_4)) \\ & & \wedge((\neg b \wedge y_4 \neq 1 \wedge y_4 = z) \vee (y_4 = 1 \wedge ((y_4 = z \wedge b) \vee P_1(b, y_4, z)))) \end{array}$$

Then if $P_0(\text{False}, y')$ is unsatisfiable, there is no run that can reach a configuration whose state is 8, so that indeed the two programs are contextually equivalent.

- $P_0(b, y')$ corresponds to the initial transition, with y' being the value stored in the reference of the first program at the end of the run.
- $P_1(b, y, y')$ corresponds to runs starting in state 1.
- The formula $(y_2 = y_1 \vee P_1(\text{True}, y_1, y_2))$ corresponds to the two transitions for Opponent in state 3: either take the OA transition and keep the heap unchanged, or follows the Oϵ-transition back to state 1. The variable y_2 thus corresponds to the value stored in the reference of the first program after any of these two choices.
- Further in the clause, the formula $(\neg b \wedge y_4 \neq 1 \wedge y_4 = z)$ corresponds to reaching a configuration whose state is 8, while the formula $(y_4 = 1 \wedge ((y_4 = z \wedge b) \vee P_1(b, y_4, z)))$ corresponds to the choice in state 7, i.e. either stop there, so that b is true, or go back to state 1.

Our implementation is able to generate automatically such CHCs, whose satisfiability can then be checked by solvers like z3 [Hoder et al. 2011] or SeaHorn [Gurfinkel et al. 2015]. Examples like the irreversible and the well-bracketed state change examples can then be automatically checked, but also more complex examples like the callback with lock example, given in Appendix K, or the factorial example of Section 8.3.

Notice that our encoding of reachability into CHCs does not handle the valuation knowledge κ. Indeed, the way we deal with OQ-transition would simply pick unrestricted variables for the

valuation η needed at this point, without remembering the ones previously seen during the interaction. This means that we over-approximate the reachability problem, so that if the CHCs we get from the encoding is indeed satisfiable, we cannot be sure that indeed a failed configuration is reachable, i.e. that the two programs are not contextually equivalent. But in practice, this situation does not happen in the example of the literature. We leave for future work the precise study of this over-approximation and the design of techniques to tame it.

9.3 Evaluation of the Tool

The evaluation of our tool has to consider two steps: the generation phase of the Constrained Horn Clauses (CHC) by SyTeCi, and the checking phase of these CHC by the z3 solver (v4.4.1). The running times given below have been estimated on a laptop equipped with an Intel Core i5-5200U (2.20GHz) with 8Gb of RAM.

Considering the standard examples of the literature, like the representation independence (Section 2.1), the "awkward" and the well-bracketed state change (Section 2.2) or the callback-with-lock (Appendix K) examples, SyTeCi generates immediately (in less than 0.1s) the CHC. z3 then checks also immediately that they are unsatisfiable, proving that they are indeed contextually equivalent.

Important example of inequivalence are the Kierstead terms $K_{n,i}$ defined as:

$$\textbf{fun}\ \mathtt{f} \to \textbf{fun}\ \mathtt{x}_1 \to \mathtt{f}(\textbf{fun}\ \mathtt{x}_2 \to \mathtt{f}(\ldots \mathtt{f}(\textbf{fun}\ \mathtt{x}_n \to \mathtt{x}_i())) \ldots)$$

of type ((Unit $\to$ Unit) $\to$ Unit) $\to$ Unit. For $(m,j) \neq (n,i)$ one has that $K_{m,j}, K_{n,i}$ are not contextually equivalent. The running times for SyTeCi to compare $K_{n,j}$ and $K_{n,i}$, for $i \neq j$ (which are independent of i, j) are:

n	number of state of the SMTM	time to generate the CHC	time to check the CHC (via z3)
10	42	$< 0.1s$	$< 0.1s$
100	403	$1.2s$	$0.8s$
200	803	$8.0s$	$3.6s$

For programs with recursion, the situation is more complex, since the generation of the SMTM relies on unfolding recursive calls, and may diverge if we do not bound this number of unfolding. In practice, it appears that either SyTeCi succeeds to build the SMTM by unfolding one or two times the recursive calls, or never succeeds to perform circular reasoning. When it succeeds, the CHCs that SyTeCi generates can be hard to check for z3, even if they are succinct. This is the case of the factorial example presented in Section 8.3, that takes 34s for z3 to check.

Existing tools to reason automatically on contextual equivalence of programs are based on algorithmic game semantics, so they only work on restricted fragments on which contextual equivalence is decidable. Among these tools, one can cite:

- HOMER for a fragment of Idealized Algol [Hopkins and Ong 2009];
- CONEQCT for a fragment of Java [Murawski et al. 2015];
- HECTOR, for a fragment of ML [Hopkins et al. 2012].

Importantly, their complexity depends heavily on the size of the data-types (integers) they consider, while SyTeCi works with unbounded data-types, and so its complexity does not depend on their size. HECTOR is the only tool that targets a similar programming language as SyTeCi. Considering the examples presented in [Hopkins et al. 2012], SyTeCi is able to handle almost all of them, as fast as HECTOR, and in the case of the Kierstead terms, an order of magnitude faster. There is nonetheless one examples that SyTeCi cannot handle: the No-snapback example, that uses an immediately diverging term. To handle it, one would need to implement the notion of "inconsistent world" as presented in [Jaber and Tabareau 2015].

10 SOUNDNESS AND COMPLETENESS OF SYTECI

Starting from the derivation graph $\mathcal{S}$ associated to two programs M_1, M_2, we now sketch how to prove that if $\mathbb{M}_{\mathcal{S}}$ is safe, then M_1, M_2 are contextually equivalent (***soundness***). When M_1, M_2 are recursion-free we also prove that if they are contextually equivalent then $\mathbb{M}_{\mathcal{S}}$ is safe (***completeness***).

10.1 Concrete Kripke Open Relations

To prove these results, we rely on an operational technique to prove contextual equivalence of programs, called Concrete Kripke Open Relations (CKORs). They are relations $\mathcal{E}_{\mathcal{A}}[\![\tau]\!]w$ on extended terms with free functional variables, but without free ground variables. They are a direct adaptation of Kripke Open Bisimulations introduced in [Jaber and Tabareau 2015], but they use a step-indexed definition rather than a coinductive one.

To reason on heaps, CKORs handles ***worlds*** w, which are triples (s, h_1, h_2) formed by a state and two heaps. The definition of CKORs are indexed by a ***transition system of worlds*** (TSW) $\mathcal{A}$. It is is a tuple $(S, S^\natural, s_0, \delta)$ formed by a set of states S together with a subset $S^\natural$ of ***failed states***, an initial state s_0 and a transition relation $\delta \subseteq \text{World} \times \text{World}$. The ***initial world*** of a TSW $\mathcal{A}$, written w_0, is defined as $(\mathcal{A}.s_0, \varepsilon, \varepsilon)$. A ***failed world*** is a world w s.t. $w.s \in \mathcal{A}.S^\natural$. A TSW $\mathcal{A}$ is ***safe***, when there is no world $w \in \mathcal{A}.\delta^*(w_0)$. s.t. .

It is crucial to notice that one can build a TSW $\mathcal{A}_{\mathbb{M}} = (S, S^\natural, s_0, \delta)$ from the execution relation $\rightarrow_{\mathbb{M}}$ associated to a MSTM $\mathbb{M} = (A, a_0, A^\natural, d_P, d_O, d_{O\epsilon})$, by taking:

- $S = A \times \text{GVal} \times \text{List}(A \times \text{GVal}) \times (A \rightharpoonup \mathcal{P}(\text{GVal}))$
- $S^\natural \triangleq A^\natural \times \text{GVal} \times \text{List}(A \times \text{GVal}) \times (A \rightharpoonup \mathcal{P}(\text{GVal}))$
- $s_0 \triangleq (a_0, \varepsilon, [], \{a_0, \varepsilon\})$ and $\delta = (\xrightarrow{\text{PA}} \cup \xrightarrow{\text{PQ}}) \circ (\xrightarrow{\text{OA}} \cup \xrightarrow{\text{OA}})\circ \xrightarrow{\text{O}\epsilon}{}^{=}$.

Notice that $\mathbb{M}$ is safe iff $\mathcal{A}_{\mathbb{M}}$ is safe.

The definition of CKORs is given in Appendix D. CKORs are shown to be sound and complete using a correspondence with the fully abstract operational game model of RefML [Jaber 2015; Laird 2007]. We simplify this model for SimpleML programs with RefML contexts in Appendix B.

The proof of soundness and completeness proof of CKORs w.r.t. the operational game model is a direct adaptation of soundness and completeness of Kripke Open Bisimulations proven in [Jaber and Tabareau 2015].

10.2 Characteristic Formula

One then need to relate Symbolic Kripke Open Relations and Concrete Kripke Open Relations, in order to import the soundness and completeness results of CKORs into SKORs. To do so, we introduce the ***characteristic formula*** $\psi_{\mathcal{S}}$ associated to a derivation graph $\mathcal{S}$. It is presented in Appendix E. This formula is written in a logic mixing arithmetic predicates, quantification over integers and locations, ***temporal*** modalities, and coinductive predicates. The notion of validity of $\psi_{\mathcal{S}}$ w.r.t. a TSW $\mathcal{A}$ and a world $\mathbf{w}$, written $\mathbf{w} \models_{\mathcal{A}} \mathcal{T}_{\mathcal{S}}$, is defined using a Kripke semantics.

SKORs provide a sound proof technique, as presented in Appendix F, in the sense that if there exists a safe TSW $\mathcal{A}$ that validates $\psi_{\mathcal{S}}$ then M_1, M_2 are contextually equivalent.

Theorem 10.1 (Soundness). *Given a derivation graph $\mathcal{S}$ whose root is $\vdash M_1 \simeq^{\Gamma}_{\text{skor}} M_2 : \tau$, and a safe TSW $\mathcal{A}$, if $\mathbf{w}_0 \models_{\mathcal{A}} \mathcal{T}_{\mathcal{S}}$, then $\Gamma \vdash M_1 \simeq_{\text{ctx}} M_2 : \tau$.*

For recursion-free terms, we show in Appendix G that SKORs are also complete: if M_1, M_2 are contextually equivalent, then there exists a transition system $\mathcal{A}$ that validates $\psi_{\mathcal{S}}$. This transition system $\mathcal{A}$ does not correspond to $\mathcal{A}_{\mathbb{M}_{\mathcal{S}}}$ for this result, but comes from a result proven in [Jaber and Tabareau 2015], that does not provide an algorithmic way to build an effective representation of $\mathcal{A}$.

Theorem 10.2 (Completeness). *Taking two recursion-free terms M_1, M_2 s.t. $\Gamma \vdash M_1 \simeq_{\text{ctx}} M_2 : \tau$, then there exists an (effectively constructible) derivation tree $\mathcal{S}$ whose root is $\vdash M_1 \simeq^{\Gamma}_{\text{skor}} M_2 : \tau$, and a safe TSW $\mathcal{A}$ s.t. $\mathbf{w}_0 \models_{\mathcal{A}} \mathcal{T}_{\mathcal{S}}$.*

We then prove that the TSW $\mathcal{A}_{\mathbb{M}_{\mathcal{S}}}$ associated to the SMTM $\mathbb{M}_{\mathcal{S}}$ always validates the characteristic formula $\psi_{\mathcal{S}}$ extracted from $\mathcal{S}$. This is the case even if the two programs M_1, M_2 that are related by $\mathcal{S}$ are not contextually equivalent, because of the possible failed worlds in $\mathcal{A}_{\mathbb{M}_{\mathcal{S}}}$, that validate any formula ψ in the Kripke semantics we use.

Theorem 10.3. *Given $\mathcal{S}$ a derivation graph, then $\mathbf{w}_0 \models_{\mathcal{A}_{\mathbb{M}_{\mathcal{S}}}} \mathcal{T}_{\mathcal{S}}$.*

It is proven in Appendix I.2 by induction over an order on the states of $\mathbb{M}_{\mathcal{S}}$ inherited from the order over the nodes of the tree-substructure of $\mathcal{S}$, via the mapping $\mathcal{F}$.

Combining this result with Theorem 10.1 and the fact that $\mathbb{M}_{\mathcal{S}}$ is safe iff $\mathcal{A}_{\mathbb{M}_{\mathcal{S}}}$ is safe, we get the following soundness result:

Corollary 10.4. *Given a derivation graph $\mathcal{S}$ whose root is $\vdash M_1 \simeq^{\Gamma}_{\text{skor}} M_2 : \tau$, if $\mathbb{M}_{\mathcal{S}}$ is safe, then $\Gamma \vdash M_1 \simeq_{\text{ctx}} M_2 : \tau$.*

Finally, for recursion-free terms, we prove that if there exists a safe transition system $\mathcal{A}$ that validate $\psi_{\mathcal{S}}$, then $\mathcal{A}_{\mathbb{M}_{\mathcal{S}}}$ is also safe, which gives us the wanted completeness result.

Theorem 10.5. *Given $\mathcal{S}$ a derivation graph, suppose that there exists a safe TSW $\mathcal{A}'$ such that $\mathbf{w}'_0 \models_{\mathcal{A}'} \mathcal{T}_{\mathcal{S}}$. Then $\mathcal{A}_{\mathbb{M}_{\mathcal{S}}}$ is safe.*

It is proven in Appendix J by introducing a notion of simulation between TSW. This means that $\mathcal{A}_{\mathbb{M}_{\mathcal{S}}}$ does not have unnecessary reachable failed worlds. Combining this result with Theorem 10.2, we get the wanted completeness result:

Corollary 10.6. *Taking two* **recursion-free** *terms M_1, M_2 s.t. $\Gamma \vdash M_1 \simeq_{\text{ctx}} M_2 : \tau$, then there exists a derivation tree $\mathcal{S}$ s.t. $\mathbb{M}_{\mathcal{S}}$ is safe.*

11 RELATED WORK

As sketched in the introduction, there is a vast literature on both logic for relational properties and automatic techniques for equivalence of "first-order" programs. We focus here only on works about contextual equivalence.

11.1 Algorithmic Game Semantics

As we explained in the introduction, algorithmic game semantics (AGS) [Cotton-Barratt et al. 2015, 2017; Hopkins et al. 2011; Murawski 2005; Murawski and Tzevelekos 2011a, 2012] is a powerful technique to decide contextual equivalence of ML programs. Our framework follows a different purpose, by reducing contextual equivalence to a problem where approximations can then be applied to check the equivalence.

While we associate a unique transition system to the two programs we try to relate, AGS associates an automaton to each program, and then compare these automata. So they do not exploit during the construction of the automata the various synchronization points (like callbacks or reduction to values) that may exist. This explain the restriction they need on the type-theoretic order, to be able to encode the pointer structure of the plays forming the denotation of terms, and get decidability results. Moreover, AGS does not reason about ground values symbolically, and thus needs to be restricted to language with finite data-types.

To deal with freshness of locations, they use nominal automata [Tzevelekos 2011]. It would be interesting to see if this could also be used in our framework.

11.2 Logics to Reason on Kripke Logical Relations

Following the seminal work of Plotkin & Abadi [Plotkin and Abadi 1993] on logic for parametric polymorphism, some logics have been designed to reason about KLRs, namely LSLR [Dreyer et al. 2009] and LADR [Dreyer et al. 2010b]. These logics are designed for richer language than SimpleML (fragment of ML with recursive and polymorphic types, and higher-order references for LADR). They try to prove contextual equivalence of programs directly in the logic, while our framework rather seeks to reduce the problem to non-reachability in TSW. It does not seem easy to perform automatic proof search in these logics, since they quantify over terms of the programming language.

Compared to our work, they abstract over step-indexes using guarded recursive definitions via the modality $\triangleright$. It would be interesting to see if it is possible to design a system of circular proof for such guarded recursive definitions.

LADR shares similarities with Symbolic Kripke Open Relations. Indeed, they use a S4 modality $\Box$ to reason about future worlds, which is similar to one of the modality we use to define characteristic formulas. However, their reasoning over future worlds is done directly inside the logic, while in our case it is done a posteriori when generating the temporal characteristic formula. Moreover, LADR is probably not complete for recursion-free terms.

12 FUTURE WORK

12.1 Richer Languages

In order to consider programs that can disclose locations, we would need to carry a span of locations in configurations, to keep track of the relation between the disclosed one. We also believe that we can handle full ground references (i.e. references that can store locations) by a finer aliasing analysis. Higher-order references are more challenging, since symbolic reduction would then need to know what is stored in the heap. A possible solution would be to perform first a static-analysis on programs to collect all the possible functions that could be stored in references. Finally, polymorphic types may also fit in our framework, by using a nominal representation of polymorphic values, following [Jaber and Tzevelekos 2016].

12.2 Finer Treatment of Recursion

Our construction of the derivation graph associated to two programs with recursion only works when recursive calls of the two programs can be directly synchronized. It would be interesting to design heuristic to deal with examples where recursive calls of the two programs has to be unfolded at a different pace, in order to synchronize them. It should also be possible to use our framework to reason on contextual approximation up-to a bounded number of unfolding of recursive calls made by the program, and explore decidability results in this case.

12.3 Program Transformations

We would like to explore the possible uses of our framework to prove soundness of program transformations like CPS-translations or defunctionalization. More precisely, considering a program program transformation T, we would like to prove that if $M \simeq_{\mathrm{ctx}} N$, then $T(M) \simeq_{\mathrm{ctx}} T(N)$. To do so, we could study the transformation on the derivation graphs that such a program transformation T would induced.

ACKNOWLEDGMENTS

This work was partially funded by ANR Project RAPIDO ANR-14-CE25-000

REFERENCES

Amal Ahmed, Derek Dreyer, and Andreas Rossberg. 2009. State-dependent Representation Independence. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL '09)*. ACM, New York, 14.

Anindya Banerjee and David A. Naumann. 2005. Ownership Confinement Ensures Representation Independence for Object-oriented Programs. *J. ACM* 52, 6 (2005), 894–960. https://doi.org/10.1145/1101821.1101824

Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, and Andrey Rybalchenko. 2015. Horn clause solvers for program verification. In *Fields of Logic and Computation II*. Springer, 24–51.

Nikolaj Bjørner, Ken McMillan, and Andrey Rybalchenko. 2013. On solving universally quantified horn clauses. In *International Static Analysis Symposium*. Springer, 105–125.

James Brotherston and Alex Simpson. 2011. Sequent Calculi for Induction and Infinite Descent. *J. Log. and Comput.* 21, 6 (Dec. 2011), 1177–1216. https://doi.org/10.1093/logcom/exq052

Ştefan Ciobâcă, Dorel Lucanu, Vlad Rusu, and Grigore Roşu. 2016. A Language-independent Proof System for Full Program Equivalence. *Form. Asp. Comput.* 28, 3 (2016), 469–497.

Conrad Cotton-Barratt, David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. 2015. Fragments of ML Decidable by Nested Data Class Memory Automata. In *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'15)*. 249–263.

Conrad Cotton-Barratt, Andrzej S. Murawski, and C.-H. Luke Ong. 2017. ML and Extended Branching VASS. In *Proceedings of the 26th European Symposium on Programming Languages and Systems (ESOP'17)*. 314–340.

Derek Dreyer, Amal Ahmed, and Lars Birkedal. 2009. Logical Step-Indexed Logical Relations. In *Proceedings of the 24th IEEE Symposium on Logic In Computer Science (LICS '09)*. IEEE, Washington, DC, USA, 71–80.

Derek Dreyer, Georg Neis, and Lars Birkedal. 2010a. The Impact of Higher-order State and Control Effects on Local Relational Reasoning. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming (ICFP '10)*. ACM, New York, 143–156.

Derek Dreyer, Georg Neis, Andreas Rossberg, and Lars Birkedal. 2010b. A Relational Modal Logic for Higher-order Stateful ADTs. In *Proceedings of the 37th ACM Symposium on Principles of Programming Languages (POPL '10)*. ACM, New York, 185–198.

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating Regression Verification. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE '14)*. ACM, New York, 349–360.

Benny Godlin and Ofer Strichman. 2009. Regression Verification. In *Proceedings of the 46th Design Automation Conference (DAC '09)*. ACM, New York, 6.

Arie Gurfinkel, Temesghen Kahsai, Anvesh Komuravelli, and Jorge A Navas. 2015. The SeaHorn verification framework. In *International Conference on Computer Aided Verification*. Springer, 343–361.

Kryštof Hoder, Nikolaj Bjørner, and Leonardo De Moura. 2011. muZ: An Efficient Engine for Fixed Points with Constraints. In *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV'11)*. Springer-Verlag, Berlin, Heidelberg, 457–462. http://dl.acm.org/citation.cfm?id=2032305.2032341

David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. 2011. A Fragment of ML Decidable by Visibly Pushdown Automata. In *Proceedings of the 38th International Conference on Automata, Languages and Programming - Volume Part II (ICALP'11)*. Springer-Verlag, Berlin, Heidelberg, 149–161.

David Hopkins, Andrzej S. Murawski, and C.-H. Luke Ong. 2012. HECTOR: An Equivalence Checker for a Higher-order Fragment of ML. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 774–780. https://doi.org/10.1007/978-3-642-31424-7_63

David Hopkins and C. H. Ong. 2009. Homer: A Higher-Order Observational Equivalence Model checkER. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV '09)*. Springer-Verlag, Berlin, Heidelberg, 654–660. https://doi.org/10.1007/978-3-642-02658-4_51

Chung-Kil Hur, Derek Dreyer, Georg Neis, and Viktor Vafeiadis. 2012. The Marriage of Bisimulations and Kripke Logical Relations. In *Proceedings of the 39th ACM Symposium on Principles of Programming Languages (POPL '12)*. ACM, New York, 59–72.

Guilhem Jaber. 2015. Operational Nominal Game Semantics. In *Proceedings of the 18th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2015)*. Springer Berlin Heidelberg, Berlin, Heidelberg.

Guilhem Jaber and Nicolas Tabareau. 2015. Kripke Open Bisimulation - A Marriage of Game Semantics and Operational Techniques. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS 2015)*.

Guilhem Jaber and Nikos Tzevelekos. 2016. Trace Semantics for Polymorphic References. In *Proceedings of the 31st ACM/IEEE Symposium on Logic in Computer Science (LICS '16)*. ACM, New York, 585–594.

Radha Jagadeesan, Corin Pitcher, and James Riely. 2009. Transactions on Aspect-Oriented Software Development V. Springer-Verlag, Berlin, Heidelberg, Chapter Open Bisimulation for Aspects, 72–132. https://doi.org/10.1007/978-3-642-02059-9_3

Vasileios Koutavas and Mitchell Wand. 2006. Small Bisimulations for Reasoning About Higher-order Imperative Programs. In *Conference Record of the 33rd ACM Symposium on Principles of Programming Languages (POPL '06)*. ACM, New York, 141–152.

Shuvendu K. Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. 2012. SYMDIFF: A Language-agnostic Semantic Diff Tool for Imperative Programs. In *Proceedings of the 24th International Conference on Computer Aided Verification (CAV'12)*. Springer-Verlag, Berlin, Heidelberg, 712–717.

James Laird. 2007. A Fully Abstract Trace Semantics for General. In *Proceedings of the 34th International Conference on Automata, Languages and Programming (ICALP'07)*. Springer-Verlag, Berlin, Heidelberg, 667–679.

Soren B. Lassen and Paul Blain Levy. 2007. Typed Normal Form Bisimulation. In *Proceedings of the 21st International Conference, and Proceedings of the 16th Annuall Conference on Computer Science Logic (CSL'07/EACSL'07)*. Springer-Verlag, Berlin, Heidelberg, 283–297. http://dl.acm.org/citation.cfm?id=2392389.2392418

Andrzej S. Murawski. 2005. Functions with local state: Regularity and undecidability. *Theoretical Computer Science* 338, 1 (2005), 315 – 349.

Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. 2015. A Contextual Equivalence Checker for IMJ*. In *Automated Technology for Verification and Analysis (ATVA'15)*. Springer International Publishing, 234–240.

Andrzej S. Murawski and Nikos Tzevelekos. 2011a. Algorithmic Nominal Game Semantics. In *Proceedings of the 20th European Conference on Programming Languages and Systems (ESOP'11)*. Springer-Verlag, Berlin, Heidelberg, 419–438.

Andrzej S. Murawski and Nikos Tzevelekos. 2011b. Game Semantics for Good General References. In *Proceedings of the 2011 IEEE 26th Annual Symposium on Logic in Computer Science (LICS '11)*. IEEE Computer Society, Washington, DC, USA, 75–84. https://doi.org/10.1109/LICS.2011.31

Andrzej S. Murawski and Nikos Tzevelekos. 2012. Algorithmic Games for Full Ground References. In *Proceedings of the 39th International Colloquium Conference on Automata, Languages, and Programming - Volume Part II (ICALP'12)*. Springer-Verlag, Berlin, Heidelberg, 312–324.

Andrew M. Pitts. 1996. Reasoning About Local Variables with Operationally-based Logical Relations. In *Proceedings of the 11th IEEE Symposium on Logic in Computer Science (LICS '96)*. IEEE, Washington, DC, USA, 152–.

Gordon D. Plotkin and Martín Abadi. 1993. A Logic for Parametric Polymorphism. In *Proceedings of the International Conference on Typed Lambda Calculi and Applications (TLCA '93)*. Springer-Verlag, London, UK, UK, 361–375.

Davide Sangiorgi, Naoki Kobayashi, and Eijiro Sumii. 2007. Environmental Bisimulations for Higher-Order Languages. In *Proceedings of the 22Nd IEEE Symposium on Logic in Computer Science (LICS '07)*. IEEE, Washington, DC, USA, 293–302.

Eijiro Sumii. 2009. A Complete Characterization of Observational Equivalence in Polymorphic lambda-calculus with General References. In *Proceedings of the 23rd CSL International Conference and 18th EACSL Conference on Computer Science Logic (CSL'09/EACSL'09)*. Springer-Verlag, Berlin, Heidelberg, 455–469.

Eijiro Sumii and Benjamin C. Pierce. 2005. A Bisimulation for Type Abstraction and Recursion. In *Proceedings of the 32Nd ACM Symposium on Principles of Programming Languages (POPL '05)*. ACM, New York, 63–74.

Nikos Tzevelekos. 2011. Fresh-register Automata. In *Proceedings of the 38th ACM Symposium on Principles of Programming Languages (POPL '11)*. ACM, New York, 295–306.

Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018. Verifying Equivalence of Database-driven Applications. *Proc. ACM Program. Lang.* 2, POPL, Article 56 (2018), 29 pages. https://doi.org/10.1145/3158144