School of Electronic Engineering and Computer Science
Queen Mary University of London

PhD Thesis

# Bounded Verification of Higher-Order Stateful Programs

Yu Yang Lin Hou

supervised by

Dr Nikos Tzevelekos

May, 2021

Submitted in partial fulfillment of the requirements of the Degree of
Doctor of Philosophy

I, Yu Yang Lin Hou, confirm that the research included within this thesis is my own work or that where it has been carried out in collaboration with, or supported by others, that this is duly acknowledged below and my contribution indicated. Previously published material is also acknowledged below.

I attest that I have exercised reasonable care to ensure that the work is original, and does not to the best of my knowledge break any UK law, infringe any third party's copyright or other Intellectual Property Right, or contain any confidential material.

I accept that the College has the right to use plagiarism detection software to check the electronic version of the thesis.

I confirm that this thesis has not been previously submitted for the award of a degree by this or any other university.

The copyright of this thesis rests with the author and no quotation from it or information derived from it may be published without the prior written consent of the author.

Signature:

Date:

Details of collaboration and publications:

- Chapter 3 references published work in collaboration with Dr. Nikos Tzevelekos, "A Bounded Model Checking Technique for Higher-Order Programs", in the proceedings of *The Symposium on Dependable Software Engineering Theories, Tools and Applications*, SETTA 2019.
- Chapter 4 references published work in collaboration with Dr. Nikos Tzevelekos, "Symbolic Execution Game Semantics", in the proceedings of *Formal Structures for Computation and Deduction*, FSCD 2020.

*To my parents, for their love, care and encouragement;*
*and to all the teachers who guided me along the way.*

# Abstract

In this thesis we explore bounded verification techniques for higher-order stateful programs. We consider two settings: *open* and *closed* higher-order, which are defined by the type-order of free variables present in each. Closed higher-order programs allow free variables only if they are of ground type, whereas open higher-order programs generalise this by allowing free variables of arbitrary order. We elaborate on the challenges involved in reasoning within said settings, and define a higher-order stateful language—an ML-like $\lambda$-calculus with recursion and higher-order global state—as our vehicle of study. We define a Bounded Model Checking technique for closed higher-order programs via defunctionalization using nominal techniques, and a Symbolic Execution Game Semantics to perform Bounded Symbolic Execution of open higher-order programs. Contributions presented in this thesis involve theoretical and experimental results. On the theoretical side, all approaches defined herein are sound and *bounded-complete* in the sense that they report errors if and only if errors are reachable up to the given bound—all results necessary to show this are included. For the experimental side, we implemented prototype tools for each technique, collected and created benchmarks to test each higher-order setting, and measured the performance of our tools to compare them to other relevant existing tools. Results presented herein for closed and open higher-order programs have been published in SETTA 2019 and FSCD 2020 respectively.

# Acknowledgements

First, I would like to thank my supervisor, Nikos Tzevelekos—the work presented here is a result of his constant support and patience. He guided me through my research studies with dedication and exemplary academic character and integrity. I greatly appreciate the time and effort he spent supervising me.

I would also like to thank my PhD progression review committee members, Professor Edmund Robinson and Dr. Greta Yorsh, for their advice, feedback and administrative work. In particular, I want to thank them for forcing me to start writing this thesis.

I likewise appreciate my colleagues in the School of Electronic Engineering and Computer Science at Queen Mary, especially my friends in the Theory Group.

Finally, I would like to thank my family for their unwavering support and faith in me.

# Contents

# List of Figures

# List of Tables

# Chapter 1

# Introduction

Higher-order languages provide significant advantages in the form of higher levels of abstraction over their first-order counterparts. Compared to first-order programs, higher-order programs, especially in a functional setting, are often shorter and more modular [39]. For instance, higher-order functions such as `foldr` and `map` provide abstractions that enable modular manipulation of recursive data structures such as lists and trees. The expressive power provided by higher-order features is desirable considering the complexity of modern software, making these features widespread over a variety of languages. At the level of functions, languages such as Python, Scala, Clojure, Haskell and OCaml all provide lambda abstractions (anonymous functions) of arbitrary order. Since Java 8 and the C++14 standard, both C++ and Java have similarly allowed the use of higher-order functions more easily and directly than previously possible. At the level of programs, languages that use libraries can all be considered higher-order in the sense that they allow the use of undefined code in programs.

With higher-order features prominent in many programming languages, it is easy to see why techniques to automatically reason about higher-order programs are desirable. However, the expressive power provided by higher-order languages introduces many challenges not present in a first-order setting and comes at a cost to formally reasoning about program correctness. With the increase in power of abstraction, higher-order behaviours, especially when combined with higher-order state, can very quickly become intractable. In contrast to the spread of higher-order programming features, techniques to statically check general higher-order behaviours are fewer relative to first-order approaches available. This is what we address herein. In this thesis, we aim to develop static depth-bounded verification techniques for higher-order programs that exhaustively report errors, and prove their correctness with respect to sound safety errors, meaning that the techniques we shall present do not report false positives. We shall also empirically evaluate said techniques and provide comparisons for each.

## 1.1 Thesis Outline

This thesis includes theoretical and empirical results on verifying higher-order software. On the theoretical side, we device techniques and prove properties that these have. The empirical aspect covers implementation, benchmarking, and comparison of our techniques with prior work. The contents of this thesis are divided into chapters as follows.

**Chapter 1** introduces the questions addressed, as well as the relevant literature and concepts in verification, open and closed higher-order programs, and game semantics. Related work will also be presented here.

**Chapter 2** defines a context for all the contributions presented in this thesis. Here we present the syntax and semantics of the higher-order language (and variants) we shall be using in the chapters that follow. We also provide all the motivational examples and background theoretical content that will be referenced throughout the thesis.

**Chapter 3** presents a Bounded Model Checking technique based on defunctionalisation and points-to analysis for *closed* higher-order programs. The Bounded Model Checking procedure is described in detail, which includes our Bounded Model Checking algorithm, theoretical results in terms of a proof of soundness and correctness of the technique, an optimisation based on points-to analysis, and a preliminary implementation of the algorithm in a prototype tool called BMC-2, which we benchmark and compare with related tools and techniques.

**Chapter 4** presents a Symbolic Execution technique for *open* higher-order programs, which generalises the question of verifying higher-order programs. In this chapter we discuss a symbolic technique based on operational game semantics. Contributions include a presentation of a game semantics suitable for verification, theoretical results in terms of proofs of soundness and completeness of the semantics, a symbolic version of the game semantics necessary for symbolic reasoning, a proof of soundness for the symbolic technique, and an implementation of the technique, which we test with a custom higher-order benchmark inspired by real-world errors.

**Chapter 5** evaluates and summarises the contributions in this thesis and presents future work. This includes tools and techniques similar to ours, and possible future directions, such as unbounded verification, further development of the techniques, and real-world application.

Results presented in Chapter 3 and Chapter 4 have been published in collaboration with Nikos Tzevelekos in [48] and [49] respectively.

### 1.1.1 Main Contributions

The main contributions presented in this thesis can be organised into two parts:

- Closed higher-order programs; published in [48]:
  - a Bounded Model Checking technique based on defunctionalisation and points-to analysis;
  - a proof of soundness and correctness of the technique;
  - and a preliminary implementation of the technique and benchmarks.
- Open higher-order programs; published in [49]:
  - a Game Semantics for higher-order libraries;
  - a proof of soundness and completeness of the Game Semantics;
  - a Symbolic Execution technique based on our Game Semantics;
  - a proof of soundness of the Symbolic Execution;
  - and a prototype implementation of the technique and benchmarks.

## 1.2 Higher-Order Stateful Programs

Let us start by defining the stateful higher-order setting in which all work presented herein shall be examined. Programs we shall be considering will be higher-order, meaning that we shall focus on languages with features that manipulate methods of arbitrary order as they would values. In particular, we shall investigate programs with *higher-order methods*, *higher-order store* and *lambda abstractions.* Respectively, these refer to methods that may take one or more methods as arguments, or may themselves return a method; references that may point at methods of arbitrary order; and the ability to dynamically create new methods. The specific syntax and semantics of the languages used will be provided in the next chapter. One can consider these languages to be extensions of a lambda calculus with references, and can be thought of as idealisations of languages such as Python and OCaml. We shall be using the term "method" to emphasize the use of stateful functions, but the terms are interchangeable.

As we shall be focusing solely on static verification approaches, programs considered are allowed to be open with undefined input variables. For instance, consider the following program (where we regard inputs to be open):

```
let main n =
let f = λ x. x + 1 in
assert(f n > n)
```

Statically verifying that the assertion holds requires reasoning about the input variable n, which is undefined. To handle input variables, we shall be analysing programs like main symbolically, and will assign free variables such as n a ***symbolic value***. These symbolic values are special free variables of ground type which act as placeholders for concrete values. This will be made clearer in the chapters that follow.

### 1.2.1 Open and Closed Higher-Order Programs

The techniques considered herein shall all be *symbolic*, meaning that terms can be characterised as being “open” by the presence of symbolic values. To avoid confusion in our terminology, we make a distinction between normal free variables and symbolic values:

> We shall only refer to terms as being “open” in the higher-order sense, that is, higher-order terms are open if they contain free variables of higher-order type, while closed terms are allowed to contain symbolic values but no free variables of higher-order type.

We thus consider two cases herein: what we call ***closed higher-order programs*** and, more generally, ***open higher-order programs***. Intuitively, closed higher-order programs only ever expect a ground-type input at a top level, while open higher-order programs include the general case that allows undefined methods expected as input. To illustrate this, consider the following closed (but ground-type open) program:

```
let mainClosed n =
let f = λ x. x n in
assert(f (λ y. y) == n)
```

in contrast to the following open higher-order program:

```
let mainOpen g n =
let f = λ x. x n in
assert(f g == n)
```

While reasoning about `mainClosed` only requires the symbolic value for `n`, reasoning about `mainOpen` additionally requires the semantics of `g`, which has been left undefined. In other words, `mainClosed` is higher-order by the presence of the higher-order function `f`, whereas `mainOpen` is additionally higher-order in that it expects a function `g`. Here lies the main difficulty in statically reasoning about open programs, that is, they are allowed to call code with undefined semantics. One has to consider the behaviour of an unknown environment that is able to call the main method and provide the program with a method to once again call back said unknown environment. As can be expected, this introduces a primary source of intractability as all programs need to be verified while taking into consideration all possible call contexts in which the main is called, and all the possible behaviours of calling `g` that result from being called in said contexts. Exhaustive generation and enumeration of these contexts will be the focus of Chapter 4, whereas the focus of Chapter 3 is in keeping track of the symbolic higher-order semantics occurring internally in terms.

### 1.2.2 Higher-Order Libraries

So far, programs given in the examples above could all be defined as single terms. That is, they are defined by a main method that defines a term to be evaluated. For instance, consider the program introduced above:

```
let mainClosed n =
let f = λ x. x n in
assert(f (λ y. y) == n)
```

Excluding the boilerplate main method declaration, to statically reason about `mainClosed` we are required to examine the term

```
let f = λ x. x n in
assert(f (λ y. y) == n)
```

where `n` is free. Similarly, the open case requires examining the term

```
let f = λ x. x n in
assert(f g == n)
```

where `n` and `g` are both free. One can see here that symbolic analysis of higher-order programs depends on how we fill in the semantics for the free variables and symbolic values present. Considering that one needs the call contexts for the main method, and also the context for behaviour occurring during the call to `g`, any approach to verifying open higher-order programs has to handle a scenario where the term to evaluate is undefined. That is, the term currently being evaluated might be defined by the environment and is not available as it is part of the unknown context in which the program runs.

Since in general the open case does not care if a term is present to evaluate to start with, we shall be considering the case where the program is simply a set of method definitions to be called in some unknown environment. We call these ***higher-order libraries***. To illustrate this, consider the following higher-order library:

```
import g,h
public f1 = λ x. g x
private f2 = λ x. h (g x)
public f3 = λ x. f2 x
```

where the **`import`** keyword states that `g` and `h` are undefined and are to be provided by the environment, the **`public`** keyword qualifying `f1` and `f3` means that both these methods are available and known to the environment, and the **`private`** keyword means `f2` is for internal use only and is unknown to the environment. Reasoning about programs in a client-library paradigm, where the client is unknown, requires one to consider all possible clients (calling contexts) with which any of the public methods of the library may be called. As with the prior examples, the syntax for libraries here is for illustration.

The actual syntax used in the rest of this thesis will be more precisely presented in the next chapter.

The client-library paradigm can be thought of a generalisation of the open environment problem as it requires the known fragment to exist in an unknown higher-order context to begin with. However, it is an equivalent problem as the client and library are symmetric: a client is equivalent to a library that is currently evaluating a term. Note that in our usage of the client-library paradigm the client encompasses the entire environment, meaning that there cannot be external components to the client besides the library and vice-versa (i.e. they close each other). This makes it sufficient to have a 2-part compositional technique to handle this setting: being able to compose the independent analysis of two components, but not generally for arbitrary many parts, is enough for reachability in our setting. This will be explained in more detail in Chapter 4.

## 1.3 Software Verification and Bounded Verification

In formal software verification, the main question asked is whether a program $M$ satisfies its specifications $\alpha$, written $M \vDash \alpha$. While the question, in a high-level sense, simply asks for a measure of quality for $M$, the breadth of research makes a discussion of every approach infeasible within this introduction. As such, we shall limit the discussion here to what shall be the focus of this thesis, that is, to formal methods that are static, check specifications of correctness in terms of safety, and output counterexamples that are sound. The first point describes techniques that are applied at compile-time, meaning directly on the source code of $M$ without executing it, in contrast to run-time approaches. The second point—correctness—qualifies techniques that use specifications which may be violated by the reachability of errors. In particular, we shall be using specifications defined through assertions, which we range over $\alpha$ and variants. The final point distinguishes *bounded correctness* techniques from *total correctness* approaches. While total correctness is sound when one can prove errors are exhaustively unreachable within every execution of $M$, bounded correctness is sound when one reports all errors that are reachable in $M$ up to a given depth. Finally, it should be noted that we shall only consider mathematically rigorous techniques and any methods we introduce herein will have proofs of soundness attached.

**Remark 1.1.** The difference between pure bug-finding and bounded correctness (bounded verification) is that bug-finding techniques may not guarantee anything besides the feasibility of the bugs reported, whereas bounded verification guarantees that all bugs up to the depth have been reported, with the program proven bounded-correct if no bugs have been found up to the given depth.

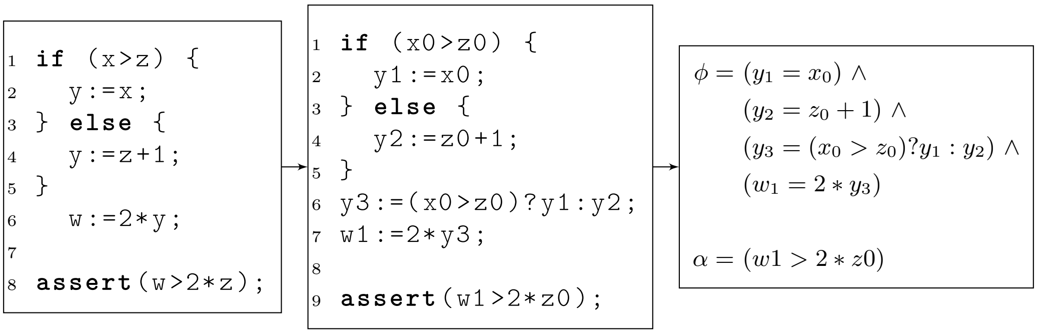
Due to the undecidable nature of exhaustively checking programs, total verification techniques are usually over-approximations, while bounded approaches are underapproximations. In practice, total correctness is too expensive, so real-world tools are restricted to specific properties checked totally, or, most commonly, falling entirely to bug-finding [27]. A common theme in under-approximation techniques is to bound the depth of analysis, which is the main kind of approach we shall be presenting in this thesis. Two widely-used techniques for under-approximation are *Bounded Model Checking* (BMC) and *Symbolic Execution* (SE). While Symbolic Execution is not bounded in principle and can even be used for total correctness [37], in practice, search is often bounded as computation can be infinite.

For our approaches herein, both BMC and SE return counterexamples, and should do so soundly. Additionally, one may consider our approaches to be *model checking* techniques, as we shall be checking the program $M$ indirectly through a model $\phi$ of its behaviour. More precisely, in model checking, a model $\phi$ that captures the semantics of $M$ is exhaustively checked against $\alpha$ for satisfiability, written $\phi \vDash \alpha$, where a violation of $\alpha$ exists in $\phi$ if and only if a corresponding error exists in $M$. While the approach is typically an *unbounded depth* (total-verification) technique based on *finite-state automata* [9] where the behaviour $\phi$ and properties $\alpha$ of $M$ are expressed as finite-state automata, we shall be focusing on bounded SAT/SMT-based verification techniques, where $\phi$ and $\alpha$ are captured by first-order (quantifier-free) formulas. We define an execution of $M$ to be a finite sequence of transitions, called a *trace* or *path*, taken from some initial configuration. This will be made more precise in Chapter 2. We also say $M$ *fails* if an assertion violation is reachable in some path of $M$. Additionally, a *counterexample* of $M$ is either ground-type, that is, an input value that causes $M$ to fail; or higher-order, that is, a trace proving that an error is reachable in the semantics of $M$. This will be made more precise in Chapter 3 and Chapter 4. In the next sections, we shall describe Bounded Model Checking and Symbolic Execution in more detail.

### 1.3.1 Bounded Model Checking

Bounded Model Checking [11] is a model checking technique that allows for highly automated and scalable SAT/SMT-based verification that has been widely and successfully used to find errors in C-like languages since the early 2000s [20, 52, 27, 6]. BMC amounts to bounding the executions of programs by unfolding loops only up to a given bound, and model checking the resulting execution graph. Since the advent of CBMC [20], the mainstream approach additionally proceeds by symbolically executing program paths and gathering the resulting path conditions in propositional formulas which can then be passed on to SAT/SMT solvers. Thus, BMC performs a syntactic translation of program source code into a propositional formula, and uses the power of SAT/SMT solvers to check the bounded behaviour of programs.

Being a Model Checking technique, BMC has the ability to produce counterexamples, which in this case are inputs that lead to the violation of desired properties with the



```
if (x>z) {
  y:=x;
} else {
  y:=z+1;
}
  w:=2*y;

assert(w>2*z);
```

```
if (x0>z0) {
  y1:=x0;
} else {
  y2:=z0+1;
}
y3:=(x0>z0)?y1:y2;
w1:=2*y3;

assert(w1>2*z0);
```

$$\begin{aligned}\phi = {} & (y_1 = x_0) \wedge \\ & (y_2 = z_0 + 1) \wedge \\ & (y_3 = (x_0 > z_0)?y_1 : y_2) \wedge \\ & (w_1 = 2 * y_3) \\ \\ \alpha = {} & (w1 > 2 * z0)\end{aligned}$$

Figure 1.1: Applying mainstream BMC to a C-like program.

corresponding execution traces. While it still suffers from the combinatorial nature of model checking, an advantage of BMC over unbounded techniques is that it avoids the full effect of state-space explosion at the expense of full verification by bounding the length of counter examples considered. Since BMC is often inconclusive if the formula is unsatisfiable, it is generally regarded as a bug-finding or underapproximation technique. It is, however, still a verification technique since programs are exhaustively checked up to the bound, which lets it avoid spurious errors. Empirically, BMC is one of, if not the most effective approach for “shallow" bugs [27, 6], whilst having a major weakness with bugs in deep loops and recursion. Additionally, despite being a bounded technique, it is still possible to prove complete correctness with BMC if bounds for loops and recursion are determinable.

Figure 1.1 illustrates how the mainstream BMC translation might be applied on a C-like language. The approach takes a program $M$ and a bound $k$, and computes the formulas $[\![M, k]\!] = (\phi, \alpha)$ by recording the effect of each command on the store. The translation to SAT, as described by its authors in [20], is done in the following steps.

1. **Control simplification.** All control constructs are translated to `while` loops, `goto` statements, and `if` statements.
2. **Loop unwinding.** `while` loops are globally unwound $k$ times: the body of the loop is duplicated $k$ times and every `while` guard is replaced with an `if` statement; each copy nested within the previous loop body. On the last copy, the original loop condition is negated and added as an assertion called the *unwinding assertion*. This last step is needed to maintain soundness of errors found, as it limits search by excluding errors beyond the bound, which may have been caused by the bounding.
3. **Goto unwinding.** All backward `goto` statements are unwound as with `while` loops. Everything between the `goto` statement and the label it points at is duplicated $k$ times and guarded at the end.
4. **Function call inlining.** All function calls are expanded by replacing each call with its respective function body. Recursive calls are unwound as `while` loops were.
5. **Static Single Assignment (SSA).** The program is then translated to SSA,

where each variable can only be assigned to once; traditionally done by pointer analysis [7, 76]. The SSA transformation can be described as a counter that keeps track of the times a variable $x$ has been assigned, and renames $x$ to $x_n$ at every assignment, where $n$ is the number of previous assignments. Each use of $x$ within its scope is then replaced with $x_n$. Wherever branching occurs, a guarded assignment, also called a $\varPhi$ function, is inserted after the branch.

6. **Building the SAT formula.** The formula is built by accumulating all assignments into a formula via conjunction. This results in a formula that is in *Conjunctive Normal Form* (CNF), meaning that it is a first-order quantifier-free formula of top-level conjunctions of atomic disjunctions. After translating the program into a CNF formula, the negation of all assertions are then added via conjunction. In other words, if a program produces constraints $\phi$, and contains properties $\alpha$, then the final CNF is $\phi \land \neg\alpha$. By doing so, we check for the existence of a configuration in our bounded program such that the property is violated. As this only uses assertions, the method described here will only handle safety properties.

The aim of the BMC translation is to compute the formula $\phi \land \neg\alpha$, which is then fed to a SAT/SMT solver. This formula is explained by the underlying goal to ensure that the execution constraints $\phi$ meet the specifications $\alpha$, that is, to ensure that $\phi \implies \alpha$ holds. To achieve this, we want to show that no counterexample that proves the opposite exists: that $\neg(\phi \implies \alpha)$ does not hold. Since $\neg(\phi \implies \alpha) \iff (\phi \land \neg\alpha)$, we can reduce finding counterexamples to SAT/SMT by proving satisfiability of $\phi \land \neg\alpha$, thus exhaustively checking $M$ up to $k$.

It should be noted that mainstream BMC is a monolithic technique in that it requires all of the analysis to not just fit in memory, but to come from a defined source. While compositional BMC techniques exist [18], the approach still typically requires dependencies to be defined, and is thus not immediately suitable for open higher-order bug finding.

### 1.3.2 Symbolic Execution

Symbolic Execution was conceptually introduced in the mid 1970s [12, 21, 43] as a technique to statically reason about program inputs. While the key concepts have been around for over four decades, SE received renewed popularity in the late 2000s due to significant developments in satisfiability solving [17]. Symbolic Execution amounts to using symbolic values in place of input variables. Given a program $M$, the aim is to explore the *execution tree* (or *computation tree*) of $M$, that is, the set of all concrete execution paths of $M$. This is done by keeping track of a *symbolic environment* ($\sigma$) and a *path condition* ($pc$) for each path in the execution tree. The symbolic environment is a function from variables in the path to symbolic expressions that constrain the value for said variables, while the path condition $pc$, given some $i$th position $p_i$ in the current path, is a first-order quantifier free formula over symbolic expressions that accumulates the conditions that must have been satisfied in order to reach $p_i$.

More precisely, a Symbolic Execution for a term $M$ would classically construct the computation tree of $M$ by case analysis on its topmost redex $r$, which includes the following fundamental cases:

- If $r$ is an assignment $x := M'$ where $M'$ contains variables $y$, then $x$ in $\sigma$ is updated to point at the closure of $\sigma$ over every $y$ in $M'$, i.e. $\sigma := \sigma[x \mapsto \sigma(M')]$.
- If $r$ is a conditional statement `if` $c$ `then` $M_1$ `else` $M_0$, then we split the current path into two paths $P_1$ and $P_0$, where $P_1$ symbolically evaluates term $M_1$ with path condition $pc \wedge \sigma(c)$, and $P_0$ symbolically evaluates $M_0$ with path condition $pc \wedge \neg\sigma(c)$, both using the same symbolic state $\sigma$. $P_1$ and $P_0$ are then added to the set of paths to symbolically explore in order to build the execution tree.

As one might have noticed, these symbolic evaluation cases are similar to evaluation rules typically seen in an operational semantics. In fact, the complete set of symbolic evaluation rules which forms the SE routine should be a sound symbolic abstraction of the concrete semantics. In order to build the execution tree, these symbolic transitions are recursively applied to every path in the set of paths to explore until the terms in the paths cannot be reduced further. Thus, the classical Symbolic Execution routine on a term $M$ produces an execution tree $T_M$ which contains the set of paths symbolically reachable by $M$. Each path $P_i$ in $T_M$ can be seen as a triple $(M_i, \sigma_i, pc_i)$ that holds a final term $M_i$ that cannot be reduced further, the symbolic environment $\sigma_i$ computed to reach $M_i$, and the path condition $pc_i$ that must be satisfiable for $P_i$ to be a valid concrete path of $M$.

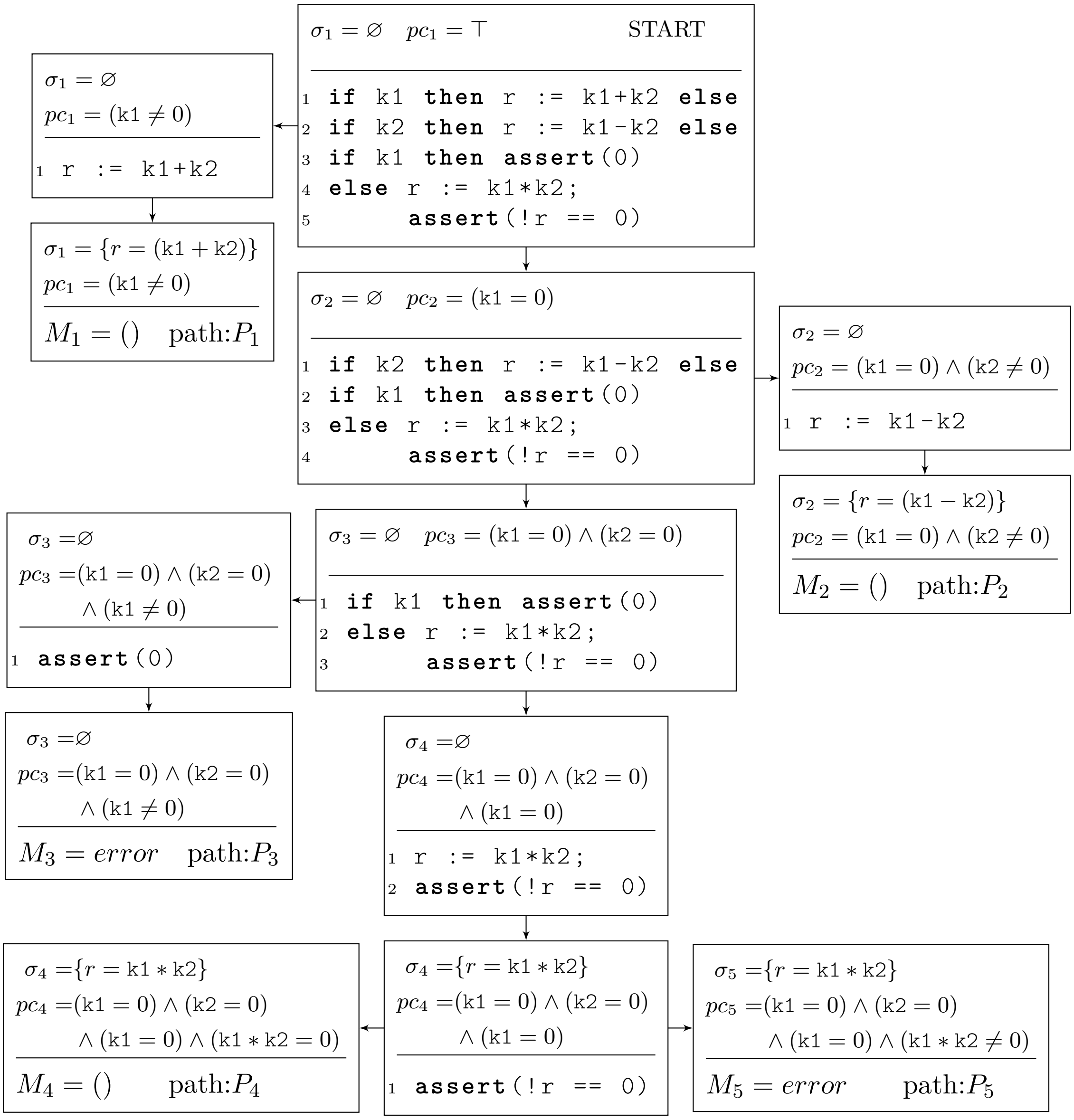
We can observe in the SE routine that all path conditions $pc_i$ are of the form

$$pc_i = \bigwedge_{j=1}^{n} c_j$$

for all $n$ path conditions $c_j$ seen in the process of evaluating $P_i$. Thus, in order to produce the set of valid concrete paths of $M$, we can feed each $pc_i$ to a SAT/SMT solver and keep every $P_i$ where $pc_i$ is satisfiable. A model $\psi \vDash pc_i$ therefore assigns to the input variables the necessary values for $M$ to concretely execute $P_i$. Note that paths may be pruned on the fly by discarding any path produced with an unsatisfiable path condition. Since the number of paths explodes combinatorially on the number of conditional statements reached, pruning paths at the cost of multiple calls to the solver is usually beneficial.

To illustrate the basic SE procedure, consider the following stateful program where `x` and `y` are input variables, `assert(z)` fails if `z` $= 0$, and `if` $c$ `then` $M_1$ `else` $M_0$ evaluates to $M_0$ if `c` $= 0$ and $M_1$ otherwise:

```
if x then r := x+y else
if y then r := x-y else
```



$\sigma_1 = \varnothing \quad pc_1 = \top$
START

```
if k1 then r := k1+k2 else
if k2 then r := k1-k2 else
if k1 then assert(0)
else r := k1*k2;
     assert(!r == 0)
```

$\sigma_1 = \varnothing$
$pc_1 = (\mathtt{k1} \neq 0)$

```
r := k1+k2
```

$\sigma_1 = \{r = (\mathtt{k1} + \mathtt{k2})\}$
$pc_1 = (\mathtt{k1} \neq 0)$
$M_1 = () \quad \text{path:}P_1$
$\sigma_2 = \varnothing \quad pc_2 = (\mathtt{k1} = 0)$

```
if k2 then r := k1-k2 else
if k1 then assert(0)
else r := k1*k2;
     assert(!r == 0)
```

$\sigma_2 = \varnothing$
$pc_2 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} \neq 0)$

```
r := k1-k2
```

$\sigma_2 = \{r = (\mathtt{k1} - \mathtt{k2})\}$
$pc_2 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} \neq 0)$
$M_2 = () \quad \text{path:}P_2$
$\sigma_3 = \varnothing \quad pc_3 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} = 0)$

```
if k1 then assert(0)
else r := k1*k2;
     assert(!r == 0)
```

$\sigma_3 = \varnothing$
$pc_3 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} = 0) \wedge (\mathtt{k1} \neq 0)$

```
assert(0)
```

$\sigma_3 = \varnothing$
$pc_3 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} = 0) \wedge (\mathtt{k1} \neq 0)$
$M_3 = error \quad \text{path:}P_3$
$\sigma_4 = \varnothing$
$pc_4 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} = 0) \wedge (\mathtt{k1} = 0)$

```
r := k1*k2;
assert(!r == 0)
```

$\sigma_4 = \{r = \mathtt{k1} * \mathtt{k2}\}$
$pc_4 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} = 0) \wedge (\mathtt{k1} = 0)$

```
assert(!r == 0)
```

$\sigma_4 = \{r = \mathtt{k1} * \mathtt{k2}\}$
$pc_4 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} = 0) \wedge (\mathtt{k1} = 0) \wedge (\mathtt{k1} * \mathtt{k2} = 0)$
$M_4 = () \quad \text{path:}P_4$
$\sigma_5 = \{r = \mathtt{k1} * \mathtt{k2}\}$
$pc_5 = (\mathtt{k1} = 0) \wedge (\mathtt{k2} = 0) \wedge (\mathtt{k1} = 0) \wedge (\mathtt{k1} * \mathtt{k2} \neq 0)$
$M_5 = error \quad \text{path:}P_5$

Figure 1.2: Execution tree constructed by applying classic SE.

```
if x then assert(0)
else r := x*y;
     assert(!r == 0)
```

Figure 1.2 is the execution tree produced by applying a classic SE routine to the program above. On the execution tree we can observe five paths: $P_1$ to $P_5$, each with their corresponding term $M_i$, symbolic state $\sigma_i$, and path condition $pc_i$. On this tree we can see that there are two paths that lead to errors: $P_3$ and $P_5$. Checking their corresponding path conditions, we would know that both $pc_3$ and $pc_5$ are unsatisfiable, meaning that neither $P_3$ or $P_5$ is reachable. Since no other errors are reachable in the execution tree, and the tree exhaustively lists all possible paths, we have successfully verified the program for total correctness.

Using Symbolic Execution, model checking of a term $M$ can be achieved by checking the following formula for satisfiability:

$$PC = \bigvee_{pc \in Error(T_M)} pc$$

where $T_M$ is the execution tree of $M$ computed through SE and $Error(T_M)$ is the set of paths $P_i$ where $M_i = error$. Thus, $PC$ holds a disjunction of all path conditions that may reach an assertion violation, and any satisfying assignment $\psi \vDash PC$ proves that an error is reachable in $M$. Note that we present the general idea here, but variants that achieve the same effect still fall under Symbolic Execution. For instance, on-the-fly path pruning allows one to verify $M$ by checking whether $Error(T_M)$ is empty. Similarly, variants that accumulate program behaviour separately from assertions may end up checking $pc \wedge \neg\alpha$ for satisfiability instead.

### 1.3.3 State Merging: BMC vs BSE

It is easy to imagine a procedure for bounded-depth Symbolic Execution where each path is expanded up to a fixed number of steps determined by some bounding condition. Taking this idea of a Bounded SE (BSE) into consideration, BMC and SE appear to be equivalent as both amount to bounded symbolic explorations of program executions. Still, while the techniques are not mutually exclusive and extensions of each may in fact blur the difference between them, in their classic presentation they can be fundamentally distinguished by whether they perform state merging. On one end, the mainstream BMC approach performs full state merging, meaning that it always inserts a $\Phi$ function after branching. This results in a string of merged states that captures the program behaviour monolithically, and is a result of the unoptimised SSA transformation used in the original technique. In other words, taking $M$ as the input term, BMC encodes in the behaviour formula $\phi$ the entire execution tree of $M$ in the form of a control-flow graph with fully merged states (i.e. the number of paths is at most two at any point), whereas

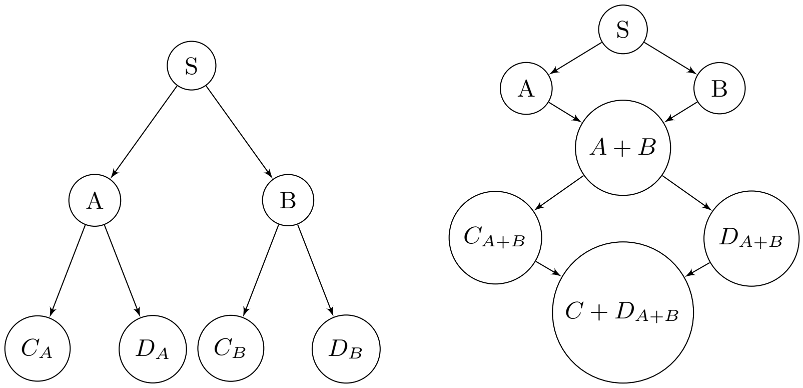


Figure 1.3: State-space graph of SE (left) and BMC (right) on the same program.

SE classically explores every path of the execution tree separately and independently. To illustrate this distinction, consider the following program consisting of two conditional statements chained one after the other:

```
(if x then A else B);
(if y then C else D)
```

Performing BMC and SE independently on this program, the resulting state space for each is as shown in Figure 1.3. In these state-space graphs we write $X_Y$ for an expansion on tern $X$ within the context having computed $Y$, where one can think of $Y$ as the trace computed so far. We can observe on the state-space graph for SE that individual exploration of each path results in an exponential increase in the size of the state space, resulting in an exponential number of total paths explored with respect to the depth of the tree, and exponential on the branching factor of the program; in this case 2. In contrast, BMC always merges the paths, and thus keeps a constant upper bound on the number of paths to consider equal to branching factor of the program—again 2—which results in a state space that is linear on the depth of exploration. Comparing the state-spaces, we can see that SE will eventually have exponentially many more instances to explore than BMC, but each instance is smaller compared to BMC. This is due to the fact SE keeps track only of the information seen so far in each path per path, whereas BMC carries around a single big instance that keeps track of all the traces seen so far.

In terms of the impact of state merging on modularity, the mainstream approach of always merging states makes BMC a monolithic technique by nature, while SE is more easily modular due to path independence. An example of the modularity that path independence allows can be observed in [64], where the authors parallelise SE by concurrently running the exploration and constraint solving of each path. In terms of performance, state merging creates trade-offs in memory usage, compilation speed, solving difficulty, and total execution time. For memory, a bounded branching factor provides BMC with a state space that is linear in the exploration depth, which results in significantly lower memory usage during exploration compared to SE. This in turn makes

BMC faster at compiling from source to a SAT/SMT instance [79]. The advantage in faster compilation, however, may be offset by a SAT/SMT instance built that is harder to solve and, in extreme cases, may not even fit in memory. In contrast, while SE has exponentially many instances, each is much smaller and thus easier to solve. However, the smaller paths in SE also involve redundancy that is not present in BMC. More specifically, given a program with a branch at position $p_i$, the prefix $p_1 \ldots p_{i-1}$ will be shared by all paths up until they diverge at $p_i$. This makes paths in SE highly redundant, which adds repeated clauses that accumulate over exponentially many instances. As such, state merging is still desirable in SE, and can be seen in [79], where the authors perform SE with *type-driven state merging* in order to improve compilation speed and reduce memory usage. However, full state merging, while more memory efficient, comes at a potential cost to total execution time. Explicit exploration of every path means that SE will encounter every opportunity where concretisation is possible, whereas BMC avoids them all. Compared to concrete execution, symbolic evaluation may be several orders of magnitude slower. For instance, [81] records the Symbolic Execution tool KLEE being three orders of magnitude slower than native execution for many real-world programs. For this reason, concretisation is also desirable and, while a direct extensive comparison is not in the scope of this thesis, the trade-offs in performance suggest that a balance in state-merging is desired. These developments of BMC and SE, however, further blur the line between them, and may lead to them being considered ends of the same spectrum. In this thesis, however, we shall consider BMC and SE distinct techniques as they fundamentally approach and encode problems differently.

## 1.4 Game Semantics

Game Semantics [5, 31] is an approach to formal semantics that, when used to model computation, models the interaction between a term and its environment as a game between two players. As a model of computation, it has been successfully applied to programming language semantics [30], particularly as a denotational semantics, and has become a fundamental concept in the Theory of Computation. Game Semantics was first established as a key concept in programming language semantics in the early 1990s, when it was used independently by three teams of researchers [4, 40, 59] to show *full abstraction* of PCF (Programming with Computable Functions) [67]—a simply-typed $\lambda$-calculus with basic arithmetic and higher-order recursion, alike the functional fragment of languages presented herein. Its ability to model an undefined environment explicitly distinguishes Games Semantics from other models computation [2] and, most notably, provides the means to model higher-order interaction, which is of particular interest when higher-order references are involved [3]. This makes Game Semantics a powerful concept for language semantics, which has lead to its use in software verification on multiple occasions [53, 25, 10, 36].

In Game Semantics for computation, two players are involved: a Proponent, in this case representing the program we want to verify, and an Opponent, depicting the environment in which the program runs. Interaction between the two players is represented as a sequence of *moves* made alternately by each player. As the theory can be formulated operationally in terms of a trace semantics for open terms [41, 47, 33], we shall be presenting games in an operational setting and shall refer to these sequences of moves as "traces". Moves in these games define the capability of each player; that is, the set of moves available to each player at any point in the game exhaustively captures all possible ways in which a given program can interact with its environment and vice versa. This capability is succinctly presented by two sets of actions available to each player: Questions, the ability to query the other player, and Answers, the ability to reply to a query of the other player. In addition to playing moves that interact with the other player, each player may also compute internally until action from the other player is required. Note that two-player games suffice to adequately model multi-party interaction [2]: players are symmetric in that, from the point of view of each player in a multi-party game, the behaviour of every other player can be captured by a single hypothetical player where any interaction between opponents is indistinguishable from an unobservable action the hypothetical player is playing internally.

In this thesis, we shall be using Game Semantics to model higher-order programs; more specifically, we shall be modelling the undefined context of open higher-order programs previously described. The use Game Semantics is easily explained by its ability to model a program in isolation from a concrete context, which is necessary for higher-order libraries. Higher-order interaction shall be modelled via two-player games as sequences of computational *moves*—representing method calls and returns—between the modelled program and its hypothetical environment. The power of the technique lies in its use of combinatorial conditions to precisely allow only those game plays that can be realised by including the program in a concrete environment. We shall discuss these games more formally in Chapter 4, where we present the games we devised for higher-order libraries and construct a symbolic variant thereof to—via Symbolic Execution—symbolically check said libraries for errors.

## 1.5 Related Work

In this section we mention existing related research, extant techniques that tackle programs similar to those in our setting, and provide a comment on the originality and innovation of our work in comparison to other approaches. We shall start with work related to closed higher-order verification, and continue with the general case next.

### 1.5.1 BMC and Closed Higher-Order Verification

Defunctionalisation [69] is a transformation that replaces higher-order functions with special first-order application functions at compile-time. In our BMC approach, we shall use an equivalent transformation in the operational semantics that replaces method application with first-order calls to a repository. While defunctionalisation is common in symbolic evaluation, and is often paired with points-to analysis, to our knowledge, it has not been used to translate entire general higher-order programs into SAT/SMT-based BMC encodings. As such, we believe we are contributing a sound alternative to model checking closed higher-order terms.

Being a common technique, there exist several BMC encodings similar to ours. For example, [29, 26, 24] are bounded approaches based on relational logic that verify Java programs using SAT/SMT solvers. Since they are applied to Java, however, and particularly prior to Java 8, these approaches do not cope with terms and store of arbitrary order. In every case, methods are inlined statically, which is not always possible with lambda abstractions. In [24] the authors define a method application that selects method bodies based on their type. This is similar to our concept of exhaustive method application that we shall be presenting later, but is only applied in [24] to resolve dynamic dispatch. Additionally, in contrast to all bounded approaches mentioned above, the technique we present shall be more general, as we shall handle all higher-order behaviours occurring internally within the term, including dynamic method creation and higher-order store.

Verification tools for closed higher-order programs that are based on a direct syntactical BMC encoding are less common. Instead, two main techniques followed are Higher-Order Recursion Schemes (HORS) modelling [44, 62], and Symbolic Execution [13, 38, 42]. In the first category, tools related to MoCHi [45, 72] perform full verification of purely functional OCaml programs by translating them into higher-order recursion schemes checked with tools specialised for checking HORS. More specifically, MoCHi is based on predicate abstraction and CEGAR using refinement types to construct higher-order terms that can be checked with HORS model checkers. While BMC approaches and MoCHi are incomparable, since MoCHi performs total correctness verification, an important difference in scope exists in that MoCHi aims to handle only pure programs, whereas we additionally cover stateful programs. In the second category, tools like Rosette [79] and Rubicon [56] respectively implement Bounded Symbolic Execution for Racket and Ruby, and thus check functional and imperative higher-order programs for correctness. This makes these approaches similar to our technique in application, and a comparison shall be made to put our BMC technique into perspective relative to extant BSE approaches. Another approach worth noting is presented in [58], which performs a symbolic execution for Racket using *software contracts* to define correctness. Their symbolic semantics allows evaluation of *arbitrarily open* higher-order programs by generating *relatively complete* higher-order counterexamples. This approach uses

Symbolic Execution to tackle a more general problem than closed programs, as it is handling the open higher-order case. We shall discuss this in more detail in the next section, under work related to open higher-order verification. From these approaches, we shall choose MoCHi and Rosette as representatives for a comparison in Chapter 3. Particularly, Rosette was selected because Racket fits the setting of a stateful higher-order language written in a functional style, and additionally allows us to implement our bounding mechanism to use with their symbolic execution of Racket, which provides a more direct comparison of the underlying techniques.

Intensionally, tools based on CBMC [52, 52, 68, 70] are inherently similar to our BMC encoding and procedure since we are taking inspiration from the CBMC translation in first place. However, being primarily applied to C-like languages means BMC encodings are less often used to tackle higher-order behaviours. In fact, CBMC itself partially tackles the problem of higher-order control flow in a fashion not unlike our own exhaustive method application, but is set apart by dynamic method creation, which CBMC does not handle. Given the fact C and C++ syntax does not encourage higher-order programming as naturally as functional languages, the lack coverage for higher-order behaviours in C-like languages is self-explanatory. In contrast, tools based on SE are more often used for functional languages and are able to produce the most extensionally similar implementations. This can be explained by how SE explores paths independently: at the expense of memory usage, SE encodes control-flow explicitly, which concretises the flow of higher-order terms (inside each path) and facilitates reasoning about higher-order terms. While SE may seem like a more obvious choice than BMC for higher-order terms, it is not strictly so due to the theoretical advantages in memory consumption and faster compilation that BMC often enjoys. A more detailed comparison of our BMC encoding and SE will be presented in Chapter 3, when we compare Rosette to our own approach.

### 1.5.2 Games, SE and Open Higher-Order Verification

In software verification, Game Semantics techniques are traditionally applied to the problem of program equivalence. The algorithmic games approach reduces program equivalence to language equivalence in a decidable automata class [32, 1]. Equivalence tools can be used for reachability but, as they perform full verification, they can only cover lower-order recursion-free language fragments to remain decidable. For example, the Coneqct [53] tool can verify the simplified DAO attack [28], a problem that requires reasoning about an unknown environment, but cannot check higher-order or recursive functions (e.g. the “file lock" and “flat combiner" examples in Chapter 2), and operates on integers concretely. Related to our approach is also Symbolic GameChecker [25], which performs symbolic model checking by using a representation of games based on symbolic finite-state automata. The tool works on recursion-free Idealized Algol with first-order functions, which supports only integer references. On the other hand, it is complete (not bounded) on the fragment that it covers.

Being a symbolic execution for games, our approach is related to tools such as KLEE [16] and jCUTE [74]. These are able to find first-order counterexamples, but are unable to produce higher-order traces that require a model for unknown code. Particularly, KLEE and jCUTE are only able to handle symbolic calls provided these can be concretised, which is insufficient to handle the environment problem; concretisation of calls is often impossible with libraries, since control may be passed to unknown code. Again, CBMC [20, 46] can be compared our technique considering it partially handles calls to unknown code by returning a non-deterministic value to such calls. This is equivalent to a game where the environment may only play by answering questions. This restriction allows CBMC to find some bugs caused by the environment, but misses errors that arise from transferring control to the environment, such as reentrancy and the side effects involved. As with before, since CBMC is generally applied to C-like languages, the mainstream BMC approach also misses bugs involving disclosure of names, e.g. the “file lock" example in Chapter 2, which involves a second-order method.

A related field of study is that of verifying *smart contracts* in the Ethereum Platform, which provides a practical use for higher-order techniques that handle errors related to unknown code, particularly focussing on reentrancy. Considering the immutable nature of smart contracts—meaning they only have one attempt at publishing bug-free code—and considering the high monetary risk involved, automated techniques for these are highly desirable. However, the semantics of the Ethereum Virtual Machine involves higher-order behaviours that result in control being passed to undefined clients. As such, preventing reentrancy due to undefined semantics is a key focus of smart contract verification. Tools like Oyente [51] and Majan [61] use pre-defined patterns and symbolic execution to find bugs related to transaction order and reentrancy, but are not sound or complete. ReGuard [50] is a tool that finds sound reentrancy bugs using a fuzzing engine to generate random transactions to check against a reentrancy automaton. In principle, it may detect reentrancy faster than direct symbolic execution due to the nature of fuzzers [81], but, in exchange, is incomplete even in a bounded setting. More closely related to our approach, [34] describes an implementation in the EtherTrust static analyser of a small-step semantics that accounts for reentrancy. They consider the possibility of an *unknown* contract $c?$ calling the *known* contract $c*$ at each higher call level since $c?$ can only modify the internal state of $c*$ by reentering it. This concept can be generalised in our game semantics as *abstract* and *public* names calling each other, which we have shown here to be a sound way to generate calling contexts. It is possible EtherTrust implements a form of game semantics, however, as a tool targeting smart contracts, the focus is on modelling reentrancy, while we handle the full range of higher-order behaviours.

Finally, also mentioned previously under work related to the closed case, [58] and its development [57] describe an approach that is very closely related to our games-based technique. The problem presented by the authors is that of verifying Racket contracts in a higher-order (stateful in [57]) setting similar to ours. Software contracts generalise higher-order pre and post conditions, which allows one to express program invariants and thus specify safety. To verify said contracts, the authors develop a symbolic semantics for higher-order terms to perform Symbolic Execution of open higher-order programs. This is further developed in [57] for total verification, and to cover stateful programs, via a abstract interpretation of their symbolic semantics though finitisation. Their symbolic semantics is based on what they call *demonic context* in prior work [78], which handles a potentially unknown environment through a method application case *AppOpq*. It does this by either returning a symbolic value to the call, or by performing a call to a known method (given an unknown argument) while inside some unknown context. It thus approximates all the possible behaviours of the environment and appears to be equivalent to the role of the opponent in our games. Considering this, the approach the authors present may seem extensionally similar to ours, since both amount to Symbolic Execution of a higher-order symbolic semantics that handles an unknown environment in some way. However, it should be noted that the techniques have been developed in complete independence from each other, and are based on different theoretical foundations and are thus very different intensionally. Additionally, we shall be handling assertions that specify correctness within the term, whereas [57] uses contracts, which can be seen as specifying correctness at the level of types. As for theoretical results, in [58] the authors prove soundness and completeness but do not prove completeness in [57] when state is added. In contrast, our approach is proven sound and complete in the higher-order stateful setting. We shall provide in Chapter 4 an empirical comparison of the techniques based on the implementations of each.

# Chapter 2

# Motivating Examples and Background Definitions

In this chapter we provide background definitions that shall be useful in the chapters that follow. This includes motivating examples for open and closed program verification, and the syntax and operational semantics for HOLi and HORef.

## 2.1 Reasoning About Higher-Order Programs

We start this chapter by illustrating some of the nuances involved in reasoning about higher-order programs. We shall start with what we call closed higher-order programs, that is, programs which involve higher-order computation internally, but do not expect higher-order input, and follow up with open higher-order programs, where the focus is on libraries.

### 2.1.1 Closed Higher-Order Examples

The main challenge associated with statically analysing (closed) higher-order terms is that it is easy to loose track of the method body to use in method application. This is a primary question in *Control Flow Analysis*, particularly for functional languages [60]. To illustrate this, we shall provide a series of examples, written in sugared HORef, starting with the simplest case. We will provide a formal description of HORef later in this chapter.

Consider an example taken from [60]:

```
let f = λ x. x 1 in
let g = λ y. y+2 in
let h = λ z. z+2 in
```

```
(f g) + (f h)
```

It is a closed program that defines a higher-order method `f`. The challenge in reasoning about `f` is that we cannot immediately tell what the body of `x` is. In isolation, analysis of `f` becomes that of open higher-order programs. However, within a closed higher-order term like this one, `f` is defined in a context that also defines its usage, which is all we need to reason about `f`. In this example, we can tell from `(f g) + (f h)` that `f` can only ever transfer control to the bodies of `g` and `h`. Indeed, the precise body to choose is known at the location where it shall be used.

However, programs we are interested in can be stateful and thus are allowed to use higher-order references. Consider the following program:

```
let f = λ x. x 1 in
r1 := λ y. y+2;
r2 := λ z. z+2;
(f (!r1)) + (f (!r2))
```

This time, the challenge involves keeping track of references `r1` and `r2`. Here, the term states that `f` is may only be applied to `r1` and `r2`, which are not modified in the term, so the analysis proceeds by simply following the added layer of indirection. However, while superficially similar to the previous problem, the layer of indirection added by references introduce a challenge, as these may change dynamically (at runtime) and without explicit record of the change. For example, consider the following program:

```
let f = λ x. x 1 in
let g = λ (). (r := λ y. y+2) in
let h = λ (). (r := λ z. z+2) in
(g (); f (!r)) + (h (); f (!r))
```

Here, a layer of indirection is added in the side effects of methods `g` and `h`. The analysis of `x` thus now depends on that of `!r`, which in turn depends on whether it is `g ()` or `h ()` which appears immediately before `!r`. To handle these cases, we combine standard techniques in *Data Flow Analysis* [60] with a nominal presentation of defunctionalisation. We shall describe this in more detail in Chapter 3. In particular, we will make use of *Static Single Assignment* (SSA) and a simple *Points-to Analysis* [7] to keep track of the flow of higher-order terms.

One might have noticed that all the examples given in this section so far are concrete, meaning, they can be simply executed. In contrast, the "closed" higher-order programs we shall be considering are allowed to be open with symbolic values (ground-type free variables). It is this addition of symbolic values to higher-order terms which will be making up the bulk of the challenge. To illustrate this, consider the following program:

```
let f = λ x. x 1 in
let f' = if i then λ y. y + 2
              else      λ z. z + 3 in
```

```
f f'
```

The program defines a method `f'` whose body depends on a ground-type free variable `i`, which, intuitively, can be considered an input value to be provided by the environment. This time, looking at the term is not sufficient to tell, at the location where `f'` is called, which body to choose; in fact, without a concrete value for `i`, it is impossible to tell in general. Thus, all bodies which may be bound to `f'` must be exhaustively considered, that is, when evaluating `x 1`, a path for $\lambda$y.y + 2 and one for $\lambda$z.z + 3 must be explored. We shall be calling this case ***exhaustive method application***. As can be expected, exhaustive method application contributes a significant source for combinatorial explosion, which is exacerbated with conditional nesting. Take, for instance, the following program:

```
let f = λ x. x 1 in
let f' = if i1 then λ y. y + 2
         else
         if i2 then λ z. z + 3
         else
         if i3 then λ w. w + 4
         else
         λ v. v + 5  in
f f'
```

Here, $\mathtt{f}'$ can take four possible bodies, the choice of which depends on symbolic values `i1`, `i2` and `i3`. With no insight on these symbolic values, the simplest solution is to branch on every method seen so far that matches the type of $\mathtt{f}'$, which, naturally, does not scale well. As shall be described in Chapter 3, it is here that Data Flow Analysis makes the biggest impact.

Finally, to illustrate the approach to be presented in Chapter 3, let us consider a simple program that showcases the expressiveness of closed higher-order stateful programs:

```
1 let f = λx.λg.λh.if x then g else h
2 in
3 r := f n (λx.x-1) (λx.x+1)
4 assert(!r n >= n)
```

where `n` is an input variable. As a BMC technique, the aim is to produce a single formula that tells us if any assertion violations are reachable. Thus, the main interest is in whether line 4 holds, where, because of the challenges previously discussed, it is not immediately obvious which body to use when dereferencing `r` in the assertion. Let us start by transforming the code according to the semantics of let bindings.

```
1 r := if n then (λx.x-1) else (λx.x+1)
2 assert(!r n >= n)
```

Using defunctionalisation, the assignment in line 1 can now be facilitated with the help of a return variable `ret` and method names `m1` and `m2` for $\lambda$x.x − 1 and $\lambda$x.x + 1 respectively, which we shall remember externally.

```
let ret = if n then m1 else m2
r := ret
assert(!r n >= n)
```

Our use of the return variable `ret` to keep track of intermediate operations is an addition to the mainstream BMC approach. In C-like languages, programs are typically defined by a chain of commands. That is, given commands $C_1$ and $C_2$, a program $C_1; C_2$ chains the commands to occur one after the other. In a functional setting, this is equivalent to running two terms $C_1$ and $C_2$ of unit type by chaining the terms in a let binding that forgets the about return of the first term (i.e., `let _` $= C_1$ `in` $C_2$). As such, in mainstream BMC, the return of each command in a chain is not usually recorded. In contrast, a functional setting (with let bindings) like ours requires keeping track of the result of each let binding, which is where `ret` comes in.

Looking at the program above, the main challenge is now to decide how to symbolically dereference `r`. In this case, the best we can do is to match it with all existing methods that match the type, namely `m1` and `m2`. We thus have two paths to consider:

- if `ret` = `m1`:

  ```
  let ret' = m1 n in
  assert(ret' >= n)
  ```

- if `ret` = `m2`:

  ```
  let ret' = m1 n in
  assert(ret' >= n)
  ```

Substituting in the bodies for each method we have:

- if `ret` = `m1`:

  ```
  let ret' = (n - 1) in
  assert(ret' >= n)
  ```

- if `ret` = `m2`:

  ```
  let ret' = (n + 1) in
  assert(ret' >= n)
  ```

We can now read off the following formula to check for the falsity of the assertion:

$$(ret' < n) \wedge (r = m1 \Rightarrow ret' = n - 1) \wedge (r = m2 \Rightarrow ret' = n + 1) \wedge (r = ret) \\ \wedge (n <= 0 \Rightarrow ret = m1) \wedge (n > 0 \Rightarrow ret = m2)$$

The formula above is satisfiable, e.g., for $n = 0$, proving that the code is not safe.

These ideas underpin the core of our BMC procedure, which is presented in Chapter 3 and proven sound and correct. The language we shall examine, called HORef is described later in this chapter. Our Points-to Analysis algorithm is also presented in Chapter 3, as an optimisation necessary for scalability.

With the challenges involved in analysis of closed higher-order programs stated, let us now continue with the case for open higher-order programs.

### 2.1.2 Open Higher-Order Examples

So far, we have been looking at programs in a closed higher-order symbolic setting. Let us now look at a few more examples, this time considering a fully open setting. The key challenge in reasoning about open higher-order terms is that we have to model the interaction between the term and its unspecified environment. As such, we must generate all possible contexts in which the term can be called. In a stateful setting like ours, this is especially important as higher-order stateful interaction leads to a range of subtle behaviours. We will be looking at libraries that exhibit errors due to high-order behaviours, for which we provide four examples: a simplified version of the DAO attack [28], a file lock example, a double deallocation example, and an unsafe implementation of flat-combining. Again, we choose a presentation using higher-order libraries as reasoning about these requires, by definition, considering a client which is left unspecified. The libraries presented below will be written in a sugared form of HOLi: a language we shall be using to study bounded verification of higher-order stateful libraries. We provide a formal description of this language in the next section.

To showcase the importance and challenges presented by the environment problem, following is a simplified implementation of "the DAO" smart contract, a failed decentralised autonomous organisation on the Ethereum blockchain platform. It should be noted that, while the semantics of Solidity is much more involved than the simplification presented here, the example still captures the control-flow of the original attack, and is still beyond first-order tools and closed higher-order tools.

```
import send:(int → unit)
int balance := 100;

public withdraw (m:int) :(unit) = {
  if (not (!balance < m)) then
    send(m);
    balance := !balance - m;
    assert(not(!balance < 0))
  else ()
};
```

While we are not going to focus specifically on smart contracts, the DAO is presented here for illustration as a simple and practical example of open higher-order code. In the DAO, a reentrancy bug in an external call, modelled here by the send method provided by the environment, caused damages worth over $100m and resulted in a fork that split the Ethereum platform [28]. Reentrancy in higher-order stateful programs is a primary source of errors due to the nature of side-effects dynamically affecting the conditions in which methods can be called. In this example, when the `send` method is called, the environment is allowed to take control and call any method in the library. If a client were to call `withdraw` within its `send` method, the recursive reentrant call would drain all the funds available, which is simulated in this example by a negative balance. This happens because the method is manipulating a global state, and is updating it after the external call. Thus, the check for sufficient funds is accessed before update and never fails. In Chapter 4 we model this kind of higher-order interaction as a sequence of moves, where the environment and library are described by an opponent and proponent respectively. Any resulting sequence of moves is a trace providing a counter example.

Running the example in HOLiK, our implementation of the symbolic game semantics for HOLi in the $\mathbb{K}$ Semantic Framework [71]—which shall be presented in Chapter 4—the following minimal symbolic trace is automatically found:

$$\begin{aligned} &\mathit{call}\langle \mathit{withdraw}, x_1\rangle \cdot \mathit{call}\langle \mathit{send}, x_1\rangle \cdot \mathit{call}\langle \mathit{withdraw}, x_2\rangle \\ &\quad \cdot \mathit{call}\langle \mathit{send}, x_2\rangle \cdot \mathit{ret}\langle \mathit{send}, ()\rangle \cdot \mathit{ret}\langle \mathit{withdraw}, ()\rangle \cdot \mathit{ret}\langle \mathit{send}, ()\rangle \end{aligned}$$

where $x_1$ is the original call parameter, and $x_2$ is the parameter for the reentrant call, satisfiable with values $x_1 = 100$ and $x_2 = 1$. A fix would be to swap line 6 and 7, to update internal state before passing control.

Following is our file lock example, which simulates a scenario where the library makes it possible for the client to update a hypothetical file without first reacquiring the lock for it. The library contains an empty private method `updateFile` that simulates file access. The library also provides a public method `openFile`, which locks the file, allows the user to update the file indirectly, and then releases the lock.

```
import userExec :((unit → unit) → unit)
int lock := 0;

private updateFile(x:unit) :(unit) = { () };
public openFile (u:unit) :(unit) = {
  if (!lock) then () else (lock := 1;
  let write =
    fun (x:unit) :(unit) →
    (assert(!lock); updateFile())
  in
  userExec(write);
  lock := 0)
};
```

The bug here is that `openFile` creates a `write` method, which it then passes to the client, via `userExec(write)`, to use whenever they want. This provides the client indirect access to the private method `updateFile`, which it can call without first acquiring the lock. Indeed, running this example in HOLiK we obtain the following minimal trace:

$$\begin{aligned}&\mathit{call}\langle \mathit{openFile}, ()\rangle \cdot \mathit{call}\langle \mathit{userExec}, m_2\rangle \cdot \mathit{ret}\langle \mathit{userExec}, ()\rangle\\ &\quad\cdot \mathit{ret}\langle \mathit{openFile}, ()\rangle \cdot \mathit{call}\langle m_2, ()\rangle\end{aligned}$$

where $m_2$ is the method *name* generated by the library and bound to the variable `write`. This example serves as a representative of a class of bugs caused by revealing methods to the environment, a higher-order problem, in this case involving the second-order method `userExec` revealing $m_2$.

Next, we simulate double deallocation using a global reference `addr` as the memory address. The library defines private methods `alloc` and `free` to simulate allocation and freeing. The empty private method `doSthing` serves as a placeholder for internal computation that does not free memory.

```
1  import getInput :(unit → int)
2  int addr := 0; // 0 means address is free
3
4  private alloc (u:unit) :(unit) = {
5    if not(!addr) then addr := 1 else ()
6  };
7  private free (u:unit) :(unit) = {
8    assert(!addr); addr := 0
9  };
10 private doSthing (i:int) :(unit) = { () };
11 public run (u:unit) :(unit) = {
12   alloc();
13   doSthing(getInput ());
14   free()
15 };
```

The error occurs in line 13, which calls the client method `getInput`. This passes control to the client, who can now call `run` again, thus causing `free` to be called twice. Again, executing the example in HOLiK, we obtain the following trace:

$$\begin{aligned}&\mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle \mathit{getInput}, ()\rangle \cdot \mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle \mathit{getInput}, ()\rangle\\ &\quad\cdot \mathit{ret}\langle \mathit{getInput}, x_1\rangle \cdot \mathit{ret}\langle \mathit{run}, ()\rangle \cdot \mathit{ret}\langle \mathit{getInput}, x_2\rangle\end{aligned}$$

As with the DAO attack, this is a reentrancy bug.

Finally, we have an unsafe implementation of a flat combiner. The library defines two public methods: `enlist`, which allows the client to add procedures to be executed by the library, and `run`, which lets the client run all procedures added so far. The higher-order global reference `list` implements a list of methods.

```
private empty(x:int) : (unit) = { () };
```

```
 2 fun list := empty;
 3 int cnt := 0; int running := 0;
 4
 5 public enlist(f:(unit → unit)) :(unit) = {
 6   if (!running) then ()
 7   else
 8     cnt := !cnt + 1;
 9     (let c = !cnt in
10      let l = !list in
11      list := (fun (z:int) :(unit) → if (z == c) then f() else l(z)))
12 };
13 public run(x:unit) :(unit) = {
14   running := 1;
15   if (0 < !cnt) then
16     (!list)(!cnt);
17     cnt := !cnt - 1;
18     assert(not (!cnt < 0));
19     run()
20   else (list := empty; running := 0)
21 };
```

The bug here is also due to a reentrant call in line 16. However, this is a much tougher example as it involves a higher-order reference `list`, a recursive method `run`, and a second-order method `enlist` that reveals client names to the library. With HOLiK, we obtain the following minimal counterexample:

$$\begin{aligned} &\mathit{call}\langle \mathit{enlist}, m_1\rangle \cdot \mathit{ret}\langle \mathit{enlist}, ()\rangle \cdot \mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle m_1, ()\rangle \\ &\quad \cdot \mathit{call}\langle \mathit{run}, ()\rangle \cdot \mathit{call}\langle m_1, ()\rangle \cdot \mathit{ret}\langle m_1, ()\rangle \cdot \mathit{ret}\langle \mathit{run}, ()\rangle \cdot \mathit{ret}\langle m_1, ()\rangle \end{aligned}$$

where $m_1$ is a client name revealed to the library. In the trace above, `enlist` reveals the method $m_1$ to the library. This name is then added to the list of procedures to execute. In `run`, the library passes control to the client by calling $m_1$. At this point, the client is allowed to call `run` again before the list is updated.

We now proceed with the syntax and semantics of the languages used.

## 2.2 HOLi: A Language for Higher-Order Libraries

Since closed higher-order terms are included in open higher-order terms, we start by introducing a language for open higher-order programs. We define HOLi, a language for higher-order libraries with higher-order store. Libraries defined in HOLi are collections of method definitions to be used by an external client, which in turn may require the client to provide definitions for external methods. HOLi is a stateful language in that it contains higher-order global store, that is, store accessible from any point in the

$$
\begin{array}{rrl}
\textit{Libraries} & L ::= & B \mid \texttt{abstract}\ m; L \\
\textit{Blocks} & B ::= & \varepsilon \mid \texttt{public}\ m = \lambda x.M; B \mid m = \lambda x.M; B \\
 & & \mid \texttt{global}\ r := i; B \mid \texttt{global}\ r := \lambda x.M; B \\
\textit{Terms} & M ::= & \texttt{assert}(M) \mid m \mid i \mid () \mid x \mid \lambda x.M \mid r := M \mid !r \mid M \oplus M \\
 & & \mid \langle M, M \rangle \mid \pi_1 M \mid \pi_2 M \mid MM \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \\
 & & \mid \texttt{letrec}\ x = \lambda x.M\ \texttt{in}\ M \mid \texttt{let}\ x = M\ \texttt{in}\ M \\
\textit{Clients} & C\ ::= & L; \texttt{main} = M
\end{array}
$$

$$
\frac{}{() : \texttt{unit}} \quad \frac{}{i : \texttt{int}} \quad \frac{x \in \texttt{Vars}_\theta}{x : \theta} \quad \frac{m \in \texttt{Meths}_{\theta,\theta'}}{m : \theta \to \theta'} \quad \frac{M, M' : \texttt{int}}{M \oplus M' : \texttt{int}}
$$

$$
\frac{M : \texttt{int} \qquad M_1, M_0 : \theta}{\texttt{if}\ M\ \texttt{then}\ M_1\ \texttt{else}\ M_0 : \theta} \quad \frac{M : \theta_1 \quad M' : \theta_2}{\langle M, M' \rangle : \theta_1 \times \theta_2} \quad \frac{\langle M, M' \rangle : \theta_1 \times \theta_2}{\pi_i \langle M, M' \rangle : \theta_i}
$$

$$
\frac{r \in \texttt{Refs}_\theta}{!r : \theta} \quad \frac{r \in \texttt{Refs}_\theta \quad M : \theta}{r := M : \texttt{unit}} \quad \frac{M' : \theta \to \theta' \quad M : \theta}{M'\, M : \theta'} \quad \frac{M : \theta' \quad x : \theta}{\lambda x.M : \theta \to \theta'}
$$

$$
\frac{x, M : \theta \quad M' : \theta'}{\texttt{let}\ x = M\ \texttt{in}\ M' : \theta'} \quad \frac{x, \lambda y.M : \theta \to \theta'' \quad M' : \theta'}{\texttt{letrec}\ x = \lambda y.M\ \texttt{in}\ M' : \theta'} \quad \frac{M : \texttt{int}}{\texttt{assert}(M) : \texttt{unit}}
$$

Figure 2.1: Syntax and typing rules of HOLi.

term. We give in HOLi an operational semantics for terms that integrates a counter for the depth of nested calls that a program phrase can make. This will be described in more detail in the following sections when addressing the operational semantics for HOLi. Our decision to focus on a client-library paradigm in HOLi is explained by the inherent open nature of libraries. Unlike terms, which may run in a first-order context, libraries are always expected to run in a context in which they are being called by the environment (a client). In addition, libraries in HOLi being higher-order means that the unknown higher-order context can provide them with undefined methods. This makes HOLi suitable as a vehicle of study for higher-order open behaviour.

### 2.2.1 Syntax and typing rules

A library in HOLi is a collection of typed higher-order methods. A client is simply a library with a main body. Types are given by the grammar:

$$\theta \ ::= \ \texttt{unit} \mid \texttt{int} \mid \theta \times \theta \mid \theta \to \theta$$

We use countably infinite sets `Meths`, `Refs` and `Vars` for method, global reference and variable names, ranged over by $m$, $r$ and $x$ respectively, and variants thereof; while $i$ is for ranging over the integers. We use $\oplus$ to range over a set of binary integer operations, which we leave unspecified. Each set of names is typed, that is, it can be expressed as a disjoint union as follows:

$$\texttt{Meths} = \biguplus_{\theta,\theta'} \texttt{Meths}_{\theta,\theta'}, \;\; \texttt{Refs} = \biguplus_{\theta \neq \theta_1 \times \theta_2} \texttt{Refs}_{\theta}, \;\; \texttt{Vars} = \biguplus_{\theta} \texttt{Vars}_{\theta}.$$

The full syntax and typing rules are given in Figure 2.1. Thus, a library consists of abstract method declarations, followed by blocks of public and private method and reference definitions. A method is considered private unless it is declared `public`. Each public/private method and reference is defined once. Abstract methods are not given definitions: these methods are external to the library. Public, private and abstract methods are all disjoint.

Libraries are well typed if all their method and reference definitions are well typed (e.g. `public` $m = \lambda x.M$ is well typed if $m : \theta$ and $\lambda x.M : \theta$ are both valid for the same type $\theta$) and only mention methods and references that are defined or abstract. A client $L$; `main` $= M$ is well typed if $M$ : `unit` is valid and $L$; $m = \lambda x.M$ is well typed for some fresh $x, m$.

**Remark 2.1.** By typing variable, reference and method names, we do not need to provide a context in typing judgements. Note that the references we use are of non-product type and, more importantly, **global** to the library: a term can use references but not create them locally or pass them as arguments (we discuss how to include local, scope extruding references in Appendix 4.2).

**Example 2.2.** For demonstration, the DAO-attack example from the Introduction can be written in HOLi as:

$$\begin{array}{l}
\texttt{abstract}\ \mathit{send};\;\; \texttt{global}\ \mathit{bal} := 100; \\
\texttt{public}\ \mathit{wdraw} = \\
\qquad \lambda x.\ \texttt{if}\ !\mathit{bal} \geq x \\
\qquad\qquad \texttt{then}\ \mathit{send}(x); \mathit{bal} := !\mathit{bal} - x; \texttt{assert}(!\mathit{bal} \geq 0) \\
\qquad\qquad \texttt{else}\ ()
\end{array}$$

where $\mathit{send}, \mathit{wdraw} \in \texttt{Meths}_{\texttt{int},\texttt{unit}}$, $\mathit{bal} \in \texttt{Refs}_{\texttt{int}}$, and we use the usual abbreviation $M; M'$ for `let` $\_ = M$ `in` $M'$. ◊

## 2.3 Operational Semantics

We provide a call-by-value operational semantics for HOLi. A library $L$ *builds* into a configuration that includes its public methods according to the rules in Figure 2.2 (top). More precisely, a built configuration $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ consists of:

- $R$ is a *method repository* mapping method names to their bodies;

$$
\begin{array}{rl}
(\texttt{abstract}\ m; L, R, S, \mathcal{P}, \mathcal{A}) & \xrightarrow{bld} (L, R, S, \mathcal{P}, \mathcal{A} \uplus \{m\}) \\
(\texttt{public}\ m = \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) & \xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S, \mathcal{P} \uplus \{m\}, \mathcal{A}) \\
(m = \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) & \xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S, \mathcal{P}, \mathcal{A}) \\
(\texttt{global}\ r := i; B, R, S, \mathcal{P}, \mathcal{A}) & \xrightarrow{bld} (B, R, S \uplus \{r \mapsto i\}, \mathcal{P}, \mathcal{A}) \\
(\texttt{global}\ r := \lambda x.M; B, R, S, \mathcal{P}, \mathcal{A}) & \xrightarrow{bld} (B, R \uplus \{m \mapsto \lambda x.M\}, S \uplus \{r \mapsto m\}, \mathcal{P}, \mathcal{A})
\end{array}
$$

$$
\begin{array}{l}
(E[\texttt{assert}(i)], R, S, k) \to (E[()], R, S, k) \quad (i \neq 0) \\
(E[r := v], R, S, k) \to (E[()], R, S[r \mapsto v], k) \\
(E[!r], R, S, k) \to (E[S(r)], R, S, k) \\
(E[\pi_j \langle v_1, v_2 \rangle], R, S, k) \to (E[v_j], R, S, k) \\
(E[i_1 \oplus i_2], R, S, k) \to (E[i], R, S, k) \quad (i = i_1 \oplus i_2) \\
(E[\texttt{if}\ i\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_j], R, S, k) \quad (j = 1 \text{ iff } i \neq 0) \\
(E[\lambda x.M], R, S, k) \to (E[m], R \uplus \{m \mapsto \lambda x.M\}, S, k) \\
(E[\texttt{let}\ x = v\ \texttt{in}\ M], R, S, k) \to (E[M\{v/x\}], R, S, k) \\
(E[\texttt{letrec}\ f = \lambda x.M\ \texttt{in}\ M'], R, S, k) \\
\qquad \to (E[M'\{m/f\}], R \uplus \{m \mapsto \lambda x.M\{m/f\}\}, S, k) \\
(E[mv], R, S, k) \to (E[\llparenthesis M\{v/x\} \rrparenthesis], R, S, k+1) \quad (R(m) = \lambda x.M) \\
(E[\llparenthesis v \rrparenthesis], R, S, k+1) \to (E[v], R, S, k)
\end{array}
$$

$$
\begin{array}{rrll}
\textit{Values} & v & ::= & m \mid i \mid () \mid \langle v, v \rangle \qquad \textit{Terms (extended)}\ \ M ::= \ \cdots \mid \llparenthesis M \rrparenthesis \\
\textit{Eval. Contexts} & E & ::= & \bullet \mid \texttt{assert}(E) \mid r := E \mid E \oplus M \mid v \oplus E \mid \langle E, M \rangle \mid \langle v, E \rangle \mid \pi_j E \\
& & & \mid EM \mid mE \mid \texttt{let}\ x = E\ \texttt{in}\ M \mid \texttt{if}\ E\ \texttt{then}\ M\ \texttt{else}\ M \mid \llparenthesis E \rrparenthesis
\end{array}
$$

Figure 2.2: Library build (top); operational semantics (bottom).

- $S$ is a *store* mapping reference names to their stored values; and
- $\mathcal{P}, \mathcal{A} \subseteq \texttt{Meths}$ are (disjoint) sets of *public* and *abstract* method names.

We say that (a well typed) $L$ builds to $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ if $(L, \emptyset, \emptyset, \emptyset, \emptyset) \xrightarrow{bld}{}^{*} (\varepsilon, R, S, \mathcal{P}, \mathcal{A})$. If $L$ builds to $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ then the client $L$; $\texttt{main} = M$ builds to $(M, R, S, \mathcal{P}, \mathcal{A})$.

We say that library $L$ and client $C$ are *compatible* if $L$ builds to some $(\varepsilon, R, S, \mathcal{P}, \mathcal{A})$ and $C$ builds to some $(M, R', S', \mathcal{P}', \mathcal{A}')$ such that:

- $\mathcal{P} \supseteq \mathcal{A}'$ and $\mathcal{A} \supseteq \mathcal{P}'$ (*complementation*);
- $\mathrm{dom}(S) \cap \mathrm{dom}(S') = \emptyset$ (*disjoint state*); and
- $\mathrm{dom}(R) \cap \mathrm{dom}(R') = \emptyset$ (*unique method ownership*).

For a library $L$, we let $\hat{L}$ be $L$ with all its abstract method declarations and `public` keywords removed; and similarly for $\hat{C}$.

**Definition 2.3.** Given compatible library $L$ and client $C$, we let their ***syntactic composition*** be the client: $L\mathbf{;}C = \hat{L}; \hat{C}$.

A built library contains public methods that can be called by a client. On the other hand, a client contains a main body that can be executed. These two scenarios constitute the operational semantics of HOLi. Both are based on evaluating (closed) terms, which we define next. Term evaluation requires: the closed term being evaluated; method definitions, provided by a method repository; reference values, provided by a store; and a call-depth counter (a natural number). Since method application is the only source of infinite behaviour in HOLi, bounding the depth of nested calls is enough to guarantee termination in program analysis. Hence we provide a mechanism to keep track of call depth.

Thus, the operational semantics involves configurations of the form $(M, R, S, k)$, where $R$ is a repository, $S$ is a store and $k$ is a natural number. $M$ is closed term taken from a syntax extending terms of Figure 2.1 with the rule:

$$M ::= \ \cdots \mid (\!|M|\!)$$

which defines ***evaluation boxes***, i.e. points inside a term where a method call has been made and has not yet returned. Evaluation boxes interplay with the counter $k$ in the semantics: they mark places where the depth has increased because of a nested call. The operational semantics of closed terms is given in Figure 2.2 (bottom). Here, $E$ defines the evaluation contexts for the call-by-value evaluation. The last two rules in the figure are the ones keeping track of call depth, and illustrate the utility of evaluation boxes: making a call increases the counter and leaves behind an evaluation box; returning form the call removes the box and decreases the counter again.

We can use the operational semantics to evaluate linked library-client pairs.

**Definition 2.4.** Given compatible $L, C$, the semantics of $L;C$ is:

$$[\![L;C]\!] = \{\rho \mid L;C \text{ builds to } (M, R, S, \emptyset, \emptyset) \wedge (M, R, S, 0) \to^* \rho\}$$

We say that $[\![L;C]\!]$ *fails* if it contains some $(E[\texttt{assert}(0)], \cdots)$.

**Example 2.5.** To illustrate how libraries and clients are used, consider the DAO example again as a library $L_{\texttt{DAO}}$. We can define a client $C_{\texttt{atk}}$:

$$\begin{aligned}&\texttt{abstract}\ wdraw;\\ &\texttt{global}\ wlet := 0;\\ &\texttt{public}\ send = \lambda x.wlet := !wlet + x; wdraw(x);\\ &\texttt{main} = wdraw(1)\end{aligned}$$

to produce the following linked client $L_{\texttt{DAO}};C_{\texttt{atk}}$:

$$\begin{aligned}&\texttt{global}\ bal := 100;\\ &wdraw = \lambda x.\ \texttt{if}\ !bal \geq x\end{aligned}$$

```
            then send(x); bal := !bal − x; assert(!bal > 0)
            else ();
    global wlet := 0;
    send = λx.wlet := !wlet + x; wdraw(x);
    main = wdraw(1)
```

We can see how $L_{\texttt{DAO}}$ is vulnerable to an attacker such as $C_{\texttt{atk}}$ after linking them. Using the operational semantics defined here, we observe that main body defines an infinitely recursive call to *wdraw*, which we can bound for analysis. ♢

### 2.3.1 Nominal Defunctionalisation

Our use of names as method identifiers stems from the use of nominal techniques in operational semantics, such as in [55]. The purpose is to flatten higher-order terms during computation by identifying every method with a unique value (a name). These names are then called via a method repository. This allows us to automatically reason about higher-order terms, for example, in SAT/SMT instances. The process of flattening higher-order structure with the aid of a call function (as we do with the repository) makes this a form of defunctionalisation. This nominal presentation of defunctionalisation is theoretically beneficial in operational semantics as it captures equivalence classes for $\alpha$-equivalence of programs via permutation [66], independently of local scope and tree structure. Names will additionally be useful to define a notion of privacy between components with respect to a library and its environment. The properties gained in using nominal sets are foundational to the theoretical results given in this thesis.

### 2.3.2 Bounding the Semantics and Nominal Determinacy

The purpose of providing a call counter $k$ is to eventually use it to bound the semantics. Provided an upper bound $k_0$ to $k$, the plan is to reach a stuck configuration if $k$ were to exceed $k_0$, thus bounding the depth of method application. However, even if the operational semantics is bounded in depth, the reduction tree of a given term can still be infinite because of the non-determinacy involved in evaluating $\lambda$-abstractions: the rule non-deterministically creates a fresh name $m$ and extends the repository with $m$ mapped to the given $\lambda$-abstraction. This kind of non-determinism, which can be seen as *determinism up to fresh name creation*, is formalised below.

Let us consider permutations $\pi : \texttt{Meths} \to \texttt{Meths}$ such that, for all $m$, if $m \in \texttt{Meths}_{\theta\to\theta'}$ then $\pi(m) \in \texttt{Meths}_{\theta\to\theta'}$. We call such a permutation $\pi$ *finite* if the set $\{a \mid \pi(a) \neq a\}$ is finite. Given a syntactic object $X$ (e.g. a term, repository, or store) and a finite permutation $\pi$, we write $\pi \cdot X$ for the object we obtain from $X$ if we swap each name $a$ appearing in it with $\pi(a)$. Put otherwise, the operation $\cdot$ is an action from finite permutations of `Meths` to the set of objects $X$. Given a set $\Delta \subseteq \texttt{Meths}$ and objects $X, X'$, we write $X \sim_\Delta X'$ whenever there exists a finite permutation $\pi$ such that:

$$\pi \cdot X = X' \wedge \forall a \in \Delta.\ \pi(a) = a$$

and say that $X$ and $X'$ are ***nominally equivalent*** up to $\Delta$.

In the following lemma we write $\twoheadrightarrow_n$ for the $n$-step composition of $\to$.

**Lemma 2.6** (Nominal determinacy)**.** Let $(T, R, S, k)$ be a valid configuration, $(T, R, S, k) \twoheadrightarrow_n (T', R', S', k')$, and let $\Delta = \mathrm{dom}(R) \cup \mathrm{dom}(S)$. Then, for all $(T'', R'', S'', k'')$ we have $(T, R, S, k) \twoheadrightarrow_n (T'', R'', S'', k'')$ iff $(T', R', S', k') \sim_\Delta (T'', R'', S'', k'')$.

### 2.3.3 Expressivity of Higher-Order Assertions

While a full formalised discussion is not in scope of this thesis, one may question whether our assertion language is expressive enough to capture higher-order properties and what these properties actually mean. Since the notion of *higher-order assertions* has no single definition, let us consider expressing properties expressible with a higher-order logic; i.e. assertions over predicates of arbitrary order.

Firstly, unlike in first-order languages, assertions in our higher-order setting are able to appear in and contain higher-order terms. Since an assertion over higher-order terms is allowed to express properties on free variables of arbitrary order present in said term, assertions in our language are indeed allowed to express properties on higher-order relations—i.e. higher-order predicates. There are, however, a few caveats since we inherit the limitations of using assertions. For instance, we are limited to reachability, which means we may only express safety properties. Moreover, assertions are implicitly universally quantified over the free variables present in them—where the domain is defined by the path—which is similar to the notion of *global* properties in linear temporal logics; analogous to $\forall \vec{x} \in FV(pc).pc(\vec{x}) \implies \Box P(\vec{x})$ for some path condition $pc$ and its free variables $FV(pc)$, and some predicate $P$. This differs from branching-time logics in that a notion for the existence of a path does not appear to be expressible with assertions alone. Finally, since assertions apply a single top-level universal quantification over all free variables present in them, and since assertions cannot be negated, we are limited in the degree of arithmetic hierarchy expressible—it appears we are limited to a single top-level non-negative universals with no existential quantifiers, which rules out a Skolem normal form conversion. Notice also that nesting assertions is no different to sequencing them at the top level, since properties in assertions can be independently accumulated.

# Chapter 3

# Bounded Model Checking Closed Higher-Order Programs

In this chapter, we present a BMC technique for closed higher-order programs, that is, higher-order programs which may only have input variables of ground type, but allow for free variables of arbitrary order during computation. This chapters focusses on HORef, the ground-type fragment of HOLi. The technique presented here is a syntactic translation that encodes a given higher-order program into a SAT instance that captures its behaviour. For the encoding, we obtain inspiration from operational semantics in the form of a presentation of defunctionalization using nominal techniques. We also make use of a points-to analysis to improve performance of our BMC procedure.

## 3.1 HORef: Closed Fragment of HOLi

Before we proceed with our BMC approach, let us present HORef, the closed fragment of HOLi we shall be using as our higher-order stateful setting. As already mentioned in Chapter 2, closed higher-order programs do not contain free variables of higher order, but may still internally reason about higher-order terms. For this, the syntax is that of terms in HOLi, which is a call-by-value $\lambda$-calculus with global higher-order references. As before, its types are given by the grammar:

$$\theta ::= \texttt{unit} \mid \texttt{int} \mid \theta \times \theta \mid \theta \to \theta$$

The syntax given in Figure 3.1 (top) is that of the canonical forms for HORef. We use terms in canonical form as it simplifies the presentation of our BMC algorithm without loosing generality. This shall also simplify the proof of correctness, and can be obtained linearly from terms in general syntax through a standard transformation.

$$\begin{aligned}
\texttt{CForms} \ni M ::= {} & \texttt{assert}(v) \mid v \mid r := v \mid !r \mid v \oplus v \mid \pi_1\, v \mid \pi_2\, v \\
& \mid x\ v \mid m\ v \mid \lambda x.M \mid \texttt{if}\ \ v\ \ \texttt{then}\ \ M\ \ \texttt{else}\ \ M \\
& \mid \texttt{let}\ \ x = M\ \ \texttt{in}\ \ M \mid \texttt{letrec}\ \ x = \lambda x.M\ \ \texttt{in}\ \ M
\end{aligned}$$

$$\begin{aligned}
&(\texttt{assert}j, R, S, k) \to ((), R, S, k) \\
&(!r, R, S, k) \to (S(r), R, S, k) \\
&(r := v, R, S, k) \to ((), R, S[r \mapsto v], k) \\
&(\pi_i \langle v_1, v_2 \rangle, R, S, k) \to (v_i, R, S, k) \\
&(i_1 \oplus i_2, R, S, k) \to (i, R, S, k) \quad (i = i_1 \oplus i_2) \\
&(\lambda x.T, R, S, k) \to (m, R[m \mapsto \lambda x.T], S, k) \\
&(\texttt{if}\ \ j\ \ \texttt{then}\ \ T_1\ \ \texttt{else}\ \ T_0, R, S, k) \to (T_1, R, S, k) \quad (j \neq 0) \\
&(\texttt{if}\ \ 0\ \ \texttt{then}\ \ T_1\ \ \texttt{else}\ \ T_0, R, S, k) \to (T_0, R, S, k) \\
&(mv, R, S, k1) \to (\llparenthesis T\{v/x\} \rrparenthesis, R, S, k-1) \quad (R(m) = \lambda x.T) \\
&(\llparenthesis v \rrparenthesis, R, S, k) \to (v, R, S, k+1) \\
&(\texttt{let}\ \ x = v\ \ \texttt{in}\ \ T, R, S, k) \to (T\{v/x\}, R, S, k) \\
&(\texttt{letrec}\ \ f = \lambda x.T\ \ \texttt{in}\ \ T', R, S, k) \to (T'\{m/f\}, R[m \mapsto \lambda x.T\{m/f\}], S, k) \\
&(E[T], R, S, k) \to (E[T'], R', S', k') \quad \text{where } (T, R, S, k) \to (T', R', S', k')
\end{aligned}$$

Figure 3.1: Canonical forms for HORef (top) and their semantics (bottom).

The operational semantics of HORef is equivalent to the term semantics of HOLi given in Figure 2.2. The presentation for HORef is differs slightly from that of HOLi because rules are provided for terms in canonical form. This means the evaluation context is omitted from each rule and replaced by the addition of an extra rule for $E[T]$, since all evaluation contexts are of the same form (nested let bindings) when operating within canonical forms. For HORef, we also make a slight change in the convention for counters in that the direction of call counters is inverted and decreasing $k$ beyond zero results in $k = \texttt{nil}$. This is done for a simple presentation of a bounded semantics that shares conventions with the BMC procedure defined in the next chapter (which in turn uses conventions inspired by prior BMC techniques). Inverting the call counters in the operational semantics is easily done by swapping addition for subtraction when manipulating $k$ in the rules for method application and return. The starting configuration is also modified to start from the bound $k = k_0$, where $k \geq 0$ must hold for method application to be applicable. This change in convention is harmless as it is equivalent to counting up to the bound.

In HORef, the usual definition for free variables holds. The difference between HORef and HOLi is that, unlike libraries, terms in HORef are closed as they do not expect the input of external methods, making HORef a ground-type fragment of HOLi. As such, when discussing a symbolic configuration for HORef, we consider only top-level inputs of ground type. Note that while HORef is grounded overall, it is still a higher-order language with higher-order store. Higher-order behaviour occurs internally in the semantics and must be considered in designing an analysis technique. This will be made clearer in the next section where we consider HORef symbolically.

## 3.2 A Bounded Translation for HORef

Let us begin with an example of the procedure we shall be defining. In Example 3.1, we can observe how our algorithm unrolls the control-flow graph of a given program and how it explores this graph to accumulate symbolic constraints about its behaviour. The example also showcases how a solver can be used to check for reachability of errors.

Intuitively, the example involves a stateful higher-order function f that, after a few recursive calls, produces a function that asserts that the store has been updated an expected n times for any n. In this case, the example has no reachable errors, meaning that there is no way in which n may reachably differ from stored counter in the assertion generated. Note, however, that since reachability only concerns safety properties, we miss a problem with termination. Specifically, the condition in f only ever checks if x is 0 to stop, meaning that if the starting x is negative, the program would never halt. This would not be provable by our technique since we are only concerned with reachability.

**Example 3.1.** Consider the following program, where $n$ is an open variable and the reference $r$ is initialised to 0.

```
letrec f =
  λ x. if x then (r++;  f (x - 1))
            else (λ y. assert (y = !r + x))
in
let g = f n in g n
```

When applied to this code, using $k = 2$, the translation follows the steps depicted in Figure 3.2. Each rectangular box represents a recursive call of the translation. The code examined is placed on the top half of the box, while the updates in $R$, $k$, $\phi$ and $\alpha$ are depicted in the bottom half (e.g. $\phi : r_0 = 0$ means that we attach $r_0 = 0$ to $\phi$ as a conjunct). Rounded boxes, on the other hand, represent joining of branches spawned by conditional statements or by exhaustive method application. In those boxes we include the updates to $\phi$ which encode the joining. The first two rounded boxes (single-lined) correspond to the joins of the two conditional statements examined. The last rounded box is where the branches spawned by examining $ret_g\, n$ are joined: $ret_g$ could be either of $m_2$ or $m_3$.

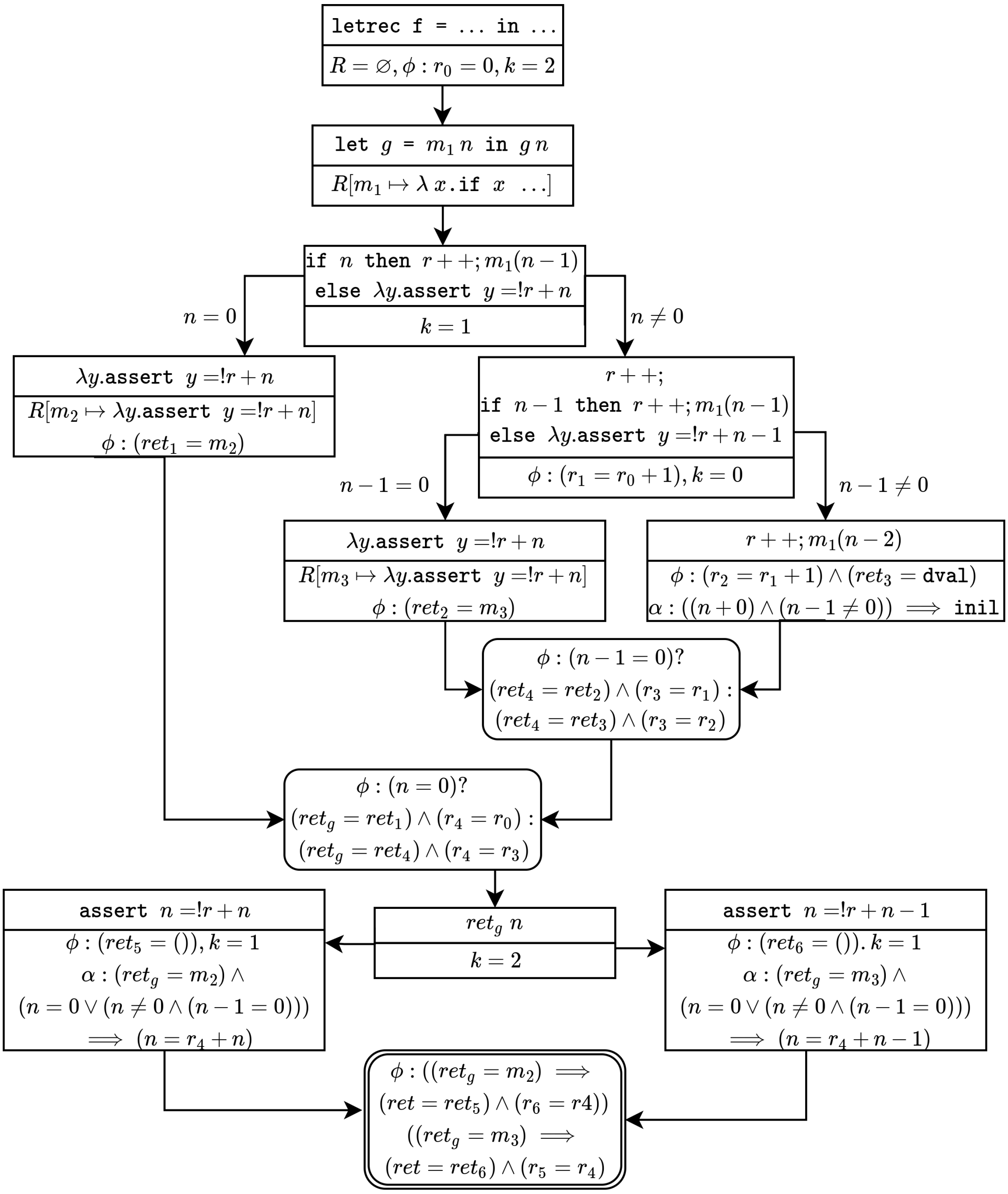

letrec f = ... in ...
$R = \varnothing, \phi : r_0 = 0, k = 2$
let $g$ = $m_1\, n$ in $g\, n$
$R[m_1 \mapsto \lambda\, x$.if $x$ ...]
if $n$ then $r++; m_1(n-1)$
else $\lambda y$.assert $y = !r + n$
$k = 1$
$n = 0$
$n \neq 0$
$\lambda y$.assert $y = !r + n$
$R[m_2 \mapsto \lambda y$.assert $y = !r + n]$
$\phi : (ret_1 = m_2)$
$r++;$
if $n-1$ then $r++; m_1(n-1)$
else $\lambda y$.assert $y = !r + n - 1$
$\phi : (r_1 = r_0 + 1), k = 0$
$n - 1 = 0$
$n - 1 \neq 0$
$\lambda y$.assert $y = !r + n$
$R[m_3 \mapsto \lambda y$.assert $y = !r + n]$
$\phi : (ret_2 = m_3)$
$r++; m_1(n-2)$
$\phi : (r_2 = r_1 + 1) \wedge (ret_3 =$ dval$)$
$\alpha : ((n+0) \wedge (n-1 \neq 0)) \implies$ inil
$\phi : (n-1 = 0)?$
$(ret_4 = ret_2) \wedge (r_3 = r_1) :$
$(ret_4 = ret_3) \wedge (r_3 = r_2)$
$\phi : (n = 0)?$
$(ret_g = ret_1) \wedge (r_4 = r_0) :$
$(ret_g = ret_4) \wedge (r_4 = r_3)$
assert $n = !r + n$
$\phi : (ret_5 = ()), k = 1$
$\alpha : (ret_g = m_2) \wedge$
$(n = 0 \vee (n \neq 0 \wedge (n - 1 = 0)))$
$\implies (n = r_4 + n)$
$ret_g\ n$
$k = 2$
assert $n = !r + n - 1$
$\phi : (ret_6 = ()). k = 1$
$\alpha : (ret_g = m_3) \wedge$
$(n = 0 \vee (n \neq 0 \wedge (n - 1 = 0)))$
$\implies (n = r_4 + n - 1)$
$\phi : ((ret_g = m_2) \implies$
$(ret = ret_5) \wedge (r_6 = r4))$
$((ret_g = m_3) \implies$
$(ret = ret_6) \wedge (r_5 = r_4)$

Figure 3.2: Control flow of BMC on Example 3.1.

The program behaviour and assertions are thus captured by the following formulas.

$$\begin{aligned}
\phi = \quad & (r_0 = 0) \land (ret_1 = m_2) \land (r_1 = r_0 + 1) \land (ret_2 = m_3) \land (r_2 = r_1 + 1) \land (ret_3 = \mathtt{dval}) \\
& \land ((n - 1 = 0)?((ret_4 = ret_2) \land (r_3 = r_1)) : ((ret_4 = ret_3) \land (r_3 = r_2))) \\
& \land ((n = 0)?((ret_g = ret_1) \land (r_4 = r_0)) : ((ret_g = ret_4) \land (r_4 = r_3))) \\
& \land (ret_5 = ()) \land (ret_6 = ()) \land ((ret_g = m_2) \implies (ret = ret_6) \land (r_5 = r_4)) \\
& \land ((ret_g = m_3) \implies (ret = ret_7) \land (r_5 = r_4)) \\
\alpha = \quad & (((ret_3 = m_2) \land (n = 0 \lor (n \neq 0 \land (n - 1 = 0)))) \implies (n = r_4 + n)) \\
& \land (((ret_3 = m_3) \land (n = 0 \lor (n \neq 0 \land (n - 1 = 0)))) \implies (n = r_4 + n - 1)) \\
& \land (((n + 0) \land (n - 1 \neq 0)) \implies \mathtt{inil})
\end{aligned}$$

◊

**Definition 3.2** (BMC translation)**.** Given a valid initial configuration $(M, R, S, k)$, let the bounded translation of $(M, R, S, k)$ be $[\![M, R, S, k]\!]$, where $[\![\cdot]\!]$ is defined in Figure 3.3, $[\![M, R, S, k]\!] = [\![init(M, R, S, k)]\!]$ for some this valid configuration, and $init(M, R, S, k)$ is defined by the sequence:

$$(M, R, S, k) \overset{init}{\longmapsto} (M, R, C_S, C_S, \phi_S, \top, \top, k)$$

In Definition 3.2 we define the bounded translation for a given initial configuration. Our algorithm produces a tuple containing (quantifier-free) first-order formulas and context components that capture its bounded semantics. Without loss of generality, we define the translation on terms in canonical form, ranged over by $M$ and variants, which are presented in Figure 3.1. For some valid input configuration $(M, R, S, k)$, where $M$ is in canonical form and may only contain free variables of ground type, the translation performs the following sequence of transformations:

$$(M, R, S, k) \overset{init}{\longmapsto} (M, R, C_S, C_S, \phi_S, \top, \top, k) \overset{[\![\cdot]\!]}{\longmapsto} (ret, \phi, R', C, D, \alpha, pc)$$

The first step is an initialisation step that transforms the tuple in the form appropriate for the main translation $[\![\cdot]\!]$, which is the essence of the entire bounded translation. We proceed with $[\![\cdot]\!]$ and will be returning to *init* later on.

$[\![\cdot]\!]$ operates on symbolic configurations of the form $(M, R, C, D, \phi, \alpha, pc, k)$, where $M$ and $R$ are a term and a repository respectively, $k$ is the bound, and:

- $C, D : \mathtt{Refs} \rightharpoonup \mathtt{SSAVars}$ are single static assignment (SSA) maps where $\mathtt{SSAVars}$ is the set of variables of the form $r_i$ (for each $r \in \mathtt{Refs}$), such that $i$ is the number of times that $r$ has been assigned to so far. The map $C$ is counting all the assignments that have taken place so far, whereas $D$ only counts those in the current path; e.g. $C(r) = r_5$ if $r$ has been assigned to five times so far in every path looked at. We write $C[r]$ to mean *update $C$ with reference $r$*: if $C(r) = r_i$, then $C[r] = C[r \mapsto r_{i+1}]$, where $r_{i+1}$ is fresh.
- $\phi$ is a first-order formula containing the (total) behaviour so far.

- $\alpha$ is a first-order formula consisting of a conjunction of statements representing assertions that have been visited by $[\![\cdot]\!]$ so far.
- $pc$ is the path condition that must be satisfied to reach this configuration.

The translation returns tuples of the form $(ret, \phi, R, C, D, \alpha, pc)$, where:

- $\phi, R, C, D, \alpha, pc$ have the same reading as above, albeit for *after reaching the end of all paths from the term $M$.*
- $ret$ is a logic variable representing the return value of the initial configuration.

Finally, returning to $init$, we have that:

- the initial SSA maps $C_S$ simply map each $r$ in the domain of $S$ to the SSA variable $r_0$, i.e. $C_S = \{r \mapsto r_0 \mid r \in dom(S)\}$;
- $\phi_S$ stipulates that each $r_0$ be equal to its corresponding value $S(r)$, i.e. $\phi_S = \bigwedge_{r \in dom(S)}(r_0 = S(r))$;
- since there is no computation preceding $M$, its $\alpha$ and $pc$ are simply $\top$ (true).

### 3.2.1 The BMC translation

The translation is given in Figure 3.3. In all cases in the figure, $ret$ is a fresh variable and $k \neq \texttt{nil}$. We also assume a common domain $\Pi = dom(C) = dom(D) \subseteq \texttt{Refs}$, which contains all references that appear in $M$ and $R$.

The translation stops when either the bound is `nil`, or when every path of the given term has been explored completely. The base cases add clauses mapping return variables to actual values of evaluating $M$. Inductive cases build the symbolic trace of $M$ by recording in $\phi$ all changes to the store, and adding clauses for return variables at each sub-term of the program, thus building a control flow graph by relating said return variables and chaining them together in the formula. Wherever branching occurs, the chaining is guarded.

In the translation, defunctionalisation occurs because every method call is replaced with a call to the repository using its respective name as an argument. Because this is a symbolic setting, however, it is possible to lose track of the specific name desired. Particularly, when applying variables as methods ($xv$, with $x : \theta$), we encode in the behaviour formula an $n$-ary decision tree where $n$ is the number of methods to consider. In such cases, we assume that $x$ could be any method in the repository $R$. This case corresponds to exhaustive method application and is fundamental for applying BMC to higher-order terms with defunctionalisation. To explore plausible paths only, we restrict $R$ to type $\theta$ (denoted $R \upharpoonright \theta$). In Section 3.4 we will be applying a points-to analysis to restrict this further.

To illustrate the algorithm, we look at a few characteristic cases:

**[nil case]** When the translation consumes its bound, we end up translating some $[\![M, R, C, D, \phi, \alpha, pc, \texttt{nil}]\!]$. In this case, we simply return a fresh variable $ret$ representing the final value, and stipulate in the program behaviour that $ret$ is equal to some default value (the latter is needed to ensure a unique model for $ret$). Breaching of the bound is recorded as a possible assertion violation, and a reserved logic variable `inil` is used for that purpose: a breach counts as an assertion violation iff `inil` is false. The returned path condition is set to false.

**[let case]** In $[\![\text{let } x = M \text{ in } M', R, C, D, \phi, \alpha, pc, k]\!]$, we first compute the translation of $M$. Using the results of said translation, we can substitute in $M'$ the fresh variable $ret_1$ for $x$, and compute its translation. In the latter step, we also feed the updated repository $R_1$, SSA maps $C_1$ and $D_1$, program behaviour $\phi_1$, assertions $\alpha_1$ (actually, a conjunction of assertions), and the accumulated path condition $pc \wedge pc_1$. To finish, we return $ret_2$ and the newly updated repository $R_2$, SSA maps, $C_2$ and $D_2$, assertions $\alpha_2$. The path condition returned is $pc_1 \wedge pc_2$, reflecting the new path conditions gathered.

**[xv case]** In $[\![xv, R, C, D, \phi, \alpha, pc, k]\!]$ we see exhaustive method application in action. We first restrict the repository $R$ to type $\theta$ to obtain the set of names identifying all methods of matching type for $x$. If no such methods exist, this means that the binding of $x$ had not succeeded due to breaching the bound earlier on, so `dval` is returned. Otherwise, for each method $m_i$ in this set, we obtain the translation of applying $m_i$ to the argument $v$. This is done by substituting $v$ for $y_i$ in the body of $m_i$. After translating all method applications, all paths are joined in $\psi$ by constructing an $n$-ary decision tree that includes the state of the store in each path. We do this by incrementing all references in $C_n$, and adding the clauses $C'_n = D_i(r)$ for each path. These paths are then guarded by the clauses $(x = m_i)$. Finally, we return a formula that includes $\psi$ and the accumulation of all intermediate $\phi_i$'s, the accumulation of repositories, the final SSA map, accumulation of assertions and new path conditions. Note that we return $C'_n$ in the position of both the $C$ and $D$ resulting from translating this term. This is because all branches have been joined and any term sequenced after this one should have all updates available to it.

**Remark 3.3.** The difference between reading ($D$) and writing ($C$) is noticeable when branching. Branching can occur in two ways: through a conditional statement, and by performing symbolic method application where we have lost track of the concrete method; more precisely, when $M$ is of the form $xv$. In the former case, we branch according to the return value of the condition (denoted by $ret_b$), and each branch translates $M_0$ and $M_1$ respectively. In this case, both branches read from the same map $D_b$, but may contain different assignments, which we accumulate in $C$. The formula $\psi_0 \wedge \psi_1$ then encodes a binary decision node in the control flow graph through guarded clauses that represent the path conditions. Similar care is taken with branching caused by symbolic method application.

**Base Cases:**

$$\llbracket \texttt{assert}\ v, R, C, D, \phi, \alpha, pc, k \rrbracket = (ret, (ret = ()) \wedge \phi, R, C, D, (pc \implies (v \neq 0)) \wedge \alpha, \top)$$
$$\llbracket M, R, C, D, \phi, \alpha, pc, \texttt{nil} \rrbracket = (ret, (ret = \texttt{dval}) \wedge \phi, R, C, D, \alpha \wedge (pc \implies \texttt{inil}), \bot)$$
$$\llbracket v, R, C, D, \phi, \alpha, pc, k \rrbracket = (ret, (ret = v) \wedge \phi, R, C, D, \alpha, \top)$$
$$\llbracket !r, R, C, D, \phi, \alpha, pc, k \rrbracket = (ret, (ret = D(r)) \wedge \phi, R, C, D, \alpha, \top)$$
$$\llbracket \lambda x.M, R, C, D, \phi, \alpha, pc, k \rrbracket = (ret, (ret = m) \wedge \phi, R', C, D, \alpha, \top)$$
$$\text{where } R' = R[m \mapsto \lambda x.M] \text{ and } m \text{ fresh}$$
$$\llbracket \pi_i\, v, R, C, D, \phi, \alpha, pc, k \rrbracket = (ret, (ret = \pi_i\, v) \wedge \phi, R, C, D, \alpha, \top)$$
$$\llbracket v_1 \oplus v_2, R, C, D, \phi, \alpha, pc, k \rrbracket = (ret, (ret = v_1 \oplus v_2) \wedge \phi, R, C, D, \alpha, \top)$$
$$\llbracket r := v, R, C, D, \phi, \alpha, pc, k \rrbracket = \text{let } C' = C[r] \text{ in let } D' = D[r \mapsto C'(r)] \text{ in}$$
$$(ret, ((ret = ()) \wedge (D'(r) = v)) \wedge \phi, R, C', D', \alpha, \top)$$

**Inductive Cases:**

$$\llbracket \texttt{let}\ x = M\ \texttt{in}\ M', R, C, D, \phi, \alpha, pc, k \rrbracket =$$
$$\text{let } (ret_1, \phi_1, R_1, C_1, D_1, \alpha_1, pc_1) = \llbracket M, R, C, D, \phi, \alpha, pc, k \rrbracket \text{ in}$$
$$\text{let } (ret_2, \phi_2, R_2, C_2, D_2, \alpha_2, pc_2) = \llbracket M'\{ret_1/x\}, R_1, C_1, D_1, \phi_1, \alpha_1, pc \wedge pc_1, k \rrbracket \text{ in}$$
$$(ret_2, \phi_2, R_2, C_2, D_2, \alpha_2, pc_1 \wedge pc_2)$$
$$\llbracket \texttt{letrec}\ f = \lambda x.M\ \texttt{in}\ M', R, C, D, \phi, \alpha, pc, k \rrbracket =$$
$$\text{let } m \text{ be fresh in}$$
$$\text{let } R' = R[m \mapsto \lambda x.M\{m/f\}] \text{ in } \llbracket M'\{m/f\}, R', C, D, \phi, \alpha, pc, k \rrbracket$$
$$\llbracket m\, v, R, C, D, \phi, \alpha, pc, k \rrbracket =$$
$$\text{let } R(m) \text{ be } \lambda x.N \text{ in } \llbracket N\{v/x\}, R, C, D, \phi, \alpha, pc, k-1 \rrbracket$$
$$\llbracket \texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0, R, C, D, \phi, \alpha, pc, k \rrbracket =$$
$$\text{let } (ret_0, \phi_0, R_0, C_0, D_0, \alpha_0, pc_0) = \llbracket M_0, R, C, D, \phi, \alpha, pc \wedge (v = 0), k \rrbracket \text{ in}$$
$$\text{let } (ret_1, \phi_1, R_1, C_1, D_1, \alpha_1, pc_1) = \llbracket M_1, R_0, C_0, D, \phi_0, \alpha_0, pc \wedge (v \neq 0), k \rrbracket \text{ in}$$
$$\text{let } C' = C_1[r_1] \cdots [r_n] \ (\mathit{\Pi} = \{r_1, \ldots, r_n\}) \text{ in}$$
$$\text{let } \psi_0 = (v = 0) \implies ((ret = ret_0) \wedge \textstyle\bigwedge_{r \in \mathit{\Pi}} (C'(r) = D_0(r))) \text{ in}$$
$$\text{let } \psi_1 = (v \neq 0) \implies ((ret = ret_1) \wedge \textstyle\bigwedge_{r \in \mathit{\Pi}} (C'(r) = D_1(r))) \text{ in}$$
$$(ret, \psi_0 \wedge \psi_1 \wedge \phi_1, R, C', C', \alpha_1, ((pc_0 \wedge (v = 0)) \vee (pc_1 \wedge (v \neq 0))))$$
$$\llbracket x^\theta\, v, R, C, D, \phi, \alpha, pc, k \rrbracket =$$
$$\text{if } R \restriction \theta = \emptyset \text{ then } (ret, (ret = \texttt{dval}) \wedge \phi, R, C, D, \alpha, \bot) \text{ else}$$
$$\text{let } R \restriction \theta \text{ be } \{m_1, ..., m_n\} \text{ and } (R, C, \phi, \alpha) \text{ be } (R_0, C_0, \phi_0, \alpha_0) \text{ in}$$
$$\text{for each } i \in \{1, ..., n\} :$$
$$\text{let } R(m_i) \text{ be } \lambda y_i.N \text{ in let } (ret_i, \phi_i, R_i, C_i, D_i, \alpha_i, pc_i) =$$
$$\llbracket N_i\{v/y_i\}, R_{i-1}, C_{i-1}, D, \phi_{i-1}, \alpha_{i-1}, pc \wedge (x = m_i), k-1 \rrbracket \text{ in}$$
$$\text{let } C'_n = C_n[r_1] \cdots [r_j] \ (\mathit{\Pi} = \{r_1, \ldots, r_j\}) \text{ in}$$
$$\text{let } \psi = \bigwedge_{i=1}^{n} \Big( (x = m_i) \implies ((ret = ret_i) \wedge \textstyle\bigwedge_{r \in \mathit{\Pi}} (C'_n(r) = D_i(r))) \Big) \text{ in}$$
$$\text{let } pc'_n = \bigvee_{i=1}^{n} (pc_i \wedge (x = m_i)) \text{ in } (ret, \psi \wedge \phi_n, R_n, C'_n, C'_n, \alpha_n, pc'_n)$$

Figure 3.3: The BMC translation.

### 3.2.2 BMC via the Translation

The steps to do a $k$-bounded model checking of some initial configuration $(M, R, S, k)$, where $M$ has free ground-type variables $\vec{x}$, using the algorithm described previously are:

1. Build $\mathit{init}(M, R, S, k) = (M, R, C_S, C_S, \phi_S, \top, \top, k)$.
2. Compute the translation: $[\![M, R, C_S, C_S, \phi_S, \top, \top, k]\!] = (\mathit{ret}, \phi, R', C, D, \alpha, \mathit{pc})$.
3. Check $\phi \wedge \texttt{inil} \wedge \neg\alpha$ for satisfiability:
   (a) If satisfiable, we have a model for $\phi \wedge \texttt{inil} \wedge \neg\alpha$ that provides values for all open variables $\vec{x}$ and, therefore, a reachable assertion violation.
   (b) Otherwise, check $\phi \wedge \neg\texttt{inil} \wedge \neg\alpha$ for satisfiability. If satisfiable, the bound was breached. Otherwise, the program has been exhaustively explored with no errors, i.e. has been fully verified.

**Remark 3.4.** Note how bound breaches are regarded as assertion violations only if `inil` is false (so `inil` means: ignore nils). Analysing Example 3.1 for a fixed $k = 2$, we obtain $\phi \wedge \texttt{inil} \wedge \neg\alpha$ unsatisfiable, but $\phi \wedge \neg\texttt{inil} \wedge \neg\alpha$ satisfiable: we cannot violate an assertion, but our analysis is not conclusive as we breach the bound.

### 3.2.3 Briefly on Complexity

Although a full analysis of the complexity of our BMC technique is not in scope for this thesis, we will attempt to provide here a brief discussion on how complexity compares to the mainstream first-order technique.

Recalling Section 1.3.3, complexity of mainstream BMC is often linear on the size of the bound because state merging maintains a single path, and the branching factor of first-order programs is constant at two paths per conditional—in contrast to pure Symbolic Execution, which has an exponential growth in the number of paths. In the higher-order case, a constant branching factor is no longer the case, as dynamic method creation coupled with state merging causes information loss along the path, which in turn requires the [**xv case**] introduced previously. Thus, in contrast to first-order BMC, higher-order BMC has a growing branching factor that grows in proportion to the number of methods dynamically introduced between symbolic method calls. While state merging still occurs after exhaustive method application, the increasing branching factor means that, even if a single path is maintained by merging after branching, complexity is no longer linear on the bound because every symbolic call is able to introduce a number of paths that is now proportional to some function of the size of the term and the depth of exploration.

To illustrate this, consider the simple case of a method $m$ that introduces an empty method when called and then branches conditionally before calling one of the methods defined so far. The conditional branch and state merging obfuscates the method to choose for the call, which triggers exhaustive method application. The branching factor is thus bounded by the number of methods present in the repository at the call site, which corresponds to the triangular number on either the depth of recursive calls or the size of the term. Thus, we have $(k^2 + 1)/2$ paths that need to be explored at each function call, where $k$ corresponds to the number of symbolic method applications that have occurred so far along the top-level path. Since every symbolic call involves every method created so far—including $m$—every call creates one more method that needs to be recorded. Generalising this, we have $b = (nk^2 - nk + 2k)/2$, for $k$ method applications and $n$ methods created per application. This means that the complexity of producing a formula is dependent on the program we are trying to analyse—specifically, most heavily in the branching factor $b$ for all the [**xv cases**] present in the program.

Taking the above into consideration, although the complexity of building the formula is inherently dependent on the program analysed, one can see that the worst case would arise from a program where the branching factor $b$ dominates the behaviour—e.g. creates new methods and triggers exhaustive method application every time the bound is consumed.

## 3.3 Soundness of the BMC Procedure

Intuitively, soundness states that, within a given bound, the algorithm reports either an error or the possibility of exceeding the bound, if and only if a corresponding concrete execution within that bound exists such that the error is reached or the bound is exceeded respectively. Soundness of the algorithm depends on its correctness, which states that every reduction reaching a final value via the operational semantics is captured by the BMC translation.

We start off with some definitions. An ***assignment*** $\sigma : \mathtt{Vars} \rightharpoonup \mathtt{CVals}$ is a finite map from variables to closed values. Given a term $M$, we write $M\{\sigma\}$ for the term obtained by applying $\sigma$ to $M$. On the other hand, applying $\sigma$ to a method repository $R$, we obtain the repository $R\{\sigma\} = \{m \mapsto R(m)\{\sigma\} \mid m \in \mathrm{dom}(R)\}$ — and similarly for stores $S$. Then, given a valid configuration $(M, R, S, k)$, we have $(M, R, S, k)\{\sigma\} = (M\{\sigma\}, R\{\sigma\}, S\{\sigma\}, k)$. Using program variables as logic variables, we can use assignments as logic assignments. Given a formula $\phi$, we let $\sigma \models \phi$ mean that the formula $\phi\{\sigma\}$ is valid (its negation is unsatisfiable). We say $\sigma$ ***represents*** $\phi$, and write $\sigma \simeq \phi$, if $\sigma \vDash \phi$ and $\phi \implies x = \sigma(x)$ is valid for all $x \in dom(\sigma)$.

**Theorem 3.5** (Soundness)**.** Given a valid configuration $(M, R, S, k)$ whose open variables are of ground type, suppose $[\![M, R, S, k]\!] = (ret, \phi, R'', C, D, \alpha, pc)$. Then, for all assignments $\sigma$ closing $(M, R, S, k)$, 1 and 2 are equivalent:

1. $\exists E, R', S'.(M, R, S, k)\{\sigma\} \twoheadrightarrow (E[\mathrm{assert0}], R', S', k')$
2. $\exists \sigma' \supseteq \sigma.\ \sigma' \models \phi \wedge \mathtt{inil} \wedge \neg\alpha.$

Moreover, if $\phi \wedge \texttt{inil} \wedge \neg\alpha$ is not satisfiable then 3 and 4 are equivalent:

3. $\exists M', R', S'.(M, R, S, k)\{\sigma\} \twoheadrightarrow (M', R', S', \texttt{nil})$
4. $\exists \sigma' \supseteq \sigma.\ \sigma' \models \phi \wedge \neg\texttt{inil} \wedge \neg\alpha.$

*Proof.* (1) $\implies$ (2) and (3) $\implies$ (4) follow directly from Lemma 3.6 below. For the reverse directions, we rely on the fact that the semantics is bounded so, in every case, $(M, R, S, k)\{\sigma\}$ should either reach a value, or a failed assertion, or breach the bound. Moreover, the semantics is deterministic, in the sense that the final configurations reached may only differ in the choice of fresh names used in the process. This allows us to employ Lemma 3.6 also for the reverse directions. □

The result above makes use of the lemmas that follow.

**Lemma 3.6** (Correctness)**.** Given $M, R, C, D, \phi, \alpha, pc, k, \sigma$ such that $\sigma \simeq \phi$, $(M, R, D, k)\{\sigma\}$ is valid, and $[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$ there exists $\sigma' \supseteq \sigma$ such that $\sigma' \simeq \phi'$ and:

- if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (v, \hat{R}, \hat{S}, \hat{k})$ then $\sigma' \vDash (pc \implies (pc' \wedge (ret = v))) \wedge ((\alpha \wedge pc) \implies \alpha')$, $R'\{\sigma'\} \supseteq \hat{R}$ and $D'\{\sigma'\} = \hat{S}$
- if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (E[\texttt{assert0}], \hat{R}, \hat{S}, \hat{k})$ then $\sigma' \vDash (\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha'$
- if $(M, R, D, k)\{\sigma\} \twoheadrightarrow (\hat{M}, \hat{R}, \hat{S}, \texttt{nil})$ then $\sigma' \vDash (pc \implies \neg pc') \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha') \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha')$.

*Proof.* Consider $\sigma \simeq \phi$, a valid configuration $(M, R, D, k)\{\sigma\}$, and its corresponding translation $[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$.

Now, by induction on the length of the transition sequence produced by the operational semantics, and, lexicographically, by induction on the size of the term, we have the following cases for configurations in canonical form.

**Terminal configurations:**

- $k = \texttt{nil}$.
  From the operational semantics we have $(M, R, D, \texttt{nil})\{\sigma\}$ as a stuck configuration. From the translation we have

$$[\![M, R, C, D, \phi, \alpha, pc, k]\!] = \\ (ret, (ret = \texttt{dval}) \wedge \phi, R, C, D, \alpha \wedge (pc \implies \texttt{inil}), \bot)$$

  We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = \texttt{dval}) \wedge \phi))$ and

$$\sigma' \vDash (pc \implies \neg\bot)$$

$$\begin{aligned}&\wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies (\alpha \wedge (pc \implies \texttt{inil})))\\&\wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg(\alpha \wedge (pc \implies \texttt{inil})))\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto \texttt{dval}]$. Since $\sigma \simeq \phi$, and since $ret$ is the only new variable in $\phi'$, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies \top) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies (\alpha \wedge (pc \implies \texttt{inil}))) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg(\alpha \wedge (pc \implies \texttt{inil})))$ holds.

- $M = \texttt{assert}0$.
  From the operational semantics we have $(\texttt{assert}0, R, D, k)\{\sigma\}$ as a stuck configuration. From the translation we have

$$\begin{aligned}&[\![\texttt{assert}0, R, C, D, \phi, \alpha, pc, k]\!] =\\&\quad(ret, (ret = ()) \wedge \phi, R, C, D, \alpha \wedge (pc \implies (0 \neq 0)), \top)\end{aligned}$$

  We thus want show that $\exists\sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = ()) \wedge \phi))$ and

$$\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \neg(\alpha \wedge (pc \implies (0 \neq 0))))$$

  Let us choose $\sigma' = \sigma[ret \mapsto ()]$. Since $\sigma \simeq \phi$, and $ret$ is the only new variable in $\phi'$, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \neg(\alpha \wedge (pc \implies \bot)))$ holds.
- $M = v$.
  From the operational semantics we have $(v, R, D, k)\{\sigma\}$ as a stuck configuration. From the translation we have

$$\begin{aligned}&[\![v, R, C, D, \phi, \alpha, pc, k]\!] =\\&\quad(ret, (ret = v) \wedge \phi, R, C, D, \alpha, \top)\end{aligned}$$

  We thus want show that $\exists\sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = v) \wedge \phi))$ and

$$\begin{aligned}&\sigma' \vDash (pc \implies (\top \wedge (ret = v))) \wedge ((\alpha \wedge pc) \implies \alpha)\\&R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

  Let us choose $\sigma' = \sigma[ret \mapsto v]$. Since $\sigma \simeq \phi$, and $ret$ is the only new variable in $\phi'$, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (\top \wedge (ret = v))) \wedge ((\alpha \wedge pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.
- $M = (\!|v|\!)$. Similar to the case above.

**Non-terminal configurations:**

- $M = \texttt{assert}i$ where $i \neq 0$.

From the operational semantics we have

$$(\texttt{assert}i, R, D, k)\{\sigma\} \to ((), R, D, k)\{\sigma\}$$

From the translation we have

$$\begin{aligned}&[\![\texttt{assert}i, R, C, D, \phi, \alpha, pc, k]\!] =\\ &\quad (ret, (ret = ()) \land \phi, R, C, D, \alpha \land (pc \implies (i \neq 0)), \top)\end{aligned}$$

We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = ()) \land \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = ())) \land ((\alpha \land pc) \implies (\alpha \land (pc \implies (i \neq 0))))\\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto ()]$. Since $\sigma \simeq \phi$, and $ret$ is the only new variable in $\phi'$, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = ()))$ holds because $\sigma'$ maps $ret$ to $()$, and $\sigma' \vDash ((\alpha \land pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.

- $M = r := v$.

From the operational semantics we have

$$(r := v, R, D, k)\{\sigma\} \to ((), R, D[r \mapsto v], k)\{\sigma\}$$

From the translation we have

$$\begin{aligned}&[\![r := v, R, C, D, \phi, \alpha, pc, k]\!] =\\ &\quad \text{let } C' = C[r] \text{ in let } D[r \mapsto C'(r)] \text{ in}\\ &\quad (ret, (ret = ()) \land (D'(r) = v) \land \phi, R, C', D', \alpha, \top)\end{aligned}$$

We thus want show that $\exists \sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = ()) \land (D'(r) = v) \land \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = ())) \land ((\alpha \land pc) \implies (\alpha \land (pc \implies (i \neq 0))))\\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D[r \mapsto C'(r)]\{\sigma'\} = D[r \mapsto v]\{\sigma'\}\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto (), C'(r) \mapsto v]$. Since $\sigma \simeq \phi$, $ret$ is the only new variable in $\phi'$, and $D'$ maps $r$ to $C'(r)$, which $\sigma'$ maps to $v$, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = ()))$ holds because $\sigma'$ maps $ret$ to $()$, and $\sigma' \vDash ((\alpha \land pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ holds trivially, and $D[r \mapsto C'(r)]\{\sigma'\} = D[r \mapsto v]\{\sigma'\}$ holds because $\sigma'$ maps $C'(r)$ to $v$.

- $M = !r$.

From the operational semantics we have

$$(!r, R, D, k)\{\sigma\} \to (v, R\{\sigma\}, D\{\sigma\}, k) \quad \text{where } v = D\{\sigma\}(r) = \sigma(D(r))$$

From the translation we have

$$\begin{aligned}&[\![!r, R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad (ret, (ret = D(r)) \land \phi, R, C, D, \alpha, \top)\end{aligned}$$

We thus want show that $\exists\sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = D(r)) \land \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = v)) \land ((\alpha \land pc) \implies \alpha) \\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto D(r)]$. Since $\sigma \simeq \phi$, and $ret$ is the only new variable in $\phi'$, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = v))$ holds because $\sigma' \supset \sigma$, $\sigma(D(r)) = v$ and $\sigma'(ret) = D(r)$, and we know $\sigma' \vDash ((\alpha \land pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.

- $M = v_1 \oplus v_2$.
  From the operational semantics we have

$$(v_1 \oplus v_2, R, D, k)\{\sigma\} \to (v, R\{\sigma\}, D\{\sigma\}, k) \quad \text{where } v = \sigma(v_1) \oplus \sigma(v_2)$$

From the translation we have

$$\begin{aligned}&[\![v_1 \oplus v_2, R, C, D, \phi, \alpha, pc, k]\!] = \\ &\quad (ret, (ret = (v_1 \oplus v_2)) \land \phi, R, C, D, \alpha, \top)\end{aligned}$$

We thus want show that $\exists\sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = (v_1 \oplus v_2)) \land \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = v)) \land ((\alpha \land pc) \implies \alpha) \\ &R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto v]$. Since $\sigma \simeq \phi$, and $ret$ is the only new variable in $\phi'$, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = v))$ holds because $\sigma' \supseteq \sigma$, $\sigma(v_1) \oplus \sigma(v_2) = v$ and $\sigma'$ maps $ret$ to $v$. Lastly, we know $\sigma' \vDash ((\alpha \land pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.

- $M = \pi_i\, v$.
  From the operational semantics we have

$$(\pi_i\, v, R, D, k)\{\sigma\} \to (v_i, R\{\sigma\}, D\{\sigma\}, k) \quad \text{where } \sigma(v) = \langle v_1, v_2\rangle$$

From the translation we have

$$\begin{aligned}&[\![\pi_i\, v, R, C, D, \phi, \alpha, pc, k]\!] =\\&\quad (ret, (ret = (\pi_i\, v)) \wedge \phi, R, C, D, \alpha, \top)\end{aligned}$$

We thus want show that $\exists\sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = (\pi_i\, v)) \wedge \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = v_i)) \wedge ((\alpha \wedge pc) \implies \alpha)\\&R\{\sigma'\} \supseteq R\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto v_i]$. Since $\sigma \simeq \phi$, and $ret$ is the only new variable in $\phi'$, we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = v))$ holds because $\sigma' \supseteq \sigma$, $\sigma(\pi_i\, v) = v_i$ and $\sigma'$ maps $ret$ to $v_i$. Lastly, we know $\sigma' \vDash ((\alpha \wedge pc) \implies \alpha)$ holds. Additionally, $R\{\sigma'\} \supseteq R\{\sigma'\}$ and $D\{\sigma'\} = D\{\sigma'\}$ hold trivially.

- $M = \lambda x.N$.
  From the operational semantics we have

$$(\lambda x.N, R, D, k)\{\sigma\} \to (\hat{m}, R[\hat{m} \mapsto \lambda x.N]\{\sigma\}, D\{\sigma\}, k)$$

From the translation we have

$$\begin{aligned}&[\![\lambda x.N, R, C, D, \phi, \alpha, pc, k]\!] =\\&\quad (ret, (ret = m) \wedge \phi, R[m \mapsto \lambda x.N], C, D, \alpha, \top)\end{aligned}$$

We thus want show that $\exists\sigma' \supseteq \sigma$ such that $(\sigma' \simeq ((ret = m) \wedge \phi))$ and, since this is a valid transition sequence, we want

$$\begin{aligned}&\sigma' \vDash (pc \implies (ret = \hat{m})) \wedge ((\alpha \wedge pc) \implies \alpha)\\&R[m \mapsto \lambda x.N]\{\sigma'\} \supseteq R[\hat{m} \mapsto \lambda x.N]\{\sigma'\} \text{ and } D\{\sigma'\} = D\{\sigma'\}\end{aligned}$$

Let us choose $\sigma' = \sigma[ret \mapsto m]$. Since $\sigma \simeq \phi$ and $ret$ is the only new variable in $\phi'$, and choosing $\hat{m}$ such that $\hat{m} = m$ by Lemma 2.6 (nominal determinism of the operational semantics), we know that $\sigma' \simeq \phi'$. We also know that $\sigma' \vDash (pc \implies (ret = \hat{m}))$ holds because $\sigma' \supset \sigma$, $m = \hat{m}$ by Lemma 2.6, and $\sigma'$ maps $ret$ to $m$. Lastly, we know $\sigma' \vDash ((\alpha \wedge pc) \implies \alpha)$ holds. Additionally, $D\{\sigma'\} = D\{\sigma'\}$ holds trivially, and $R[m \mapsto \lambda x.N]\{\sigma'\} \supseteq R[\hat{m} \mapsto \lambda x.N]\{\sigma'\}$.

- $M = mv$.
  From the operational semantics we have

$$(mv, R, S, k)\{\sigma\} \to (N\{v/x\}, R, S, k-1)\{\sigma\}$$

where $R(m) = \lambda x.N$.

From the translation we have

$$[\![mv, R, C, D, \phi, \alpha, pc, k]\!] = [\![N\{v/x\}, R, C, D, \phi, \alpha, pc, k-1]\!]$$

As such, this case holds directly from the inductive hypothesis.

- $M = \texttt{letrec}\ f = \lambda x.N\ \texttt{in}\ M'$.
  From the operational semantics we have

  $$(\texttt{letrec}\ f = \lambda x.N\ \texttt{in}\ M', R, S, k)\{\sigma\} \to (M'\{m/f\}, R[m \mapsto \lambda x.N\{m/f\}], S, k)\{\sigma\}$$

  where $R(m) = \lambda x.N$.
  From the translation we have

  $$\begin{aligned}&[\![\texttt{letrec}\ f = \lambda x.N\ \texttt{in}\ M', R, C, D, \phi, \alpha, pc, k]\!] =\\ &\quad [\![M\{m/f\}, R[m \mapsto \lambda x.N\{m/f\}], C, D, \phi, \alpha, pc, k]\!]\end{aligned}$$

  We know $\sigma \simeq \phi$. As such, this case holds directly from the inductive hypothesis.
- $M = (\texttt{let}\ x = v\ \texttt{in}\ M')$. Similar to the case above.
- $M = \texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0$.
  From the operational semantics we have

  $$(\texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0, R, S, k)\{\sigma\} \to (M_i, R, S, k)\{\sigma\}$$

  where $i = 0$ if $\sigma(v) = 0$, and $i = 1$ otherwise.
  From the translation we have

  $$\begin{aligned}&[\![\texttt{if}\ v\ \texttt{then}\ M_1\ \texttt{else}\ M_0, R, C, D, \phi, \alpha, pc, k]\!] =\\ &\quad \text{let } (ret_0, \phi_0, R_0, C_0, D_0, \alpha_0, pc_0) = [\![M_0, R, C, D, \phi, \alpha, pc \wedge (v = 0), k]\!] \text{ in}\\ &\quad \text{let } (ret_1, \phi_1, R_1, C_1, D_1, \alpha_1, pc_1) = [\![M_1, R_0, C_0, D, \phi_0, \alpha_0, pc \wedge (v \neq 0), k]\!] \text{ in}\\ &\quad \text{let } C' = C_1[r_1] \cdots [r_n]\ (\mathit{\Pi} = \{r_1, \ldots, r_n\}) \text{ in}\\ &\quad \text{let } \psi_0 = (v = 0) \implies ((ret = ret_0) \wedge \bigwedge_{r \in \mathit{\Pi}} (C'(r) = D_0(r))) \text{ in}\\ &\quad \text{let } \psi_1 = (v \neq 0) \implies ((ret = ret_1) \wedge \bigwedge_{r \in \mathit{\Pi}} (C'(r) = D_1(r))) \text{ in}\\ &\quad (ret, \psi_0 \wedge \psi_1 \wedge \phi_1, R, C', C', \alpha_1, ((pc_0 \wedge (v = 0)) \vee (pc_1 \wedge (v \neq 0))))\end{aligned}$$

  We now have two cases for $\sigma(v)$:

  1. $\sigma(v) = 0$.
     (1) By the inductive hypothesis we have $\sigma_0 \supseteq \sigma$, where $\sigma_0 \simeq \phi_0$, and all necessary conditions $P_0$ are satisfied.
     (2) By Lemma 3.7, we know $\exists \sigma_1 \supseteq \sigma_0.(\sigma_1 \simeq \phi_1)$.
     Let $\sigma' = \sigma_1[ret \mapsto ret_0, C'(r) \mapsto D_0(r)]$. Since $\sigma' \supset \sigma_1 \supseteq \sigma_0$, we know that $P_0$ is also satisfied. Thus, this case holds by (1) and (2).
  2. $\sigma(v) \neq 0$.
     (1) By Lemma 3.7, we know $\exists \sigma_0 \supseteq \sigma.(\sigma_0 \simeq \phi_0)$.

By Lemma 3.8, we know $R$ must be preserved in $R_0$, so $R_0\{\sigma_0\} \supseteq R$. Thus, by the inductive hypothesis:
(2) we have $\sigma_1 \supseteq \sigma_0 \supseteq \sigma$, where $\sigma_1 \simeq \phi_1$, and all necessary conditions $P_1$ are satisfied.
Let $\sigma' = \sigma_1[ret \mapsto ret_1, C'(r) \mapsto D_1(r)]$. Since $\sigma' \supset \sigma_1$, we know that $P_1$ is also satisfied. Thus, this case holds by (1) and (2).

- $M = xv$.
  From the operational semantics we have

$$(xv, R, S, k)\{\sigma\} \rightarrow (N_i\{v/y_i\}, R, S, k-1)\{\sigma\}$$

where $\sigma(x) = m_i$ and $R(m_i = \lambda y_i.N_i)$.
Let $\sigma_0 = \sigma$.
From the translation we have

```
$[\![x^\theta\, v, R, C, D, \phi, \alpha, pc, k]\!] =$
  if $R \restriction \theta = \emptyset$ then $(ret, (ret = \texttt{nil}) \wedge \phi, R, C, D, \alpha, pc)$ else
  let $R \restriction \theta$ be $\{m_1, ..., m_n\}$ and $(R, C, \phi, \alpha)$ be $(R_0, C_0, \phi_0, \alpha_0)$ in
  for each $i \in \{1, ..., n\}$ :
    let $R(m_i)$ be $\lambda y_i.N$ in
    let $(ret_i, \phi_i, R_i, C_i, D_i, \alpha_i, pc_i) =$
        $[\![N_i\{v/y_i\}, R_{i-1}, C_{i-1}, D, \phi_{i-1}, \alpha_{i-1}, pc \wedge (x = m_i), k-1]\!]$ in
  let $C'_n = C_n[r_1]\cdots[r_j]\ (\mathit{\Pi} = \{r_1, \ldots, r_j\})$ in
  let $\psi = \bigwedge_{i=1}^{n} \left( \begin{array}{l} (x = m_i) \implies \\ ((ret = ret_i) \wedge \\ \bigwedge_{r \in \mathit{\Pi}} (C'_n(r) = D_i(r))) \end{array} \right)$ in
  let $pc'_n = \bigvee_{i=1}^{n} (pc_i \wedge (x = m_i))$ in
  $(ret, \psi \wedge \phi_n, R_n, C'_n, C'_n, \alpha_n, pc'_n)$
```

Since $R \restriction \theta = \{m_1, \ldots, m_n\}$, it must be the case that $i \in \{1 \ldots n\}$.
It must be the case then that either (1) $i = 1$ or (2) $1 < i \leq n$.

1. (1) By the inductive hypothesis, we have $\exists \sigma_1 \supseteq \sigma_0.(\sigma_1 \simeq \phi_1)$, and all necessary properties $P_0$ hold.
   (2) By Lemma 3.7 applied repeatedly, we have $\exists \sigma_n \supseteq \cdots \supseteq \sigma_1.(\sigma_n \simeq \phi_n)$.
   Since $\sigma_n \supseteq \sigma_1$, properties $P_0$ hold, so this case holds by (1) and (2).
2. (1) By Lemma 3.7 applied repeatedly, we know $\exists \sigma_{i-1} \supseteq \cdots \supseteq \sigma_0.(\sigma_{i-1} \simeq \phi_{i-1}) \wedge \cdots \wedge (\sigma_0 \simeq \phi_0)$.
   (2) By Lemma 3.8 applied repeatedly, we also know $R_{i-1} \supseteq \cdots \supseteq R_0$.
   (3) By the inductive hypothesis, we know $\exists \sigma_i \supseteq \sigma_{i-1}.(\sigma_i \simeq \phi_i)$ and the necessary properties $P_i$ hold.

(4) By Lemma 3.7 again applied repeatedly, we know $\exists \sigma_n \supseteq \cdots \supseteq \sigma_i.(\sigma_n \simeq \phi_n) \wedge \cdots \wedge (\sigma_i \simeq \phi_i)$.

Since $\sigma_n \supseteq \sigma_i$, properties $P_i$ hold, so this holds by (1), (2), (3) and (4).

- $M = (\texttt{let}\ x = M'\ \texttt{in}\ M'')$, with $M'$ not a value.

Let us write $M$ as $E[M']$. From the operational semantics we have

$$(E[M'], R, S, k)\{\sigma\} \rightarrow (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow \ldots$$

where $(M', R, S, k)\{\sigma\} \rightarrow (\hat{M}, \hat{R}, \hat{S}, \hat{k})\{\sigma\}$.

We now have the following translation.

$$
\begin{aligned}
&[\![E[M'], R, C, D, \phi, \alpha, pc, k]\!] = \\
&\quad \text{let } (ret_1, \phi_1, R_1, C_1, D_1, \alpha_1, pc_1) = [\![M', R, C, D, \phi, \alpha, pc, k]\!] \text{ in} \\
&\quad \text{let } (ret_2, \phi_2, R_2, C_2, D_2, \alpha_2, pc_2) = [\![E[ret_1], R_1, C_1, D_1, \phi_1, \alpha_1, pc \wedge pc_1, k]\!] \text{ in} \\
&\quad (ret_2, \phi_2, R_2, C_2, D_2, \alpha_2, pc_1 \wedge pc_2)
\end{aligned}
$$

Since $E[M']$ can lead to either (1) some value, (2) an assertion `assert0`, or (3) a stuck configuration where the bound is `nil`, we have three cases to consider.

1. $(E[M'], R, S, k)\{\sigma\} \rightarrow (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1})$.

   (1) By the inductive hypothesis: $\sigma_1 \vDash (pc \implies (pc_1 \wedge (ret_1 = \hat{v_1}))) \wedge ((pc \wedge \alpha) \implies \alpha_1)$ and $R_1\{\sigma_1\} \supseteq R, D_1\{\sigma_1\} = \hat{S}$.

   $(E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1}) \twoheadrightarrow (\hat{v}', \hat{R}', \hat{S}', \hat{k})'$.

   (2) By the inductive hypothesis: $\sigma_2 \vDash (pc_1 \implies (pc_2 \wedge (ret_2 = \hat{v}'))) \wedge ((pc_1 \wedge \alpha_1) \implies \alpha_2)$ and $R_2\{\sigma_2\} \supseteq \hat{R_1}, D\{\sigma_1\} = \hat{S}$.

   Since $\sigma_2 \supseteq \sigma$, we know $\sigma_2 \simeq \phi_2$.

   From (1) and (2), we know $\sigma_2 \vDash (pc \implies (pc_1 \wedge pc_2 \wedge (ret = \hat{v}\ ))) \wedge ((pc \wedge \alpha) \implies \alpha_2)$. Case holds.

2. (a) $(E[M'], R, S, k)\{\sigma\} \rightarrow (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\texttt{assert0}], \hat{R_1}, \hat{S_1}, \hat{k_1})$.

   (1) By the inductive hypothesis: $\sigma_1 \vDash ((pc \wedge \alpha \wedge \texttt{inil}) \implies \neg\alpha_1)$ such that $\sigma_1 \simeq \phi_1$.

   (2) By Lemma 3.7, we know $\exists \sigma_2 \supseteq \sigma_1.\sigma_2 \simeq \phi_2$

   By Lemma 3.9, and (1) and (2), we know $\alpha_2 \implies \alpha_1$, thus $\neg\alpha_1 \implies \neg\alpha_2$. We therefore have $\sigma_1 \vDash ((pc \wedge \alpha \wedge \texttt{inil}) \implies \neg\alpha_2)$. Case holds.

   (b) $(E[M'], R, S, k)\{\sigma\} \rightarrow (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1})$.

   (1) By the inductive hypothesis: $\sigma_1 \vDash (pc \implies (pc_1 \wedge (ret_1 = \hat{v_1}))) \wedge ((pc \wedge \alpha) \implies \alpha_1)$ and $R_1\{\sigma_1\} \supseteq R, D_1\{\sigma_1\} = \hat{S}$.

   $(E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1}) \twoheadrightarrow (E'[\texttt{assert0}], \hat{R}', \hat{S}', \hat{k})'$

   (2) By the inductive hypothesis: $\sigma_2 \vDash ((pc \wedge pc_1 \wedge \alpha_1 \wedge \texttt{inil}) \implies \neg\alpha_2)$ such that $\sigma_2 \simeq \phi_2$.

From (1) and (2) we have $\sigma_2 \vDash (pc \implies \neg(pc_1 \wedge pc_2)) \wedge (\alpha \wedge pc \wedge \texttt{inil}) \implies \neg\alpha_2$. Case holds.

3. (a) $(E[M'], R, S, k)\{\sigma\} \to (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\hat{M_1}], \hat{R_1}, \hat{S_1}, \texttt{nil})$.

   (1) By the inductive hypothesis: $\sigma_1 \vDash (pc \implies \neg pc_1) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha_1) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha_1)$ such that $\sigma_1 \simeq \phi_1$.
   (2) By Lemma 3.7, we know $\exists \sigma_2 \supseteq \sigma_1 . \sigma_2 \simeq \phi_2$.
   (3) By Lemma 3.9, we know $\alpha_2 \implies \alpha_1$, so $\neg\alpha_1 \implies \neg\alpha_2$.

   From (1) we have that $\sigma_1 \vDash (pc \implies \neg(pc_1 \wedge pc_2))$.
   From (1) and (3) we have that $\sigma_1 \vDash ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha_2) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha_2)$.
   From (2) we have that $\sigma_2 \vDash (pc \implies \neg(pc_1 \wedge pc_2)) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha_2) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha_2)$ such that $\sigma_2 \simeq \phi_2$. Case holds.

   (b) $(E[M'], R, S, k)\{\sigma\} \to (E[\hat{M}], \hat{R}, \hat{S}, \hat{k})\{\sigma\} \twoheadrightarrow (E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1})$.

   (1) By the inductive hypothesis: $\sigma_1 \vDash (pc \implies (pc_1 \wedge (ret_1 = \hat{v_1}))) \wedge ((pc \wedge \alpha) \implies \alpha_1)$ and $R_1\{\sigma_1\} \supseteq R, D_1\{\sigma_1\} = \hat{S}$.

   $(E[\hat{v_1}], \hat{R_1}, \hat{S_1}, \hat{k_1}) \twoheadrightarrow (E'[\hat{M}'], \hat{R}', \hat{S}', \texttt{nil})'$
   (2) By the inductive hypothesis: $\sigma_2 \vDash ((pc \wedge pc_1) \implies \neg pc_2) \wedge ((\texttt{inil} \wedge \alpha_1 \wedge pc \wedge pc_1) \implies \alpha_2) \wedge ((\neg\texttt{inil} \wedge \alpha_1 \wedge pc \wedge pc_1) \implies \neg\alpha_2)$ such that $\sigma_2 \simeq \phi_2$.

   From (1) and (2) we have $\sigma_2 \vDash (pc \implies \neg(pc_1 \wedge pc_2)) \wedge ((\texttt{inil} \wedge \alpha \wedge pc) \implies \alpha_2) \wedge ((\neg\texttt{inil} \wedge \alpha \wedge pc) \implies \neg\alpha_2)$ such that $\sigma_2 \simeq \phi_2$. Case holds.

□

**Lemma 3.7** (Uniqueness of the translation)**.** Given an assignment $\sigma$ and formula $\phi$ such that $\sigma \cong \phi$, and a translation

$$[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$$

we know there exists some $\sigma' \supseteq \sigma$ such that $\sigma' \cong \phi'$.

*Proof.* Assuming $\sigma \cong \phi$, by induction on $k$ and then by induction on the size of $M$, we have the base cases:

1. $k = \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto \texttt{dval}]$.
2. $M = \texttt{assert}(v)$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto ()]$.
3. $M = v$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto v]$.
4. $M =!r$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto D(r)]$.
5. $M = \lambda x.N$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto m]$.
6. $M = \pi_i\, v$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto \pi_i\, v]$.
7. $M = v_1 \oplus v_2$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto v_1 \oplus v_2]$.

8. $M = r := v$ and $k \neq \texttt{nil}$: shown by choosing $\sigma' = \sigma[ret \mapsto (), D'(r) = v]$.

With base cases done, we have the following inductive cases:

1. $M = \texttt{let } x = M' \texttt{ in } M''$:
   (1) By the inductive hypothesis on $[\![M, R, C, D, \phi, \alpha, pc, k]\!]$, we have $\sigma_1 \simeq \phi_1$.
   (2) By the inductive hypothesis on $[\![M'\{ret_1/x\}, R_1, C_1, D_1, \phi_1, \alpha_1, pc \wedge pc_1, k]\!]$, we have $\sigma_2 \simeq \phi_2$.
   This case holds by (1) and (2).
2. $M = \texttt{letrec } f = \lambda x.N \texttt{ in } M'$:
   (1) By the inductive hypothesis on $[\![M'\{m/f\}, R', C, D, \phi, \alpha, pc, k]\!]$, we have $\sigma_1 \simeq \phi_1$.
   This case holds by (1).
3. $M = mv$:
   (1) By the inductive hypothesis on $[\![N\{v/x\}, R', C, D, \phi, \alpha, pc, k]\!]$, we have $\sigma_1 \simeq \phi_1$.
   This case holds by (1).
4. $M = \texttt{if } v \texttt{ then } M_1 \texttt{ else } M_0$:
   (1) By the inductive hypothesis on $[\![M_0, R, C, D, \phi, \alpha, pc \wedge (v = 0), k]\!]$, we have $\sigma_0 \simeq \phi_0$.
   (2) By the inductive hypothesis on $[\![M_1, R_0, C_0, D_0, \phi_0, \alpha_0, pc \wedge (v \neq 0), k]\!]$, we have $\sigma_1 \simeq \phi_1$.
   We now have two cases on $\sigma_1(v)$:
   (a) $\sigma_1(v) = 0$. Choose $\sigma' = \sigma_1[ret \mapsto ret_0, C'(r) \mapsto D_0(r)]$ for all $r \in \Pi$.
   (b) $\sigma_1(v) = i \neq 0$. Choose $\sigma' = \sigma_1[ret \mapsto ret_1, C'(r) \mapsto D_1(r)]$ for all $r \in \Pi$.
5. $M = xv$:
   (1) By the inductive hypothesis on
   $[\![N_1\{v/y_1\}, R_0, C_0, \ldots]\!]$ to $[\![N_n\{v/y_n\}, R_{n-1}, C_{n-1}, \ldots]\!]$,
   we have $\exists \sigma_n \supseteq \cdots \supseteq \sigma_1 \supseteq \sigma_0.(\sigma_n \simeq \phi_n) \wedge \cdots \wedge (\sigma_1 \simeq \phi_1) \wedge (\sigma_0 \simeq \phi_0)$.
   Since $\sigma(x) \in R$, let $\sigma(x) = m_i$ for $i \in \{1..n\}$.
   Let $\sigma' = \sigma_n[ret \mapsto ret_i, C'_n(r) \mapsto D_i(r)]$ for all $r \in \Pi$.
   This case holds by (1).

□

**Lemma 3.8** (Preservation of the repository)**.** Given a translation

$$[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$$

we know the input repository must be preserved; i.e. $R' \supseteq R$.

*Proof.* By inspection of the translation rules. □

**Lemma 3.9** (Propagation of preconditions)**.** Given a translation

$$[\![M, R, C, D, \phi, \alpha, pc, k]\!] = (ret, \phi', R', C', D', \alpha', pc')$$

we know that preconditions $\phi$ and $\alpha$ must be propagated and included in $\phi'$ and $\alpha'$; i.e. $\phi' = \psi \land \phi$ and $\alpha' = \beta \land \alpha$ where $[\![M, R, C, D, \top, \top, \top, k]\!] = (ret, \psi, R', C', D', \beta, pc'')$

*Proof.* By inspection of the translation rules. □

## 3.4 A Points-to Analysis for Names

The presence of exhaustive method application in our BMC translation is a primary source of state space explosion. As such, a more precise filtering of $R$ is necessary for scalability. In this section we describe a simple analysis to restrict the number of methods considered. We follow ideas from points-to analysis [7], which typically computes an overapproximation of the *points-to set* of each variable inside a program, that is, the set of locations that it may point to.

Our analysis computes the set of methods that may be bound to each variable while unfolding. We do this via a finite map $pt : (\texttt{Refs} \cup \texttt{Vars}) \rightharpoonup \texttt{Pts}$ where `Pts` contains all *points-to sets* and is given by: $\texttt{Pts} \ni A ::= X \mid \langle A, A\rangle$ where $X \subseteq_{fin} \texttt{Meths}$. Thus, a points-to set is either a finite set of names or a pair of points-to sets. These need to be updated whenever a method name is created, and are assigned to references or variables according to the following cases:

| | |
|---|---|
| $r := M$ | add in $pt$: $r \mapsto pt(M)$ |
| $\texttt{let}\ x = M\ \texttt{in}\, M'$ | add in $pt$: $x \mapsto pt(M)$ |
| $xM$ | add in $pt$: $ret(M) \mapsto pt(M)$ |

where $ret(M)$ is the variable assigned to the result of $M$. The `letrec` binder follows a similar logic. The need to have sets of names, instead of single names, in the range of $pt$ is that the analysis, being symbolic, branches on conditionals and applications, so the method pointed to by a reference cannot be decided during the analysis. Thus, when joining after branching, we merge the $pt$ maps obtained from all branches.

The points-to algorithm is presented in Figure 3.4. Given a valid configuration $(M, R, S, k)$, the algorithm returns $PT(M, R, S, k) = (ret, A, R, pt)$, where $A$ is the points-to set of $ret$, and $pt$ is the overall points-to map computed.

The merge of points-to maps is given by:

$$merge(pt_1, \ldots, pt_n) = \{x \mapsto \bigcup_i \hat{pt}_i \mid x \in \bigcup_i \mathrm{dom}(pt_i)\}$$

where $\hat{pt}_i(x) = pt_i(x)$ if $x \in dom(pt_i)$, $\emptyset$ otherwise, and $A \cup B$ is defined by $\langle A_1 \cup B_1, A_2 \cup B_2\rangle$ if $A, B = \langle A_1, A_2\rangle, \langle B_1, B_2\rangle$, and just $A \cup B$ otherwise.

```
Base Cases:
  $PT(M, R, pt, nil) = (ret, \varnothing, R, pt)$
  $PT(v, R, pt, k) = (ret, \varnothing, R, pt)$  where $v = i, ()$
  $PT(m, R, pt, k) = (ret, \{m\}, R, pt)$
  $PT(\lambda x.M, R, pt, k) = (ret, \{m\}, R[m \mapsto \lambda.M], pt)$
  $PT(x, R, pt, k) = (ret, pt(x), R, pt)$
  $PT(r := v, R, pt, k) = (ret, \varnothing, R, pt[r \mapsto pt(v)])$
  $PT(!r, R, pt, k) = (ret, pt(r), R, pt)$
  $PT(\pi_i\, v, R, pt, k) = (ret, \pi_i\, (pt(v)), R, pt)$
  $PT(\langle v_1, v_2 \rangle, R, pt, k) = (ret, \langle pt(v_1), pt(v_2) \rangle, R, pt)$
  $PT(v_1 \oplus v_2, R, pt, k) = (ret, \varnothing, R, pt)$
  $PT(\texttt{assert}v, R, pt, k) = (ret, \varnothing, R, pt)$
Inductive Cases:
  $PT(\texttt{let } x = M \texttt{ in } M', R, pt, k) =$
    let $(ret_1, A_1, R_1, pt_1) = PT(M, R, pt, k)$ in $PT(M'\{ret_1/x\}, R_1, pt_1[ret_1 \mapsto A_1], k)$
  $PT(\texttt{letrec } f = \lambda x.M \texttt{ in } M', R, pt, k) =$
    let $m$ be fresh in $PT(M'\{m/f\}, R[m \mapsto \lambda x.M\{m/f\}], pt, k)$
  $PT(m\, v, R, pt, k) =$ let $R(m)$ be $\lambda x.N$ in $PT(N\{v/x\}, R, pt, k-1)$
  $PT(\texttt{if } v \texttt{ then } M_1 \texttt{ else } M_0, R, pt, k) =$
    let $(ret_0, A_0, R_0, pt_0) = PT(M_0, R, pt, k)$ in
    let $(ret_1, A_1, R_1, pt_1) = PT(M_1, R_0, pt_b, k)$ in $(ret, A_0 \cup A_1, R_1, merge(pt_0, pt_1))$
  $PT(x^\theta\, v, R, pt, k) =$
    let $R$ be $R_0$ and $pt(x)$ be $\{m_1, ..., m_n\}$ in
    if $n = 0$ then $(ret, \emptyset, R_0, pt)$ else: for each $i \in \{1, ..., n\}$ :
      let $R(m_i)$ be $\lambda y_i.N$ in let $(ret_i, A_i, R_i, pt_i) = PT(N_i\{v/y_i\}, R_{i-1}, pt, k-1)$ in
      $(ret, A_1 \cup ... \cup A_n, R_n, merge(pt_1, \ldots, pt_n))$
```

Figure 3.4: The points-to analysis algorithm.

### 3.4.1 Comparison with Conventional Points-to Analyses

Two common kinds of points-to analyses are *Andersen-style* [7] and *Steengaard-style* [76] analyses, also called *inclusion-based* and *unification-based* flow-insensitive analyses respectively [35]. These are often implemented as constraint-based analyses that pass through the program code to allocate constraints for each reference/variable assignment, and subsequently solve these constraints in a global manner—as their names suggest, constraints are often based on inclusion or unification. Both analyses are typically context-insensitive and flow-insensitive algorithms. For example, let $pts(r)$ be the points-to set of $r$. The following are some cases and corresponding constraints for ANSI-C programs in Andersen-style analysis:

| | |
|---|---|
| `p := &x` | $x \in pts(p)$ |
| `p := q` | $pts(p) \supseteq pts(q)$ |

$$\texttt{*p := q} \qquad \forall r \in pts(p).pts(r) \supseteq pts(q)$$
$$\texttt{p := *q} \qquad \forall r \in pts(q).pts(p) \supseteq pts(r)$$

Our analysis has analogous rules for assignment and variable binding inspired on these kinds of analyses. However, in contrast to ANSI-C our references are a simpler setting without pointer arithmetic. More significantly, our analysis is intended to interface with the BMC traversal only for the purpose of recording flow of higher-order values at intermediate points. As such it is not a global solution, but a forward analysis that gives online results for the data-flow along the path so far. Moreover, while it does not over-approximate globally, it does, however, over-approximate after symbolic branching since the analysis does not perform intermediate calls to the solver—nor does it accumulate constraints to be solved. Globally, our points-to analysis is less precise than one that is fully flow sensitive, and is also *incomplete* as it only gives a depth-bounded result (e.g. if a reference is eventually assigned after $k+1$ steps, and we only look at $k$ steps, we would not know about said assignment). That said, the way our analysis is used makes it more precise than flow and context insensitive algorithms because within a given path, i.e. before merging after branching or returning from a method call, we have the advantage of flow and context sensitivity from the BMC traversal itself. Finally, since we only care about the flow of methods, a complete graph relating all references is unnecessary. For this reason, our points-to sets do not need to include references, but method names only.

### 3.4.2 The Optimised BMC Translation

We can now incorporate the points-to analysis in the BMC translation to get an optimised translation which operates on symbolic configurations augmented with a points-to map, and returns:

$$[\![M, R, C, D, \phi, \alpha, pc, pt, k]\!]_{PT} = (ret, \phi', R', C', D', \alpha, pc, A, pt')$$

The optimised BMC translation is defined by lock-stepping the two algorithms presented above (i.e. $[\![\cdot]\!]$ and $PT(\cdot)$) and letting $[\![\cdot]\!]$ be informed from $PT(\cdot)$ in the $xM$ case, which now restricts the choices of names for $x$ to the set $pt(x)$. Its soundness is proven along the same lines as the basic algorithm.

**Example 3.10.** To illustrate the significance of reducing the set of names, consider the following program which recursively generates names to compute triangular numbers.

```
letrec f = λ x.
  if x leq 0 then 0
  else let g = (λ y.x + y) in g (f (x-1))
in
letrec f' = λ x.if x leq 0 then 0 else x + (f' (x-1))
in assert(f n = f' n)
```

Without points-to analysis, since $f$ creates a new method per call, and the translation considers all methods of matching type per recursive call, the number of names to apply at depth $m \leq n$ when translating $f(n)$ is approximately $m!$. This means that the number of paths explored grows by the factorial of $n$, with the total number of methods created being the left factorial sum $!n$. In contrast, $f'(n)$ only considers $n$ names with a linear growth in number of paths. With points-to analysis, the number of names considered and created in $f$ is reduced to that of $f'$. $\Diamond$

### 3.4.3 Briefly on Complexity Once Again

The points-to analysis algorithm mirrors the BMC translation previously defined as it is designed to run step-locked with the translation in order to limit the branching incurred by exhaustive method application. As such, the complexity of our points-to analysis algorithm is identical to the translation algorithm up-to the exhaustive method application case. The difference is, of course, a direct result of the points-to analysis itself, which decreases the number of methods to consider for exhaustive method application.

The improvement depends on the nature of the program analysed. In the best case, all symbolic methods can be resolved precisely to one method body, which decreases branching at exhaustive method application from $n$ methods created so far to 1. In worst case, symbolic conditional branching obfuscates the assignment of the function body, and points-to analysis cannot do any better to distinguish the methods. In general, points-to analysis reduces the number of methods to consider down, from all methods created so far, to the smallest possible set of methods defined by *unresolvable symbolic branching* at the point of binding. By unresolvable symbolic branching we mean that the precise method body to use cannot be resolved. For instance, if a variable $x$ is bound to a method $m_1$ and $m_2$, where the choice is guarded by a symbolic expression that cannot be uniquely resolved either to true or false, then the smallest set of plausible methods bound to $x$ needs to consider both $m_1$ and $m_2$. This points-to analysis is thus a forward-analysis that propagates this smallest set around, combining unresolvable sets where necessary, where partial results during the analysis are used for exhaustive method application. Once combined with the translation, since they run step-locked, would run with exactly the same complexity in terms of number of paths explored—which dominates the space and time complexity. Notice also that the analysis would not call a solver by itself, as it does not accumulate constraints to be solved. As such, all symbolic branching would be considered unresolvable by our analysis.

## 3.5 Implementation

We implemented the translation algorithm in a prototype tool to model check higher-order programs called BMC-2. The implementation and benchmarks can be found at:

`https://github.com/LaifsV1/BMC-2`

The tool takes program source code written in an ML-like language, and produces a propositional formula in SMT-LIB 2 format. This is then be fed to an SMT solver such as Z3 [23]. Syntax of the input language is based on the subset of OCaml that corresponds to HORef. Differences between OCaml and our concrete syntax are for ease of parsing and lack of type inference. For instance, all input programs must be either written in "Barendregt Convention", meaning all bound variables must be distinct, and different from free variables. Additionally, all bound variables are annotated with types. Internally, BMC-2 implements an abstract syntax that extends HORef with vector arguments and integer lists. This means that functions can take multiple arguments at once. Lists are handled for testing, but not discussed here as they are not relevant to the theory. BMC-2 itself was written in OCaml.

To illustrate our input language, following is the sample program `mc91-e` from [45] translated from OCaml to our syntax.

```
Methods:
  mc91 (x:Int) :(Int) =
   if x >= 101 then x + -10
   else mc91 (mc91 (x + 11));
Main (n:Int) :(Unit):
  if n <= 102
  then assert((mc91 n)==91)
  else skip
```

The keyword `Methods` is used to define all methods in the repository. The keyword `Main` is used to define the main method. For this sample program, our tool builds a translation with $k = 1$ for which Z3 correctly reports that the assertion fails if $n = 102$. One can see on the example above that, for ease of parsing, we do not have a subtraction operator. Instead, we parse negative numbers and add them to subtract.

### 3.5.1 Tool Architecture and Usage

As of commit `66d0167`, BMC-2 consists of three OCaml files and two parser files to use with the Menhir LR(1) parser generator for OCaml. The files are organised as follows:

- `AbstractSyntax.ml` contains the abstract syntax for our tool, which corresponds to the data structures to hold program trees for HORef, and any helper functions necessary to process the different components of the program configurations.
- `Translation.ml` contains an implementation of the BMC translation in Figure 3.3. A simple implementation for variable substitution is provided in this file as well.
- `TopLevel.ml` contains the top level interface to our tool. This file thus combines all modules by first parsing the given file and then translating it into our BMC

encoding.

- `Lexer.mll` contains the lexer or tokeniser for use with Menhir. It tokenises according to a list of keywords and regular expressions.
- `Parser.mly` contains the grammar rules to use with Menhir to generate a parser for sugared HORef.

The tool itself does not include a type checking component; the program tree is immediately translated to be fed to a solver. Since we include no type inference, type annotations for all bound variables in the input program are necessary. The formula produced by our tool is then annotated with these types, which must then pass the solver's type checker.

BMC-2 is compiled using OCamlbuild (which automatically determines the sequence of calls to the OCaml compiler) with the Menhir option enabled:

```
ocamlbuild -I parser -use-menhir TopLevel.native
```

This requires an OCaml compiler version 4.04 or above and Menhir. From testing, as long as Menhir and OCamlbuild are present, the tool can be compiled on any Linux distribution and on Windows machines running the Cygwin POSIX-compatible environment.

Using this tool requires calling the compiled top level file with the desired parameters:

```
./TopLevel.native <file-path> <bound> <mode>
```

where `<file-path>` is the path for a file written in sugared HORef, `<bound>` is a natural number, and `<mode>` is either 1 or 0 for reachable bugs and reachable bounds respectively. This outputs a formula in SMT-LIB 2 which can be immediately fed to Z3. For instance:

```
./TopLevel.native mc91-e.txt 4 1 | z3 -in
```

feeds a translation of `mc91-e.txt` to Z3 with a bound 4 and is satisfiable if and only if an assertion violation is reachable.

**Remark 3.11.** While the presentation of our BMC algorithm in Figure 3.3 is for terms in canonical form, the tool itself parses arbitrary HORef terms in sugared form. The translation to canonical form is linear and can even be done seamlessly by the parser when constructing the abstract syntax tree.

**Example 3.12.** Consider the `mc91-e` example. We execute BMC-2 on this example with a bound $k = 1$ in bug-finding mode:

```
./TopLevel.native mc91-e.txt 1 1
```

which produces an SMT-LIB 2 formula involving 30 variables and 20 complex clauses (containing multiple disjunctions). Feeding this to Z3, we obtain the following output:

```
sat
((n 102))
```

which states that the formula is satisfiable when the input variable n $= 102$.

For this run (on a machine equipped with an 8th Gen Intel Core i7 clocked at 1.9GHz), total execution time from source-code to SMT-LIB 2 was approximately 0.001s, whereas Z3 was recorded to have taken 0.009s. Timing information for BMC-2 is reported by the tool as comments in the SMT-LIB formula it produces. ◊

## 3.6 Benchmarks

We tested our implementation on a set of 40 programs that include a selection from the MoCHi benchmark [45]. This is a set of higher-order programs written in OCaml, originally used to test the higher-order model checking tool MoCHi [45] and subsequently used for benchmarking [77, 15, 73]. We added custom samples with references (`ref-1`, `ref-1-e`, `ref-2`, `ref-2-e`, `ref-3`), as well as programs of varying lengths—100, 200 and 400 lines of code—constructed by combining the other samples. To combine programs, we refactored and concatenated methods and main methods from different files into a single file, and switch between the methods based on user input, thus forcing BMC-2 to consider all mains. In this set we have unsafe versions of safe programs denoted by the `-e` termination in their filename. Unsafe programs were constructed by slight modifications to the assertions of the original safe programs. For our experiment, the programs were manually translated to our input language and checked using our tool and Z3. Care was taken to keep all sample programs as close to the original source code as our concrete syntax allows. All experiments ran on an Ubuntu machine equipped with an Intel Core i7-6700 CPU clocked at 3.40GHz and 16GB RAM. All tests were set to time-out after 3 minutes, and up to a maximum bound $k = 15$. These limits were chosen due to the combinatorial nature of model checking and the sample programs used. BMC-2 ran twice per program per bound, and the average was recorded.

## 3.7 Evaluation

Figure 3.5 plots the average time taken for BMC-2 to check all the benchmark programs. We can observe that performance of BMC-2 heavily depends on the program it is checking, making the possibility of full verification entirely dependent on the nature of the program. For example, `ack`, which is an implementation of the Ackermann function, is a deeply

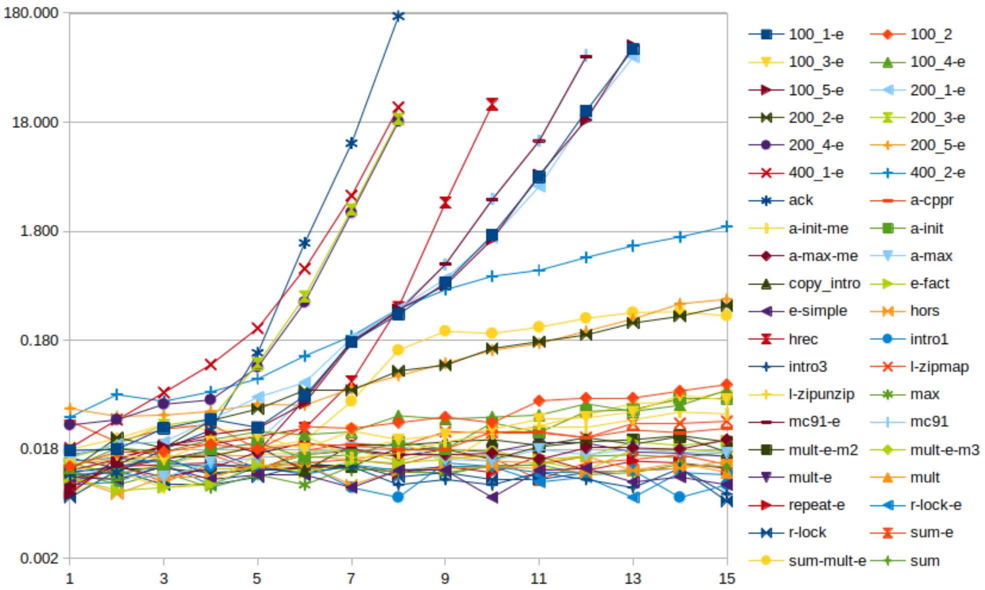


Figure 3.5: Average execution time(s) for BMC-2 vs. bounds $k = 1..15$.

recursive program that diverges rapidly, and thus cannot be translated by our algorithm any better than its normal growth. This agrees with the intuition that BMC is not appropriate to find bugs in deep recursion. In exchange, as mentioned before, BMC has been shown to be effective on shallow bugs in industry. This can be seen with our examples for 100 to 400 lines of code (LoC), which were correctly shown to have bugs with little difficulty despite the increase in program size (e.g. 0.03s on average for 100 LoC vs 0.08s for 400 LoC at $k = 4$). Our approach being bounded-complete means that, for safe programs, the tool correctly found that no bugs were reachable up to the bound even on the larger safe programs, again, with little difficulty (e.g. 0.02s at $k = 4$ vs 0.07s at $k = 15$ for `102_2.txt`).

From the fact some programs ran out of memory, we can observe that, unlike mainstream BMC approaches, the procedure is not linear in state-space with respect to the bound. Instead, we occasionally have exponential behaviour. This is entirely explained by dynamic method creation in some programs. Due to state merging, the mainstream approach can be linear on the bound if the branching factor is bounded throughout the evaluation. In a higher-order setting, where dynamic method creation is more common, this is not true, since the branching factor may increase if the control flow cannot be determined after the addition of a new method. This seems to suggest that BMC looses its advantages with regards to state merging when applied to higher-order programs, but faster compilation and stable behaviour on larger programs still suggests BMC is a practically useful alternative.

In addition to testing BMC-2, we also ran comparison experiments on prior tools MoCHi [45] and Rosette [79]. All experiments ran on the same machine used to test BMC-2. This will be described in more detail in the following sections. For Rosette, we used an implementation of our bounded semantics in Rosette provided to us by an

| | 4 | 7 | 10 | 13 | 15 | MoCHi |
|---|---|---|---|---|---|---|
| 100_1-e | 0.034 | 0.173 | 1.661 | 84.130 | - | c |
| 100_2 | 0.020 | 0.028 | 0.032 | 0.053 | 0.071 | c |
| 100_3-e | 0.021 | 0.027 | 0.028 | 0.040 | 0.051 | 10.734 |
| 200_1-e | 0.034 | 0.188 | 1.572 | 71.296 | - | m |
| 200_2-e | 0.033 | 0.063 | 0.151 | 0.259 | 0.372 | - |
| 200_3-e | 0.034 | 2.849 | - | - | m | 1.742 |
| 400_1-e | 0.108 | 3.805 | - | - | m | m |
| 400_2-e | 0.061 | 0.196 | 0.696 | 1.321 | 1.991 | - |
| ack | 0.027 | 11.519 | - | - | - | 0.525 |
| a-cppr | 0.031 | 0.020 | 0.026 | 0.027 | 0.028 | 28.584 |
| a-init | 0.018 | 0.016 | 0.032 | 0.042 | 0.053 | c |
| e-fact | 0.009 | 0.014 | 0.016 | 0.021 | 0.022 | 0.629 |
| e-simple | 0.010 | 0.008 | 0.007 | 0.009 | 0.009 | 0.098 |
| hrec | 0.020 | 0.075 | 26.175 | - | - | 0.867 |
| r-lock-e | 0.009 | 0.013 | 0.013 | 0.007 | 0.011 | 0.216 |
| ref-2 | 0.013 | 0.008 | 0.011 | 0.012 | 0.010 | u |
| ref-2-e | 0.011 | 0.013 | 0.010 | 0.013 | 0.011 | u |
| ref-3 | 0.019 | 0.018 | 0.047 | 0.211 | 0.211 | u |

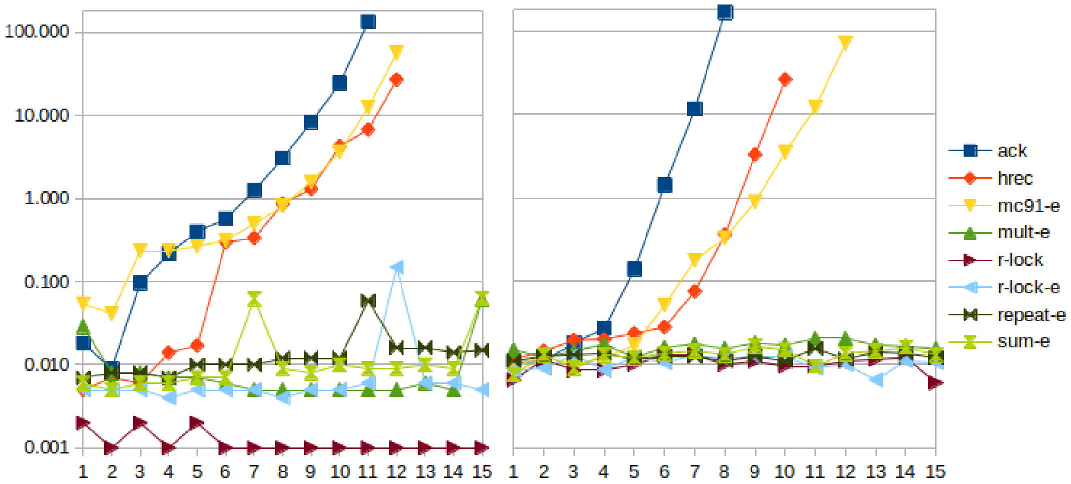
| | | | |
|---|---|---|---|
| $-$ | time-out | $c$ | crash |
| $m$ | out of memory | $u$ | unsupported |

Table 3.1: Execution time(s) for BMC-2 ($k = 4..15$) and MoCHi.

anonymous reviewer. With the semantics implemented, we compare Rosette's symbolic execution of HORef to BMC-2 with Z3's translation and solving of the same terms.

### 3.7.1 Comparison with MoCHi

Though the goals of each tool are different, we attempted to compare our approach to MoCHi. Being unable to build from source, we decided to used the Dockerfile on the Ubuntu machine from before. In Table 3.1, we have the time taken for BMC-2 and MoCHi for a smaller set of programs—the full range of results can be found in the tool page. We noticed that MoCHi is very sensitive to the operators and functions used in the assertions, while BMC-2 appears to be less dependent on these. For instance, checking `mult-e` with `assert(mult m m <= mult n n)` was three orders of magnitude slower than the original, while, at $k = 1$, BMC-2 takes 0.012 seconds; an increase of 20% from the original 0.010s to find a bug. We also noticed that MoCHi is less consistent with larger programs. For 100 to 400 lines of code, MoCHi correctly found bugs in 4 out of 12 samples, but halted unexpectedly on the remaining 8. BMC-2 found all 11 bugs of the 12 programs, and found no bugs in the safe program. Finally, we included 5 examples with references, which BMC-2 correctly checked, whereas MoCHi does not support state.



Depth of search: $k = 1 \ldots 15$. (Time excludes initialisation for Rosette)

Figure 3.6: Execution time(s) for Rosette (left) and BMC-2 (right) vs. search depth.

### 3.7.2 Comparison with Rosette for Racket

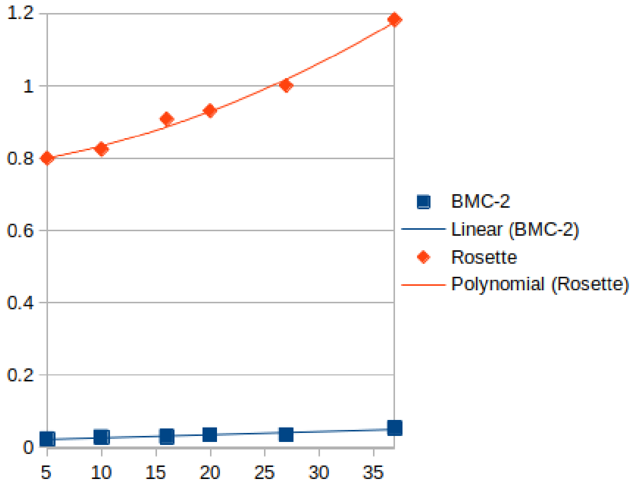
We found that BMC-2 and Rosette are very similar in their ability to check higher-order programs. Since Racket is a stateful higher-order language like HORef, and Rosette employs a symbolic virtual machine with symbolic execution techniques for Racket, we can expect this similarity. Fundamentally, Rosette and BMC-2 provide different approaches to verification as the former is related to symbolic evaluation, while the latter is a monolithic BMC translation. We were particularly interested in Rosette's ability to implement bounded verification for higher-order programs. With our bounding mechanism defined in Rosette, we compared its symbolic evaluation to BMC-2 on the Ubuntu machine. Figure 3.6 showcases a comparison based on 8 sample programs from the MoCHi benchmarks without counting initialisation time, while Table 3.2 and Figure 3.7 showcase a comparison on increasingly larger program size in terms of number of method definitions at $k = 2$ taking into account initialisation time. The latter comparison was made on a benchmark built by combining method definitions from other files, and then refactoring and repeating the definitions present in the file.

Rosette and BMC-2 are comparable in scalability, with BMC-2 being less optimised for small diverging programs such as `ack`. This could be due to the way Rosette performs *type-driven state merging*, which may allow it to be more memory efficient while still providing opportunities for concretisation. In contrast, we perform a suboptimal SSA transformation which could benefit from *dominance frontiers* for optimal merging of control flow. BMC, however, has the theoretical advantage of faster compilation time over symbolic execution [79], which can be observed in Table 3.2 and Figure 3.7. As the number of method definitions in the program increases, time taken for Rosette including initialisation appears to grow more steeply than BMC-2 does, which suggests initialisation time is not constant. Thus, while internal execution time of Rosette appears to scale similarly if not better than BMC-2 with regards to depth of search, it appears BMC-2

| Method definitions | Lines of code | Time (s) | Lines of code | Time (s) |
|---|---|---|---|---|
| 5 | 71 | 0.024 | 78 | 0.800 |
| 10 | 160 | 0.029 | 128 | 0.825 |
| 16 | 160 | 0.030 | 188 | 0.908 |
| 20 | 195 | 0.036 | 228 | 0.931 |
| 27 | 244 | 0.036 | 289 | 1.001 |
| 37 | 326 | 0.054 | 390 | 1.183 |

(Time includes initialisation for both tools, bound fixed at $k = 2$)

Table 3.2: Execution time(s) for BMC-2 (left) and Rosette (right) vs. program size.



(Time includes initialisation for both tools, bound fixed at $k = 2$)

Figure 3.7: Execution time(s) for BMC-2 and Rosette vs. program size.

can scale better in terms of program size measured by number of method definitions.

# Chapter 4

# Symbolic Games for Open Higher-Order Programs

In this chapter we present, to our knowledge, the first game-semantics-based approach to symbolic execution for reachability of assertion violations in open higher-order programs. We do this by defining a trace semantics for HOLi, the higher-order language introduced in Chapter 2, that models the environment of higher-order terms. We show that the trace semantics of libraries is sound and complete for reachability of errors under any definable client. We follow by defining a symbolic version of the trace semantics for the purpose of bug-finding through symbolic execution. We prove the symbolic semantics is sound with respect to errors found via a strong notion of equivalence with regards to the concrete semantics (bisimilarity), and implement a bounded version of the symbolic semantics in a prototype tool called HOLiK.

## 4.1 A Trace Semantics for HOLi

Recall the syntax of HOLi in Figure 2.1:

$$
\begin{array}{rrl}
\textit{Libraries} & L ::= & B \mid \texttt{abstract}\ m; L \\
\textit{Blocks} & B ::= & \varepsilon \mid \texttt{public}\ m = \lambda x.M; B \mid m = \lambda x.M; B \\
 & & \mid \texttt{global}\ r := i; B \mid \texttt{global}\ r := \lambda x.M; B \\
\textit{Terms} & M ::= & \texttt{assert}(M) \mid m \mid i \mid () \mid x \mid \lambda x.M \mid r := M \mid !r \mid M \oplus M \\
 & & \mid \langle M, M \rangle \mid \pi_1 M \mid \pi_2 M \mid MM \mid \texttt{if}\ M\ \texttt{then}\ M\ \texttt{else}\ M \\
 & & \mid \texttt{letrec}\ x = \lambda x.M\ \texttt{in}\ M \mid \texttt{let}\ x = M\ \texttt{in}\ M \\
\textit{Clients} & C\ ::= & L; \texttt{main} = M
\end{array}
$$

We extend the operational semantics of HOLi to handle libraries and terms that may call abstract methods, i.e. calls external methods that are not bound to a definition. The approach we follow is based on operational game semantics [41, 47, 33] and in particular

$$\text{(INT)}\quad \frac{(M,R,S,k) \to (M',R',S',k')}{(\mathcal{E},M,R,S,\mathcal{P},\mathcal{A},k)_p \to (\mathcal{E},M',R',S',\mathcal{P},\mathcal{A},k')_p}$$

$$\begin{array}{ll}
\text{(PQ)} & (\mathcal{E},E[mv],R,S,\mathcal{P},\mathcal{A},k)_p \xrightarrow{\mathtt{call}(m,v)} ((m,E) :: \mathcal{E},0,R,S,\mathcal{P}',\mathcal{A},k)_o \\
\text{(OQ)} & (\mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o \xrightarrow{\mathtt{call}(m,v)} ((m,l+1) :: \mathcal{E},mv,R,S,\mathcal{P},\mathcal{A}',k)_p \text{ if } R(m) = \lambda x.M \\
\text{(PA)} & ((m,l) :: \mathcal{E},v,R,S,\mathcal{P},\mathcal{A},k)_p \xrightarrow{\mathtt{ret}(m,v)} (\mathcal{E},l,R,S,\mathcal{P}',\mathcal{A},k)_o \\
\text{(OA)} & ((m,E) :: \mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o \xrightarrow{\mathtt{ret}(m,v)} (\mathcal{E},E[v],R,\mathcal{P},\mathcal{A}',k)_p
\end{array}$$

$$\begin{array}{ll}
(\mathbf{PC}) & m \in \mathcal{A} \text{ and } \mathcal{P}' = \mathcal{P} \cup (\mathtt{Meths}(v) \cap \mathit{dom}(R)) \\
(\mathbf{OC}) & m \in \mathcal{P} \text{ and } \mathcal{A}' = \mathcal{A} \cup (\mathtt{Meths}(v) \setminus \mathit{dom}(R))
\end{array}$$

Rules (PQ), (PA) assume condition (**PC**), and (OQ), (OA) assume (**OC**).
$\mathtt{Meths}(v)$ contains all method names appearing in $v$.

Figure 4.1: Trace (game) semantics rules for HOLi.

the semantics is given by means of traces of method calls and returns (called *moves*) between the library and its client. In between such moves, the semantics proceeds with the operational semantics of HOLi as defined in Chapter 2.

When computing the semantics of a library, the library and its methods are the *Player (P)* of the computation game, while the (intended) client is the *Opponent (O)*. As the semantics is given in absence of an actual client, $O$ actually represents every possible client for the library $P$ (also called *Proponent*). When computing the semantics of a client, the roles are reversed. In either case, the same set of rules applies and there is no need to specify whether the semantics is evaluating from the perspective of a library or that of a client. In the definitions that follow we shall see that the only difference in their semantics is the starting configurations: libraries start from an opponent configuration, whereas clients start from a proponent configuration holding their main term.

In Figure 4.1, the trace semantics uses ***game configurations***, which are divided into (proponent) $P$*-configurations* and (opponent) $O$*-configurations* given respectively as:

$$(\mathcal{E},M,R,S,\mathcal{P},\mathcal{A},k)_p \quad \text{and} \quad (\mathcal{E},l,R,S,\mathcal{P},\mathcal{A},k)_o\,.$$

In a $P$-configuration, a term $M$ is being evaluated, meaning, the library is currently in control and is internally evaluating a term. In an $O$-configuration, the environment (client) is in control and is expected to either return to a pending library call if there is one, or to make a call of its own to the library. The components $M,R,S,\mathcal{P},\mathcal{A},k,l$ are as in the operational semantics of HOLi, while $\mathcal{E}$ is an *evaluation stack*:

$$\mathcal{E} \;::=\; \varepsilon \mid (m,E) :: \mathcal{E} \mid (m,l) :: \mathcal{E}$$

which keeps track of the computations that are on hold due to external calls. The trace semantics is generated by the rules given in Figure 4.1, where label INT stands for *internal* transition; PQ for *P-question* (i.e. call) and PA for *P-answer* (i.e. return); and similarly for OQ and OA.

To potentially bound the semantics, and thus maintain a terminating analysis, we extend the counting semantics to also keep track of a newly added source of infinite execution, namely endless consecutive calls from an external component: since the trace semantics of libraries is complete, a client that keeps calling library methods—and thus does not terminate—is allowed. Given a concrete client, this *chattering* behaviour of the opponent would be bounded by the size of its term; i.e. intuitively, this sequence of calls and returns corresponds to chained method calls in the opponent term. This leads us to consider a semantics with two counters, $k$ and $l$, where $k$ keeps track of internal nesting of method calls and $l$ records the number of consecutive calls made from the external component. Note that, since $l$ counts only consecutive calls at a given call context, it is refreshed for any configuration within a nested call context. This is understood intuitively if we consider that $l$ represents the size of the largest term the opponent is allowed to hold; i.e. pushing an old $l$ into the evaluation stack and then refreshing the current working $l$ upon a change in the call context is analogous to the opponent term concretely making a method call, which pushes its term into the stack and starts evaluating a completely fresh term.

The formulation in Figure 4.1 follows closely the operational game semantics technique. For example, from a $P$-configuration $(\mathcal{E}, M, R, S, \mathcal{P}, \mathcal{A}, k)_p$, there are 3 options:

1. If $M$ can make an internal reduction, i.e. in the operational semantics in context $(R, S, k)$, then $(\mathcal{E}, M, R, S, \mathcal{P}, \mathcal{A}, k)_p$ performs this reduction (via (INT)).
2. If $M$ is stuck at a method application for a method that is not in the repository $R$, then that method must be abstract (i.e. external) and needs to be called externally. This is achieved be issuing a call move and moving to an $O$-configuration (via (PQ)). The current evaluation context and the called method name are stored, in order to resume once the call is returned (via (OA)).
3. If $M$ is a value and the evaluation stack is non-empty, then $P$ has completed a method call that was issued by $O$ (via (OQ)) and can now return (via (PA)).

On the other hand, from an $O$-configuration $(\mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o$, there are 2 options:

1. either return the last open method call (made by $P$) via (OA), or
2. call one of the public methods (from $\mathcal{P}$) using (OQ).

The role of conditions (PC) and (OC) is to ensure that each player calls the methods owned by the other, or return their own, and update the sets of public and abstract names according to the method names passed inside $v$.

**Remark 4.1.** The novelty of Figure 4.1 with respect to previous work on trace semantics for open libraries (e.g. [55]) lies in the use of $l$ in order to bound the ability of $O$ to ask repeated questions for finite analysis. The way rules (OQ) and (PA) are designed is such that any sequence of consecutive $O$-calls and $P$-returns has maximum length $2n$ if we bound $l$ to $n$ (i.e. $l \leq n$), as each such pair of moves increases $l$ by 1. On the other hand, each $P$-call supplies to $O$ a fresh counter ($l = 0$) to be used in contiguous (OQ)-(PA)'s. Thus, $l$ can be seen as keeping track of the *insistence* of $O$ in calling, which in turn depends on the size of the term $O$ holds. Another difference of this trace semantics is the absence of name-refreshing when names are passed between players. This is done for simplicity, since we remain sound at the expense of full abstraction, which is not needed for verification.

Finally, we can define the trace semantics of libraries.

**Definition 4.2** (Trace Semantics)**.** Let $L$ be a library. The semantics of $L$ is :

$$[\![L]\!] = \{(\tau, \rho) \mid (L, \emptyset, \emptyset, \emptyset, \emptyset) \xrightarrow{bld}{}^{*} (\varepsilon, R, S, \mathcal{P}, \mathcal{A}) \wedge (\varepsilon, 0, R, S, \mathcal{P}, \mathcal{A}, 0)_o \xrightarrow{\tau} \rho\}$$

where $\rho$ is produced by a trace of moves $\tau$ as defined in Figure 4.1. We additionally say that $[\![L]\!]$ *fails* if it contains some $(\tau, (\mathcal{E}, E[\mathtt{assert}(0)], \cdots))$.

**Example 4.3.** Consider the DAO example as library $L_{\mathtt{DAO}}$. Evaluating the game semantics we know the following sequence is in $[\![L_{\mathtt{DAO}}]\!]$. For economy, we hide $R, \mathcal{P}, \mathcal{A}$ and show only the top of the stack in the configurations. We also use $m(v)?$ and $m(v)!$ for calls and returns. We write $S_i$ for the store $[bal \mapsto i]$.

$$\begin{aligned}
&(\varepsilon, 0, S_{100}, 0)_o \xrightarrow{wdraw(42)?} ((wdraw, 1), wdraw(42), S_{100}, 0)_p \\
&\quad \to^{*} ((wdraw, 1), E[send(42)], S_{100}, 1)_p \xrightarrow{send(42)?} ((send, E), 2, S_{100}, 1)_o \\
&\quad \xrightarrow{wdraw(100)?} ((wdraw, 1), wdraw(100), S_{100}, 1)_p \\
&\quad \to^{*} ((wdraw, 1), E'[send(100)], S_{100}, 2)_p \xrightarrow{send(100)?} ((send, E), 2, S_{100}, 2)_o \\
&\quad \xrightarrow{send(())!} ((wdraw, 1), E'[()], S_{100}, 2)_p \to^{*} ((wdraw, 1), (), S_0, 2)_p \\
&\quad \xrightarrow{wdraw(())!} ((send, E), 1, S_0, 2)_o \xrightarrow{send(())!} ((wdraw, 1), E[()], S_0, 1)_p \\
&\quad \to^{*} ((wdraw, 1), E[\mathtt{assert}(-42 \geq 0)], S_{-42}, 1)_p
\end{aligned}$$

Taking a look at the moves played:

$$\begin{aligned}
&call\langle wdraw, 42\rangle \cdot call\langle send, 42\rangle \cdot call\langle wdraw, 100\rangle \\
&\quad \cdot call\langle send, 100\rangle \cdot ret\langle send, ()\rangle \cdot ret\langle wdraw, ()\rangle \cdot ret\langle send, ()\rangle
\end{aligned}$$

The sequence of moves is a concrete instance of the symbolic trace provided for the DAO example in Chapter 2. Here, a call is made with parameter 42, and a reentrant call with 100, which leads to the assertion violation $\mathtt{assert}(-42 \geq 0)$. Also note that a bound of $k \leq 2$ is sufficient to find this assertion violation, showing that, while shallow, errors like these can be due to intricate higher-order behaviour that is easy to miss. ◊

In the following sections we shall establish two focal properties of the trace semantics:

bounding $k$ and $l$ ensures termination (Theorem 4.7), and that it is sound and complete with respect to library errors (Theorem 4.11).

## 4.2 ML-like References

Before continuing with Theorem 4.7 and 4.11, we briefly address a remark about HOLi, which is that references are global and cannot be created or passed as values. The rationale for global higher-order references is that these suffice to code all of our examples and, moreover, allow us to prove completeness (every error has a realising client). Additionally, the precise nature of the references is not of focal relevance to the theory, so an arguably simpler but equivalent approach was taken. However, we present here a sketch of how our games can be extended with (locally created, scope extruding) ML-like references, following e.g. [47, 33]. First, the following extension to types and terms are required.

$$\theta ::= \cdots \mid \texttt{ref}\ \theta \qquad M ::= \cdots \mid\ !M \mid \texttt{ref}\ M \mid M = M \qquad v ::= \cdots \mid r$$

The term $!M$ allows dereferencing terms $M$ which evaluate to references, while $\texttt{ref}\ v$ dynamically creates a fresh name $r \in \texttt{Refs}_\theta$ (if $v : \theta$)—the semantic purpose is to update the store $S \uplus \{r \mapsto v\}$ when evaluating $\texttt{ref}\ v$. Note that this introduces another notion for how references may be of higher-order: one can store references to references, and so on. Finally, the construct $M = M$ to compare references for name equality is needed.

We then modify our games to handle reference passing and the resulting shared store. First, game configurations need to be extended with a set $\mathcal{L} \subseteq \texttt{Refs}$ that keeps track of reference names disclosed to the other player. References being passed as values means that the client can update the references belonging to the client, and viceversa. When playing a move, each reference $r$ passed is added to $\mathcal{L}$ and its value mapped to in the store. A reference (location name) may be passed in a move either within the method argument or return value (depending on whether it is a question or answer), or via the disclosed part of the store (i.e. by having references $r \in \mathcal{L}$ point at an undisclosed reference). Every newly passed reference is added into $\mathcal{L}$, and transition labels would be modified to explicitly pass the portion of the store corresponding to $\mathcal{L}$. Intuitively, this new interaction is modelled by having the player hold a local copy of the external store and telling the opponent which references have been disclosed and what their new values are (i.e. on the move label). In the other direction, the opponent directly adds their own location names to the store and tells the proponent which these are and what values these point at. Since the opponent could potentially modify all known common references, whenever the opponent passes control, all references in $\mathcal{L}$ would need to be updated with opponent values to remain complete. From the other side, the proponent manipulates all references as it would regularly do. This works because any newly introduced proponent location names would be found through a transitive closure of the disclosed names upon passing control.

With reference passing, players would manipulate references almost exactly as they already do (except for the creation of new locations), but passing and updating these would incur a heavy overhead in the moves, for which we went for the simpler and more feasible choice that sufficed.

We now proceed with Theorem 4.7 and 4.11.

## 4.3 Boundedness of Games

We prove our game semantics can be bounded, that is, games on independent components will always terminate if we bound the call counters. More precisely, Theorem 4.7 states that our game semantics is strongly normalising when call counters are bounded, meaning that every transition sequence produced from a given configuration is finite. We approach the proof in two steps: first we classify all possible transitions any given configuration can make, thus classifying all reachable configurations, and then we prove that the transition classes form a terminating sequence.

We begin by defining the classes for ordering of moves.

**Definition 4.4.** We write $|\rho| = (k_0 - k, |M|, l_0 - l)$ for the *size* of $\rho$. Where an element is not present in the configuration, we use instead the top-most occurrence of the missing component in the evaluation stack $\mathcal{E}$. i.e., opponent configurations will have size $(k_0 - k, |E|, l_0 - l)$ where $E$ is the top-most one in $\mathcal{E}$, whereas proponent configurations will have size $(k_0 - k, |M|, l_0 - l)$ where $l$ is the top-most one in $\mathcal{E}$.

**Definition 4.5.** Given configurations $\rho$ and $\rho'$, we define an order $|\rho| < |\rho'|$ to be the lexicographic ordering of the triples $(k_0 - k, |M|, l_0 - l)$, with bounds $k_0$ and $l_0$ such that $k \leq k_0$ and $l \leq l_0$.

**Lemma 4.6.** For any transition sequence $\rho_0 \to \cdots \to \rho_i \to \ldots$ and each $i > 0$, we have the following two classes of configurations:

(A) either $|\rho_i| < |\rho_{i-1}|$, or
(B) there exists $j < i - 1$ such that $|\rho_i| < |\rho_j|$

*Proof.* Let $\rho$ be a configuration. Considering all moves available to $\rho$, we have the following cases.

1. If $\rho \to \rho'$ is an (Int) move, we have two possibilities.
   (a) For a transition $(E[(\!|v|\!)], R, S, k) \to (E[v], R, S, k+1)$, where $k + 1 \leq k_0$, we have a class (B) configuration since there must be a $(E[mv], R, S, k)$ such that $(E[mv], R, S, k) \to^* (E[v], R, S, k)$ which is lexicographically ordered since $|v| < |mv|$.

   (b) Every other transition sequence is class (A) since they reduce the size of the term.

2. If $\rho \to \rho'$ is a (PQ) move, we have that $\rho'$ is a class (A) configuration since $(k, |E|, l_0) < (k, |E[mv]|, l_0 - l)$ by lexicographic ordering.
3. If $\rho \to \rho'$ is an (OA) move, we have a transition

$$((m, E) :: \mathcal{E}, l, \dots, k)_o \xrightarrow{ret(m,v)} (\mathcal{E}, E[v], \dots, k)_p$$

   which must be a result of the prior proponent question

$$(\mathcal{E}, E[mv], \dots, k)_p \xrightarrow{call(m,v)} ((m, E) :: \mathcal{E}, l_0, \dots, k)_o$$

   where $\mathcal{E}$ has an $l'$ on top. We thus have the following sequence

$$(\mathcal{E}, E[mv], \dots, k)_p \to^* (\mathcal{E}, E[v], \dots, k)_o$$

   where $(k, |E[v]|, l) < (k, |E[mv]|, l')$, so $\rho'$ is a class (B) configuration.
4. If $\rho \to \rho'$ is an (OQ) move, we have the transition

$$\begin{aligned}(\mathcal{E}, l, \dots, k)_o &\xrightarrow{call(m,v)} ((m, l+1) :: \mathcal{E}, mv, \dots, k)_p \\ &\to ((m, l+1) :: \mathcal{E}, (\!|M\{v/x\}|\!), \dots, k+1)\end{aligned}$$

   Ignoring the configuration in between, we take

$$(\mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o \xrightarrow{call(m,v)} ((m, l+1) :: \mathcal{E}, (\!|M\{v/x\}|\!), R, S, \mathcal{P}, \mathcal{A}, k+1)_p$$

   to be our new transition. We thus have that $\rho'$ is a class (A) configuration since $(k_0 - (k+1), |(\!|M\{v/x\}|\!)|, l_0 - (l+1)) < (k_0 - k, |E|, l_0 - l)$ by lexicographic ordering.
5. If $\rho \to \rho'$ is a (PA) move, we have the transition

$$((m, l) :: \mathcal{E}, v, \dots, k)_p \xrightarrow{ret(m,v)} (\mathcal{E}, l, \dots, k)_o$$

   which must be the result of a prior opponent question

$$\begin{aligned}(\mathcal{E}, l+1, \dots, k)_o &\xrightarrow{call(m,v)} ((m, l) :: \mathcal{E}, (\!|M\{v/x\}|\!), \dots, k+1)_p \\ &\to^* ((m, l) :: \mathcal{E}, (\!|v|\!), \dots, k+1)_p \\ &\to ((m, l) :: \mathcal{E}, v, \dots, k)_p \\ &\xrightarrow{ret(m,v)} (\mathcal{E}, l, \dots, k)_o\end{aligned}$$

   where $E'$ is the topmost evaluation context in $\mathcal{E}$. We thus have that $(k_0 - k, E', l_0 - l) < (k_0 - k, E', l_0 - (l+1))$, so $\rho'$ is a class (B) configuration.

□

**Theorem 4.7** (Boundedness)**.** For any game configuration $\rho$, provided an upper bound

$k_0$ and $l_0$ for call counters $k$ and $l$, the labelled transition system starting from $\rho$ is strongly normalising.

*Proof.* Let us assume there is an infinite sequence

$$\rho_0 \to \cdots \to \rho_j \to \cdots \to \rho_i \to \ldots$$

Since all reachable configurations fall into either (A) or (B) class by Lemma 4.6, we know that the sequence must comprise only (A) and (B) configurations. In this infinite sequence, we know that all sequences of (A) configurations are in descending size, so (A) sequences cannot be infinite. We also observe that (B) configurations are padded with (A) sequences and may contain nested (B) sequences. For instance, if $\rho_i$ is a (B) configuration, and $\rho_j$ is its matching configuration, there may be nested (B) configurations between $\rho_j$ and $\rho_i$, with (A) sequences padding these.

Additionally, these (B) configurations can only occur as a return to a call, so we know they only occur together with the introduction of evaluation boxes $(\!|\bullet|\!)$. Since these brackets occur in pairs and are introduced in a nested fashion, we know $\mathcal{E}$ can only contain evaluation contexts with well-bracketed evaluation boxes, meaning that there cannot be interleaved sequences of (B) configurations where their target configurations intersect. More specifically, the sequence

$$\rho_0 \to \cdots \to \rho_j \to \cdots \to \rho'_j \to \cdots \to \rho_i \to \cdots \to \rho'_i \to \ldots$$

where $\rho'_i$ matches $\rho'_j$ and $\rho_i$ matches $\rho_j$ is not possible.

Now, ignoring all (A) and nested (B) sequences, we are left with an infinite stream of top-level (B) sequences which are also in descending order. Since starting size is finite, we cannot have an infinite stream of (B) sequences. Thus, the assumption does not hold, so our semantics is strongly normalising. □

Additionally, we have a useful property of the term semantics in Lemma 4.8, which states that application preserves the proponent call counter, meaning that successfully applying and returning from a function call leaves the $k$ counter unchanged. This property of the call counter will be useful in the next section, when we prove soundness of our symbolic games.

**Lemma 4.8** ($k$-counter preserved after application)**.** Given the following sequences of game moves:

$$(1)\ (\mathcal{E}, E[M], R, S, \mathcal{P}, \mathcal{A}, k)_p \twoheadrightarrow (\mathcal{E}, E[v], R', S', \mathcal{P}', \mathcal{A}', k')_p$$
$$(2)\ ((m, E) :: \mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o \twoheadrightarrow (\mathcal{E}, E[v], R', S', \mathcal{P}', \mathcal{A}', k')_p$$

where in both (1) and (2) we apply $\twoheadrightarrow$ until we reach the first occurrence of $\mathcal{E}$ and $E[(\!|\bullet|\!)]$

in the sequence of moves, and $\twoheadrightarrow$ is the reflexive transitive closure of game transitions ($\rightarrow$), it must be the case that $k = k'$ in both (1) and (2).

*Proof.* Suppose we have the following transition sequences

$$\begin{aligned}&(1)\ (\mathcal{E}, E[M], R, S, \mathcal{P}, \mathcal{A}, k)_p \twoheadrightarrow (\mathcal{E}, E[v], R', S', \mathcal{P}', \mathcal{A}', k')_p\\ &(2)\ ((m, E) :: \mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o \twoheadrightarrow (\mathcal{E}, E[v], R', S', \mathcal{P}', \mathcal{A}', k')_p\end{aligned}$$

By induction on the length of the transition sequence (1) and mutually on the length of (2), we have the following cases, where we say $IH_p$ and $IH_o$ for the inductive hypotheses of (1) and (2) respectively:

**Base cases:**

- **Case (1):** If $M = v$, then $(\mathcal{E}, E[v], R, S, \mathcal{P}, \mathcal{A}, k)_p$ is a zero-step transition. This case holds since $k = k$.
- **Case (2):** If the opponent returns, then we have a one-step transition

$$\begin{aligned}&((m, E) :: \mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o\\ &\qquad \xrightarrow{\mathtt{ret}(m,v)} (\mathcal{E}, E[v], R', S, \mathcal{P}, \mathcal{A}', k)_p\end{aligned}$$

This case holds since $k = k$.

**Inductive cases (1):**

- if the sequence contains only internal moves, i.e. no call to the opponent is made, then we have the following transition sequence by the assumption in (1) that a value is reached.

$$(\mathcal{E}, E[M], R, S, \mathcal{P}, \mathcal{A}, k)_p \twoheadrightarrow (\mathcal{E}, E[v], R', S', \mathcal{P}', \mathcal{A}', k')_p$$

By the inductive hypothesis $IH_p$, we know that $k = k'$.
- if the sequence of internal moves gets stuck, i.e. a call to the opponent is made, then we have the following transition sequence where $m \notin dom(R')$.

$$\begin{aligned}&(\mathcal{E}, E[M], R, S, \mathcal{P}, \mathcal{A}, k)_p \twoheadrightarrow (\mathcal{E}, E[E'[mv]], R', S', \mathcal{P}', \mathcal{A}', k')_p\\ &\qquad\qquad \xrightarrow{\mathtt{call}(m,v)} ((m, E[E'[\bullet]]) :: \mathcal{E}, l, R', S', \mathcal{P}'', \mathcal{A}', k')_o\end{aligned}$$

where $\mathcal{E}$ is of the form $(m, l) :: \mathcal{E}'$. By our assumption in (1) and (2), we know that the configuration must eventually lead to a value $v$. As such, the following transition must eventually occur.

$$\begin{aligned}&((m, E[E[\bullet]]) :: \mathcal{E}, l, R', S', \mathcal{P}'', \mathcal{A}', k')_o\\ &\quad\twoheadrightarrow (\mathcal{E}, E[E[v]], R', S', \mathcal{P}', \mathcal{A}', k'')_p\end{aligned}$$

By the inductive hypothesis $IH_o$, we know that $k' = k''$. In addition, by our assumption that a value must be reached, it is the case that the following transition occurs.

$$\begin{aligned}&(\mathcal{E}, E[E[v]], R', S', \mathcal{P}', \mathcal{A}', k'')_p\\ &\quad\twoheadrightarrow (\mathcal{E}, E[v], R'', S'', \mathcal{P}'', \mathcal{A}'', k''')_p\end{aligned}$$

By the inductive hypothesis $IH_p$, we know that $k = k'''$.

**Inductive cases (2):**

- if a call to the proponent is made, then we have the following transition.

$$\begin{aligned}&(\mathcal{E}', l, R, S, \mathcal{P}, \mathcal{A}, k)_o\\ &\quad\xrightarrow{\texttt{call}(m',v)} ((m', l+1) :: \mathcal{E}', m'v, R', S, \mathcal{P}, \mathcal{A}', k)_p\end{aligned}$$

from the assumption that a value must be reached, we know that the following transition occurs.

$$\begin{aligned}&((m', l+1) :: \mathcal{E}', m'v, R', S, \mathcal{P}, \mathcal{A}', k)_p\\ &\quad\twoheadrightarrow ((m', l+1) :: \mathcal{E}', v, R'', S', \mathcal{P}', \mathcal{A}'', k')_p\end{aligned}$$

From the inductive hypothesis $IH_p$, we know that $k = k'$.

□

**Remark 4.9.** Without considering infinite-branching due to integers, complexity of our games is directly dependent on the number of paths explored for a given program. The number of paths explored grows at least exponentially up to the given bound. This is with regards to the number public methods defined and leaked, and the given bounds to which each path is explored since games are typically infinite. This sits on top of the exponential path growth of symbolic execution with respect to the The intricate interaction between $k$ and $l$, however, make an exact upper bound hard to pin down.

## 4.4 Soundness and Completeness of Games

We prove here that the trace semantics for libraries is sound and complete: for any error that can be reached in the trace semantics there is a client such that linking the library with the client reaches the same value/error, and vice versa. Intuitively, by completeness we mean that all reachable errors will be found by our games, whereas by soundness we mean that our games will only reach true errors. These two requirements are summarised in Theorem 4.11 as directions 1 to 2 (soundness) and 2 to 1 (completeness). Note that we are only interested in the library failing. As such, errors introduced by the hypothetical client are not as interesting, so we shall consider *good* clients in the proofs that follow. As before, by *error* we mean *assertion violations* (i.e. reachability of `assert(0)`). We use a bisimulation argument similar to [55] in the proofs that follow.

**Definition 4.10.** We say a configuration $\rho$ *failed* if it holds the term $E[\texttt{assert}(0)]$. We also say a configuration $\rho$ *fails* if $\rho \xrightarrow{\tau} \chi$ where $\chi$ is a configuration that failed. Similarly, we say a library $L$ fails if $[\![L]\!]$ fails (as per Definition 4.2).

**Theorem 4.11** (Soundness and Completeness)**.** We call a client *good* if it contains no assertions. For any library $L$, the following are equivalent:

1. $[\![L]\!]$ fails (reaches an assertion violation)
2. there exists a good client $C$ such that $[\![L;C]\!]$ fails

*Proof.* 1 to 2: Suppose now that $(\tau, \rho) \in [\![L]\!]$ for some trace $\tau$ and failed $\rho$. By Theorem 4.13, we have that there is a good client $C$ realising the trace $\tau$. So then, by Lemma 4.12, we have that $[\![L;C]\!]$ fails.

2 to 1: Suppose $[\![L;C]\!]$ fails for some good client $C$. Then, by Lemma 4.12, there are $\tau, \rho, \rho'$ such that $(\tau, \rho) \in [\![L]\!]$, $(\tau, \rho') \in [\![C]\!]$, and $\rho$ failed. □

The latter relies on an auxiliary lemma (well-composing of libraries and clients), and a definability result akin to game semantics definability arguments.

**Lemma 4.12** (L-C Compositionality)**.** For any library $L$ and compatible good client $C$, $[\![L;C]\!]$ fails if and only if there exist $(\tau_1, \rho_1) \in [\![L]\!]$ and $(\tau_2, \rho_2) \in [\![C]\!]$ such that $\tau_1 = \tau_2$ and $\rho_1 = (\mathcal{E}, E[\texttt{assert}(0)], \cdots)$.

**Theorem 4.13** (Definability)**.** Let $L$ be a library and $(\tau, \rho) \in [\![L]\!]$. There is a good client compatible with $L$ such that $(\tau, \rho') \in [\![C]\!]$ for some $\rho'$.

These auxiliary results are proven in the following sections.

### 4.4.1 Semantic Composition

We start by defining a notion of composition that combines the traces produced by two configurations. These are supposed to correspond to a library and a client, but for now we will only require that the configurations satisfy a set of compatibility conditions.

We say configurations $\rho$ and $\rho'$ of *opposite polarity* (one is $p$ if the other is $o$) are ***compatible*** ($\rho \asymp \rho'$) if:

- their stores are disjoint: $\texttt{Refs}(\rho) \cap \texttt{Refs}(\rho') = \varnothing$
- $\rho$ closes and is closed by $\rho'$: $\mathcal{P} = \mathcal{A}'$ and $\mathcal{P}' = \mathcal{A}$
- undisclosed names of $\rho$ do not occur in $\rho'$ and vice versa: $(\texttt{Meths}(\rho) \setminus (\mathcal{A} \cup \mathcal{P})) \cap \texttt{Meths}(\rho') = \varnothing$
- their evaluation stacks are compatible, written $\mathcal{E} \asymp \mathcal{E}'$, which means:
    - $\mathcal{E} = \mathcal{E}' = \varepsilon$; or
    - $\mathcal{E} = (m, l) :: \mathcal{E}_1$ and $\mathcal{E}' = (m, E) :: \mathcal{E}_1'$, and $\mathcal{E}_1 \asymp \mathcal{E}_1'$; or
    - $\mathcal{E} = (m, E) :: \mathcal{E}_1$ and $\mathcal{E}' = (m, l) :: \mathcal{E}_1'$, and $\mathcal{E}_1 \asymp \mathcal{E}_1'$.

Note that compatibility of evaluation stacks expects that compatible configurations are always of opposite polarity. This reflects the fact that we compose libraries with closing clients.

With these definitions, we follow by defining different notions of composition.

**Definition 4.14.** Let $\rho_1, \rho_2, \rho_1', \rho_2'$ be game configurations. The following rules define the semantic composition of two configurations.

$$\frac{\rho_1 \rightarrow' \rho_1' \qquad \rho_2' = \rho_2}{\rho_1 \oslash \rho_2 \rightarrow' \rho_1' \oslash \rho_2'}\ \textsc{Int}_L \qquad \frac{\rho_2 \rightarrow' \rho_2' \qquad \rho_1' = \rho_1}{\rho_1 \oslash \rho_2 \rightarrow' \rho_1' \oslash \rho_2'}\ \textsc{Int}_C$$

$$\frac{\rho_1 \xrightarrow{\texttt{call}(m,v)}{}' \rho_1' \qquad \rho_2 \xrightarrow{\texttt{call}(m,v)}{}' \rho_2'}{\rho_1 \oslash \rho_2 \rightarrow' \rho_1' \oslash \rho_2'}\ \textsc{Call}$$

$$\frac{\rho_1 \xrightarrow{\texttt{ret}(m,v)}{}' \rho_1' \qquad \rho_2 \xrightarrow{\texttt{ret}(m,v)}{}' \rho_2'}{\rho_1 \oslash \rho_2 \rightarrow' \rho_1' \oslash \rho_2'}\ \textsc{Ret}$$

**Definition 4.15.** Given a library $L$ and a compatible client $C$, we call $[\![L]\!] \oslash [\![C]\!]$ the semantic composition of $L$ and $C$.

### 4.4.2 Composite Semantics and Internal Composition

We now introduce the notion of composing game configurations ***internally***, which occurs when merging two compatible game configurations into a single composite semantics configuration. We first refine the operational semantics and produce a ***composite semantics***. This is necessary for our compositionality argument since there is an asymmetry between the call counters of the opponent and proponent configurations. Proponent configurations count calls internally while opponent configurations have no internal counters, and thus only count calls when playing moves. This requires that we keep track of two pairs of counters, one for each component, which may change at different rates.

With this in mind, to define the composite semantics, we extend the term configurations to obtain tuples of the following form:

$$(M, R_1, R_2, S, k_1, k_2, l_1, l_2) \text{ for which we shall write } (M, \vec{R}, S, \vec{k}, \vec{l})$$

where $R_1$ and $R_2$ are the library and client methods respectively, such that $dom(R_1) \cap dom(R_2) = \varnothing$, $S$ is the combined store, and $k_1, l_1$ and $k_2, l_2$ are counters managed by the library and client. All operators tagged with $i$ will be operating on the $i$th component; e.g. $\vec{R}[m \mapsto M]_i$ states that $R_i[m \mapsto M]$ in $\vec{R}$. We also extend $M$ by tagging all method names (written $m^i$) as well as all lambda-abstractions (written $\lambda^i$) with $i \in \{1, 2\}$ to show whether they are being called from the library (1) or the client (2). We write $M^i$ to be the term $M$ with all its methods and lambdas tagged with $i$. Evaluation contexts are also extended to mark methods which are being called from the opposite polarity:

$$E ::= \cdots \mid (\!|E|\!)^i \mid (\!|E|\!)^{\langle i,l \rangle}$$

Intuitively, $i$ is the component that is currently at a proponent configuration in the equivalent game semantics, while $l$ in $(\!|E|\!)^{\langle i,l \rangle}$ is the opponent counter for component $3-i$. This will be used particularly when evaluating a method call $m^i v$ when $m \notin dom(R_i)$. Applying these changes, we define the semantics for composite terms ($\rightarrow_{1,2}$).

$$(E[\texttt{assert}(i)], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[()], \vec{R}, S, \vec{k}, \vec{l}) \quad (i \neq 0)$$

$$(E[!r], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[S(r)], \vec{R}, S, \vec{k}, \vec{l})$$

$$(E[r := v], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[()], \vec{R}, S[r \mapsto v], \vec{k}, \vec{l})$$

$$(E[\pi_j \langle v_1, v_2 \rangle], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[v_j], \vec{R}, S, \vec{k}, \vec{l})$$

$$(E[i_1 \oplus i_2], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[i], \vec{R}, S, \vec{k}, \vec{l}) \quad (i = i_1 \oplus i_2)$$

$$(E[\lambda^i x.M], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[m], \vec{R}[m \mapsto \lambda x.M]_i, S, \vec{k}, \vec{l}) \quad (m \notin dom(\vec{R}))$$

$$(E[\texttt{if } i \texttt{ then } M_1 \texttt{ else } M_0], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[M_j], \vec{R}, S, \vec{k}, \vec{l}) \quad (j = 1 \text{ iff } i \neq 0)$$

$$(E[\texttt{let } x = v \texttt{ in } M], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[M\{v/x\}], \vec{R}, S, \vec{k}, \vec{l})$$

$$(E[\texttt{letrec } f = \lambda^i x.M \texttt{ in } M'], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[M'\{m/f\}], \vec{R}[m \mapsto \lambda x.M\{m/f\}]_i, S, \vec{k}, \vec{l})$$

$$(E[m^i v], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[(\!|M\{v/y\}^i|\!)^i], \vec{R}, S, \vec{k} +_i 1, \vec{l}) \quad (R_i(m) = \lambda y.M)$$

$$(E[m^i v], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[(\!|m^{3-i} v|\!)^{\langle i, l+_{3-i}1 \rangle}], \vec{R}, S, \vec{k}, \vec{l}[l_i \mapsto 0]) \quad (R_{3-i}(m) = \lambda y.M)$$

$$(E[(\!|v|\!)^i], \vec{R}, S, \vec{k} +_i 1, \vec{l}) \rightarrow_{1,2} (E[v^i], \vec{R}, S, \vec{k}, \vec{l})$$

$(E[\llparenthesis v \rrparenthesis^{\langle i,l\rangle}], \vec{R}, S, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[v^i], \vec{R}, S, \vec{k}, \vec{l}[l_{3-i} \mapsto l, l_i \mapsto \mathit{last}(E)])$ if $\mathit{last}(E)$ is defined and $\mathit{last}(E) = \hat{l}$ if $E = E_1[\llparenthesis E_2 \rrparenthesis^{\langle j,\hat{l}\rangle}]$ provided $E_2$ has no tags $\llparenthesis \bullet \rrparenthesis^{\langle j',\hat{l}'\rangle}$

**Definition 4.16.** For a library $L$ and compatible client $C$, their composite semantics is:

$$\llbracket L \rrbracket \curlywedge \llbracket C \rrbracket = \{\rho \mid \rho_0 \curlywedge \rho_0' \rightarrow_{1,2}^{*} \rho\}$$

where $\rho_0 \in \llbracket L \rrbracket$ and $\rho_0' \in \llbracket C \rrbracket$ are the respective starting configurations, and $\rightarrow_{1,2}^{*}$ is the reflexive transitive closure of $\rightarrow_{1,2}$.

We continue by defining the ***internal composition*** of compatible configurations $\rho_1 \asymp \rho_2$. We define the internal composition $\rho_1 \curlywedge \rho_2$ to be a configuration in our new composite semantics by pattern matching on the configuration polarity and evaluation stacks according to the following rules. For clarity, we annotate opponent and proponent configurations with $o$ and $p$ respectively.

**Definition 4.17.** Given compatible configurations $\rho_1 \asymp \rho_2$, the internal composition of two configurations is defined as follows:

**Initial Configurations:**

$$\begin{aligned}
\rho_1 &= ([], -, R_1, S_1, \mathcal{P}_1, \mathcal{A}_1, 0, 0)_o \\
\rho_2 &= ([], M_0, R_2, S_2, \mathcal{P}_2, \mathcal{A}_2, 0, -)_p \\
\rho_1 \curlywedge \rho_2 &= (\llparenthesis \bullet \rrparenthesis^{\langle 1,0\rangle}[M_0^2], R_1, R_2, S_1 \uplus S_2, 0, 0, 0, 0)
\end{aligned}$$

**Interim Configurations (OP):**

$$\begin{aligned}
\rho_1 &= (\mathcal{E}_1, -, R_1, S_1, \mathcal{P}_1, \mathcal{A}_1, k_1, l_1)_o \\
\rho_2 &= (\mathcal{E}_2, M, R_2, S_2, \mathcal{P}_2, \mathcal{A}_2, k_2, -)_p \\
\mathcal{E}_1 &= (m, E) :: \mathcal{E}_1' \quad \mathcal{E}_2 = (m, l_2) :: \mathcal{E}_2' \\
\rho_1 \curlywedge \rho_2 &= ((\mathcal{E}_1' \curlywedge \mathcal{E}_2')[E^1[\llparenthesis M^2 \rrparenthesis^{\langle 1,l_2\rangle}]], R_1, R_2, S_1 \uplus S_2, k_1, k_2, l_1, l_2)
\end{aligned}$$

**Interim Configurations (PO):**

$$\begin{aligned}
\rho_1 &= (\mathcal{E}_1, M, R_1, S_1, \mathcal{P}_1, \mathcal{A}_1, k_1, -)_p \\
\rho_2 &= (\mathcal{E}_2, -, R_2, S_2, \mathcal{P}_2, \mathcal{A}_2, k_2, l_2)_o \\
\mathcal{E}_1 &= (m, l_1) :: \mathcal{E}_1' \quad \mathcal{E}_2 = (m, E) :: \mathcal{E}_2' \\
\rho_1 \curlywedge \rho_2 &= ((\mathcal{E}_1' \curlywedge \mathcal{E}_2')[E^2[\llparenthesis M^1 \rrparenthesis^{\langle 2,l_1\rangle}]], R_1, R_2, S_1 \uplus S_2, k_1, k_2, l_1, l_2)
\end{aligned}$$

where $\mathcal{E}_1' \curlywedge \mathcal{E}_2'$ is a single evaluation context resulting from the composition of compatible stacks $\mathcal{E}_1'$ and $\mathcal{E}_2'$, which we define as follows:

$$\varepsilon \curlywedge \varepsilon = \bullet$$
$$((m', E) :: \mathcal{E}_1'') \curlywedge ((m', l) :: \mathcal{E}_2'') = (\mathcal{E}_1'' \curlywedge \mathcal{E}_2'')[E^1[\llparenthesis \bullet \rrparenthesis^{\langle 1,l\rangle}]]$$

$$((m', l) :: \mathcal{E}_1'') \curlywedge ((m', E) :: \mathcal{E}_2'') = (\mathcal{E}_1'' \curlywedge \mathcal{E}_2'')[E^2[\llparenthesis \bullet \rrparenthesis^{\langle 2,l \rangle}]]$$

Interim rules are divided into OP and PO cases for opponent-proponent and proponent-opponent composition. Notice that there is only one case for initial configurations since the game must start from an opponent-proponent configuration where both stacks are empty. Additionally, there are no PP and OO cases for interim configurations: proponent-proponent configurations are not reachable; and reachability of opponent-opponent configurations is only necessary where full compositionality (compositionality with arbitrary arity) is needed, which we do not need in our library-client paradigm.

**Definition 4.18.** Given a library $L$ and compatible client $C$, we call $[\![L]\!] \curlywedge [\![C]\!]$ the internal composition of $L$ and $C$.

### 4.4.3 Bisimilarity of Semantic and Internal Composition

We begin by defining bisimilarity for the semantic and internal composition.

**Definition 4.19.** A set $\mathcal{R}$ with elements of the form $(\rho_1, \rho_2)$, where $\rho_1$ is a configuration of the form $\rho_1' \oslash \rho_1''$ and $\rho_2$ is from the composite semantics, is a ***bisimulation*** if for all $(\rho_1, \rho_2) \in R$:

- if $\rho_1 \to' \rho_1'$ then $\rho_2 \to_{1,2}^* \rho_2'$ and $(\rho_1', \rho_2') \in \mathcal{R}$;
- if $\rho_2 \to_{1,2} \rho_2'$ then $\rho_1 \to'^* \rho_1'$ and $(\rho_1', \rho_2') \in \mathcal{R}$.

**Definition 4.20.** We say that two game configurations $\rho, \rho'$ are *bisimilar*, and write $\rho \sim \rho'$, if there is a bisimulation $\mathcal{R}$ such that $\rho \mathcal{R} \rho'$.

Lemma 4.21 states that, given game configurations, it is possible to obtain the composite semantics ($\to_{1,2}$) from the semantic composition of the corresponding compatible configurations, and vice versa.

**Lemma 4.21.** Given game configurations $\rho \asymp \rho'$, it is the case that $(\rho \oslash \rho') \sim (\rho \curlywedge \rho')$.

*Proof.* We want to show that $\mathcal{R} = \{(\rho_1 \oslash \rho_2, \rho_1 \curlywedge \rho_2) \mid \rho_1 \asymp \rho_2\}$ is a bisimulation. Suppose $(\rho_1 \oslash \rho_2, \rho_1 \curlywedge \rho_2) \in \mathcal{R}$. We begin with case analysis on the transitions available to the semantic composite. If $(\rho_1 \oslash \rho_2) \to' (\rho_1' \oslash \rho_2')$, then $\rho_1' \asymp \rho_2'$. Now, by cases of the transitions, we prove that composite semantics can be obtained from the semantic composition.

1. If $(\rho_1 \oslash \rho_2) \to' (\rho_1' \oslash \rho_2')$ is an $(\textsc{Int}_L)$ move, then we have internal moves in the execution of $\rho_1$ up to $\rho_1'$. Since the composite semantics is concrete and, by construction, equivalent to operational semantics when no methods of opposite polarity are called, we can see that $(\rho_1 \curlywedge \rho_2) \to_{1,2} (\rho_1' \curlywedge \rho_2)$.

2. If $(\rho_1 \oslash \rho_2) \rightarrow' (\rho_1' \oslash \rho_2')$ is a (Call) move, then we have that $\rho_1 \xrightarrow{\texttt{call}(m,v)}{}' \rho_1'$ and $\rho_2 \xrightarrow{\texttt{call}(m,v)}{}' \rho_2'$. We thus have two cases: (1) $m$ is defined in $R_1$ and (2) it is in $R_2$. In case (1), we have the following semantics for $\rho_1$ and $\rho_2$ where the evaluation stacks are not equal:

$$\begin{aligned}
&((m', E') :: \mathcal{E}_1, -, R_1, S_1, \mathcal{P}_1, \mathcal{A}_1, k_1, l_1)_o \\
&\quad \xrightarrow{\texttt{call}(m,v)}{}' ((m, l_1 + 1) :: (m', E') :: \mathcal{E}_1, mv, R_1, S_1, \mathcal{P}_1, \mathcal{A}_1', k_1, -)_p \\
&((m', l_2) :: \mathcal{E}_2, E[mv], R_2, S_2, \mathcal{P}_2, \mathcal{A}_2, k_2, -)_p \\
&\quad \xrightarrow{\texttt{call}(m,v)}{}' ((m, E) :: (m', l_2) :: \mathcal{E}_2, -, R_2, S_2, \mathcal{P}_2', \mathcal{A}_2, k_2, l_0)_o
\end{aligned}$$

We thus have:

$$\begin{aligned}
\rho_1 \Yup \rho_2 &= ((\mathcal{E}_1 \Yup \mathcal{E}_2)[E'^1[\llparenthesis E^2[m^2 v]\rrparenthesis^{\langle 1, l_2 \rangle}]], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l}) \\
\rho_1' \Yup \rho_2' &= ((\mathcal{E}_1 \Yup \mathcal{E}_2)[E'^1[\llparenthesis E^2[\llparenthesis m^1 v\rrparenthesis^{\langle 2, l_1+1 \rangle}]\rrparenthesis^{\langle 1, l_2 \rangle}]], \\
&\qquad \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l}[l_2 \mapsto 0] +_1 1)
\end{aligned}$$

From the composite semantics evaluating $\rho_1 \Yup \rho_2$ we have:

$$\begin{gathered}
((\mathcal{E}_1 \Yup \mathcal{E}_2)[E'^1[\llparenthesis E^2[m^2 v]\rrparenthesis^{\langle 1, l_2 \rangle}]], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l}) \\
\rightarrow_{1,2} ((\mathcal{E}_1 \Yup \mathcal{E}_2)[E'^1[\llparenthesis E^2[\llparenthesis m^1 \hat{v}\rrparenthesis^{\langle 2, l_1+1 \rangle}]\rrparenthesis^{\langle 1, l_2 \rangle}]], \\
\vec{R}, S_1 \cup S_2, \vec{k}, \vec{l}[l_2 \mapsto 0] +_1 1)
\end{gathered}$$

Since $v = \hat{v}$ by determinism of the operational semantics, we have that $(\rho_1 \Yup \rho_2) \rightarrow_{1,2} (\rho_1' \Yup \rho_2')$. In addition, we can observe that the case for equal evaluation stacks is proven by substituting the initial stacks with equal ones, which results in an empty evaluation context. Similarly, the dual case (2), where $m$ is defined in $R_1$, is identical but with polarities swapped–i.e. shown by the polar complement of $(\rho_1 \Yup \rho_2) \rightarrow_{1,2} (\rho_1' \Yup \rho_2')$.

3. If $(\rho_1 \oslash \rho_2) \rightarrow' (\rho_1' \oslash \rho_2')$ is a (Ret) move, then we have that $\rho_1 \xrightarrow{\texttt{ret}(m,v)}{}' \rho_1'$ and $\rho_2 \xrightarrow{\texttt{ret}(m,v)}{}' \rho_2'$. As with the Call case, if $m \in dom(R_2)$ and stacks are not equal, we have:

$$\begin{aligned}
&((m, E) :: \mathcal{E}_1, -, R_1, S_1, \mathcal{P}_1, \mathcal{A}_1, k_1, l_1)_o \\
&\quad \xrightarrow{\texttt{ret}(m,v)}{}' (\mathcal{E}_1, E[v], R_1, S_1, \mathcal{P}_1, \mathcal{A}_1', k_1, -)_p \\
&((m, l_2) :: \mathcal{E}_2, v, R_2, S_2, \mathcal{P}_2, \mathcal{A}_2, k_2, -)_p \\
&\quad \xrightarrow{\texttt{ret}(m,v)}{}' (\mathcal{E}_2, -, R_2, S_2, \mathcal{P}_2', \mathcal{A}_2, k_2, l_2)_o
\end{aligned}$$

Here, we have two cases: $\mathcal{E}_1 = \mathcal{E}_2$, and otherwise. We start with the case where $\mathcal{E}_1 \neq \mathcal{E}_2$, since the opposite case is a simpler version of it. Again, we have the following composite configurations:

$$\begin{aligned}\rho_1 \curlywedge \rho_2 &= ((\mathcal{E}_1 \curlywedge \mathcal{E}_2)[E^1[\llparenthesis v^2 \rrparenthesis^{\langle 1,l_2\rangle}]], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l})\\ \rho_1' \curlywedge \rho_2' &= ((\mathcal{E}_1' \curlywedge \mathcal{E}_2')[E'^2[\llparenthesis E^1[v^1] \rrparenthesis^{\langle 2,l_1'\rangle}]],\\ &\qquad\qquad \vec{R}, S_1 \cup S_2, \vec{k}, l_1', l_2)\end{aligned}$$

where $\mathcal{E}_1 = (m', l_1') :: \mathcal{E}_1'$ and $\mathcal{E}_2 = (m', E') :: \mathcal{E}_2$.
Now, from the composite semantics, we have:

$$\begin{aligned}&((\mathcal{E}_1 \curlywedge \mathcal{E}_2)[E^1[\llparenthesis v^2 \rrparenthesis^{\langle 1,l_2\rangle}]], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l})\\ &\to_{1,2} ((\mathcal{E}_1 \curlywedge \mathcal{E}_2)[E^1[\hat{v}^1]], \vec{R}, S_1 \cup S_2, \vec{k}, \mathit{last}((\mathcal{E}_1 \curlywedge \mathcal{E}_2)[E^1[\bullet]]), l_2)\\ &= ((\mathcal{E}_1' \curlywedge \mathcal{E}_2')[E'^2[\llparenthesis E^1[\hat{v}^1] \rrparenthesis^{\langle 2,l_1'\rangle}]], \vec{R}, S_1 \cup S_2, \vec{k}, l_1', l_2)\end{aligned}$$

We can observe that $\mathit{last}(E) = l_1'$ since $E$ comes directly from the evaluation stack and is, thus, untagged, and the top-most counter is $l_1'$ since

$$(\mathcal{E}_1' \curlywedge \mathcal{E}_2')[E'^2[\llparenthesis E^1[\bullet] \rrparenthesis^{\langle 2,l_1'\rangle}]] = (\mathcal{E}_1 \curlywedge \mathcal{E}_2)[E^1[\bullet]]$$

Finally, we have that $k_2 = k_2'$ when returning a value since, from Lemma 4.8, $k$ must always decrease back to its original value after evaluating a method call.
We thus have $(\rho_1 \curlywedge \rho_2) \to_{1,2} (\rho_1' \curlywedge \rho_2')$. As previously, the case for empty stacks is a simpler version of this, while the dual case (2) is the polar complement of the configurations.

Having shown that external composition produces composite semantics transitions, we continue with the other direction of the argument, which aims to show that the external composition can be produced from composite semantics transitions. We now derive the corresponding semantic compositions by case analysis on the composite semantics rules.

1. If we have an untagged transition, or one where the redex involves no names of opposite polarity being called, then we have an exact correspondence with internal moves, since the composite semantics are identical to the operational semantics on closed terms.
2. If the transition involves a method called from an opposite polarity, we have a transition of the form

$$(E[m^i v], \ldots, \vec{l}) \to_{1,2} (E[\llparenthesis m^{3-i} v \rrparenthesis^{\langle i, l_{3-i}+1\rangle}], \ldots, \vec{l}[l_i \mapsto 0] +_{3-i} 1)$$

which corresponds to evaluating the semantics on an initial configuration $\rho_1 \curlywedge \rho_2$ with the following cases:

(a) for an OP configuration, we have the following:

$$\rho_1 = (\mathcal{E}_1, -, R_1, S_1, k_1, l_1)_o$$

$$\rho_2 = (\mathcal{E}_2, E[mv], R_2, S_2, k_2, -)_p$$

where $\mathcal{E}_1 = (m', E') :: \mathcal{E}'_1$ and $\mathcal{E}_2 = (m', l_2) :: \mathcal{E}'_2$. Let us set $E[m^i v] = (\mathcal{E}'_1 \curlywedge \mathcal{E}'_2)[E'^1[\llparenthesis M^2 \rrparenthesis^{\langle 1,l_2 \rangle}]]$ and $M^2 = E''[m^i v]$, where $m \notin R_2$, $i = 2$, and $E''$ is untagged. We therefore have:

$$\begin{aligned}&((\mathcal{E}'_1 \curlywedge \mathcal{E}'_2)[E'^1[\llparenthesis M^2 \rrparenthesis^{\langle 1,l_2 \rangle}]], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l})\\ &\to_{1,2} (E[\llparenthesis m^1 v \rrparenthesis^{\langle 2,l_1+1 \rangle}], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l}[l_2 \mapsto 0] +_1 1)\end{aligned}$$

We now want to show that semantically composing the configurations results in an equivalent transition $\rho_1 \oslash \rho_2 \to' \rho'_1 \oslash \rho'_2$. Since this is a Call move, we know that $\rho_1 \xrightarrow{\mathtt{call}(m,v)}{}' \rho'_1$ and $\rho_2 \xrightarrow{\mathtt{call}(m,v)}{}' \rho'_2$. Evaluating those transitions, we have that

$$\rho'_1 = ((m, l_1 + 1) :: \mathcal{E}_1, mv, \ldots, k_1, -)_o$$

$$\rho'_2 = ((m, E'') :: \mathcal{E}_2, -, \ldots, k_2, 0)_p$$

which, when syntactically composed, form the configuration

$$((\mathcal{E}_1 \curlywedge \mathcal{E}_2)[E''^2[\llparenthesis (mv)^1 \rrparenthesis^{\langle 2,l_1+1 \rangle}]], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l}[l_2 \mapsto 0] +_1 1)$$

We can observe that the resulting configurations are equivalent since $E'' = E''^2$, which follows from $E''[m^i v] = M^2$. Additionally, since

$$(\mathcal{E}'_1 \curlywedge \mathcal{E}'_2)[E'^1[\llparenthesis E''^2[\bullet] \rrparenthesis^{\langle 1,l_2 \rangle}]] = (\mathcal{E}_1 \curlywedge \mathcal{E}_2)[E''^2[\llparenthesis \bullet \rrparenthesis^{\langle 2,l_1+1 \rangle}]]$$

it suffices to show $(mv)^1 = m^1 v$, particularly that $v = v^1$. Now, since the composite semantics ensures that $v$ will be tagged with 1 when called from a method $m^1$, as it reduces to $M\{v/y\}^1$, we have that $v = v^1$, meaning that the transitions are equal.

(b) for a PO configuration, the polar complement of case (a) suffices.

(c) for an initial configuration OP, we have a simpler version of case (a) where the evaluation stacks are equal, resulting in an empty evaluation context $\mathcal{E}'_1 \curlywedge \mathcal{E}'_2 = \bullet$.

3. If the transition involves a tagged value and is of the form

$$\begin{aligned}&(E[\llparenthesis v \rrparenthesis^{\langle i,l \rangle}], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l})\\ &\to_{1,2} (E[v^i], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l}[l_{3-i} \mapsto l, l_i \mapsto \mathit{last}(E)])\end{aligned}$$

then we want to show an equivalence to a Ret move in the semantic composite. As with case (2), we start by defining this transition as the syntactic composite transition $(\rho_1 \curlywedge \rho_2) \to_{1,2} (\rho'_1 \curlywedge \rho'_2)$. Then, by case analysis on $\rho_1 \curlywedge \rho_2$:

(a) for an OP configuration, we have the following:

$$\rho_1 = (\mathcal{E}_1, -, R_1, S_1, k_1, l_1)_o$$

$$\rho_2 = (\mathcal{E}_2, v, R_2, S_2, k_2, -)_p$$

where $\mathcal{E}_1 = (m, E') :: \mathcal{E}_1'$ and $\mathcal{E}_2 = (m, l_2) :: \mathcal{E}_2'$. Let $E[v] = (\mathcal{E}_1' \curlywedge \mathcal{E}_2')[E'^1[\llparenthesis v^2 \rrparenthesis^{\langle 1, l_2 \rangle}]]$. We thus have:

$$(E[\llparenthesis v^2 \rrparenthesis^{\langle 1, l_2 \rangle}], \vec{R}, S_1 \cup S_2, \vec{k}, \vec{l}) \rightarrow_{1,2} (E[v^1], \vec{R}, S_1 \cup S_2, \vec{k}, last(E), l_2)$$

We then show that semantic composition produces an equivalent transition $\rho_1 \oslash \rho_2 \rightarrow' \rho_1' \oslash \rho_2'$. Given we have a Ret move, we know that $\rho_1 \xrightarrow{\mathtt{ret}(m,v)}{}' \rho_1'$ and $\rho_2 \xrightarrow{\mathtt{ret}(m,v)}{}' \rho_2'$, such that:

$$\rho_1' = (\mathcal{E}_1', E'[v], \ldots, k_1, -)_p$$

$$\rho_2' = (\mathcal{E}_2', -, \ldots, k_2, l_2)_o$$

where $\mathcal{E}_1' = (m', l_1') :: \mathcal{E}_1''$ and $\mathcal{E}_2' = (m', E') :: \mathcal{E}_2''$. Internally composing these resulting configurations, we have:

$$((\mathcal{E}_1'' \curlywedge \mathcal{E}_2'')[E''[\llparenthesis E'^1[v^1] \rrparenthesis^{\langle 2, l_1' \rangle}]], \vec{R}', S_1 \cup S_2, \vec{k}, l_1', l_2)$$

Since $(\mathcal{E}_1'' \curlywedge \mathcal{E}_2'')[E''[\llparenthesis \bullet \rrparenthesis^{\langle 2, l_1' \rangle}]] = (\mathcal{E}_1' \curlywedge \mathcal{E}_2')[\bullet]$, we have that $(\mathcal{E}_1' \curlywedge \mathcal{E}_2')[E'^1[v^1]]$, from which we have $(\mathcal{E}_1'' \curlywedge \mathcal{E}_2'')[E''[\llparenthesis E'^1[v^1] \rrparenthesis^{\langle 2, l_1' \rangle}]] = E[v^1]$, and that $last(E) = l_1'$ since $E_1'$ is untagged. Thus, the transition produces the composition.

(b) for a PO configuration, we have the polar complement of (a) as previously.

(c) for an initial OP configuration, we again have a simplification of (a), where the evaluation stacks are equal and the resulting evaluation context is empty.

With this, we are done showing the equivalence of transitions. Lastly, we can observe that $\rho$ is final iff $\rho'$ is final since they are both leaf nodes generated by equivalent terminal rules. We therefore have $(\rho \oslash \rho') \sim (\rho \curlywedge \rho')$. □

### 4.4.4 Library-Client Compositionality

We can now prove compositionality of the modified trace semantics. For this, we want to show that syntactic composition (Def. 2.3) can be obtained from the semantic counterpart (Def. 4.15) and vice versa. Since we have bisimilarity between semantic and internal composition, the goal here is to show that internal composition (Def. 4.18) is related to syntactic composition under some notion of equivalence.

We shall present in Lemma 4.12 our notion of equivalence between internal and syntactic composition in the form of reachability of errors. Intuitively, the goal is to show that any error reachable in the game semantics of a library $L$ will be reachable in the semantics of the syntactic composite $L;C$ for some client $C$ that is able to trigger the error in $L$ and does not introduce any errors of its own, and vice-versa. The decision to have $C$ not incur errors of its own is because it suffices to know whether the library code checked is able to fail, since errors caused by a hypothetical client occur trivially often.

***Lemma 4.12*** For any library $L$ and compatible good client $C$, $[\![L\boldsymbol{;}C]\!]$ fails if and only if there exist $(\tau_1, \rho_1) \in [\![L]\!]$ and $(\tau_2, \rho_2) \in [\![C]\!]$ such that $\tau_1 = \tau_2$ and $\rho_1 = (\mathcal{E}, E[\texttt{assert}(0)], \cdots)$.

*Proof.* We have a case for each direction.

$(1 \implies 2)$**:**

1. Consider $L\boldsymbol{;}C$ that reaches $\chi$.
2. By inspection of the composite semantics, we have that $[\![L]\!] \curlywedge [\![C]\!]$ reaches $\chi$.
3. By bisimilarity (Lemma 4.21) we have that $[\![L]\!] \oslash [\![C]\!]$ reaches $\chi$.
4. By definition of semantic composition, we know there are traces $\tau \in [\![L]\!]$ and $\tau \in [\![C]\!]$ such that $[\![L]\!] \xrightarrow{\tau}\!\!\!\!\!\rightarrow' \chi$.

$(2 \implies 1)$**:**

1. Consider traces $\tau \in [\![L]\!]$ and $\tau \in [\![C]\!]$ such that $[\![L]\!] \xrightarrow{\tau}\!\!\!\!\!\rightarrow' \chi$.
2. By definition of semantic composition we have that $[\![L]\!] \oslash [\![C]\!]$ reaches $\chi$.
3. By bisimilarity (Lemma 4.21) we have that $[\![L]\!] \curlywedge [\![C]\!]$ reaches $\chi$.
4. By inspection of the composite semantics, we know $L\check{;}C$ reaches $\chi$.

□

### 4.4.5 Definability

In this section we show that every trace $\tau$ in the semantics of a library $L$ has a corresponding good client that realises the same trace in its semantics.

Let $L$ be a library with public names $\mathcal{P}$ and abstract names $\mathcal{A}$. Given a trace $\tau$ produced by $L$, with $\mathcal{P}'$ and $\mathcal{A}'$ respectively the public and abstract names introduced in $\tau$, we set:

$$\begin{aligned}
\mathcal{N} &= \mathcal{P} \cup \mathcal{P}' \cup \mathcal{A} \cup \mathcal{A}' \\
\Theta_v &= \{\theta \mid \exists m \in \mathcal{N}.\ m : \theta' \wedge \theta \text{ a syntactic subtype of } \theta'\} \\
\Theta_m &= \{\theta \in \Theta \mid \theta \text{ a method type}\}
\end{aligned}$$

Note that the above sets are finite, since $\tau, \mathcal{P}, \mathcal{A}$ are finite. We assume a fixed enumeration of $\mathcal{N} = \{m_1, m_2, \cdots, m_n\}$. Moreover, for each type $\theta$, we let $\mathbf{defval}_\theta$ be a default value,

```
global cnt := 0
global meth := 0
global ref$_i$ := $m_i$              # for each $m_i \in \mathcal{P}$
global ref$_i$ := defval         # for each $m_i \in \mathcal{P}'$
global val$_\theta$ := defval         # for each $\theta \in \Theta_v$

public $m_i$ = $\lambda$x.                # for each $m_i \in \mathcal{A}$
  cnt++; meth:=i; val$_{\theta_1}$:=x; oracle()

$m_i$ = $\lambda$x.                       # for each $m_i \in \mathcal{A}'$
  cnt++; meth:=i; val$_{\theta_1}$:=x; oracle()

oracle = $\lambda$().
  match (!cnt) with         # number of P-moves played so far (max $|\tau|/2$)
  | i $\rightarrow$
    # if $i > 0$ and $i$-th P-move of $\tau$ is cr $m_j(v)$, with $m_j : \theta_1 \rightarrow \theta_2$, then
    # - if cr = ret then $d = 0$ and $\theta = \theta_2$
    # - if cr = call then $d = j$ and $\theta = \theta_1$
    # diverge if the last P-move played is different from cr $m_j(v)$
    if not (!meth = d and !val$_\theta$ $\overset{\triangle}{=}_\theta$ v) then diverge
    else for $m_i$ in $fresh$(!val$_\theta$) do ref$_i$ := $m_i$

    # if $(i+1)$-th O-move of $\tau$ is cr$'$ $m_k(u)$, with $m_k : \theta_1 \rightarrow \theta_2$, then
    # - if cr$'$ = ret then $c = 0$
    # - if cr$'$ = call then $c = k$
    if c then let x = (!ref$_k$)u in      # call $m_k(u)$
              cnt++; meth:=0; val$_{\theta_2}$:=x; oracle(); !val$_{\theta_2}$
    else val$_{\theta_2}$:=u                      # return $u$

main = oracle()
```

Figure 4.2: The client $C_{\tau,\mathcal{P},\mathcal{A}}$.

and **diverge**$_\theta$ a term that on evaluation diverges by infinite recursion. We then construct a client $C_{\tau,\mathcal{P},\mathcal{A}}$ as in Figure 4.2.

The code is structured as follows.

1. We start off by defining global references:
   - *cnt* counts the number of $P$ (Library) moves played so far;
   - *meth* stores an index that records the move made by P: if the move was a return then *meth* stores 0; if it was call to $m_i$ then *meth* stores $i$;
   - each $ref_i$ will store the method $m_i \in \mathcal{P} \cup \mathcal{P}'$, either since the beginning (if $m_i \in \mathcal{P}$), or once P plays it (if $m_i \in \mathcal{P}'$);
   - each $val_\theta$ will be used for storing the value played by P in their last move.

   In the latter case above, there is a light abuse of syntax as $\theta$ can be a product type, of which HOLi does not have references. But we can in fact simulate references of arbitrary type by several HOLi references.
2. For each $m_i : \theta_1 \rightarrow \theta_2 \in \mathcal{A}$, we define a public method $m_i$ that simulates the behaviour of O whenever $m_i$ is called in $\tau$:

  - it starts by increasing *cnt*, as a call to $m_i$ corresponds to a P-move being played;
  - it continues by storing $i$ and $x$ in *meth* and $val_{\theta_1}$ respectively;
  - it calls the private method *oracle*, which is tasked with simulating the rest of $\tau$ and storing the value that $m_i$ will return in $val_{\theta_2}$;
  - it returns the value in $val_{\theta_2}$.

3. For each $m_i : \theta_1 \to \theta_2 \in \mathcal{A}'$ we produce a method just like above, but keep it private (for the time being).
4. The method *oracle* performs the bulk of the computations, by checking that the last move played by P was the expected one and selecting the next move to play (and playing it if is a call).
   - The oracle is called after each P-move is played, so it starts with increasing *cnt*.
   - It then performs a case analysis on the value of *cnt*, which above we denote collectively by assuming the value is $i$ – this notation hides the fact that we have one case for each of the finitely many values of $i$.

     For each such $i$, the oracle first checks if the previous P-move (if there was one), was the expected one. If the move was a call, it checks whether the called method was the expected one (via an appropriate value of $d$), and also whether the value was the expected one. Value comparisons ($\overset{\triangle}{=}_\theta$) only compare the integer components of $\theta$, since we cannot compare method names. If this check is successful, the oracle extracts from $u$ any method names played fresh by P and stores them in the corresponding $ref_i$.

     Next, the oracle prepares the next move. If, for the given $i$, the next move is a call, then the oracle issues the call, stores the return value of that call, increases *cnt* and recurs to itself – when the issued call returns, it would be through a P-move. If, on the other hand, the next move is a return, the oracle simply stores the value to be returned in the respective *val* reference – this would allow to the respective $m_i$ to return that value.
5. The **main** method simply calls the oracle.

Let us begin with useful definitions. First, let us consider the game semantics for HOLi with all call counters removed since they do not affect computation. Let $L$ be a library with public names $\mathcal{P}$ and abstract names $\mathcal{A}$ that produces a trace $\tau$. Let $C_{\tau,\mathcal{P},\mathcal{A}}$ be the client constructed from $\tau$, which we shall shorthand as $C_\tau$ assuming the correct name sets have been provided. Finally, let us annotate every move in $\tau$ with subscripts $O$ and $P$ for its polarity, starting from $O$ since libraries are always called first.

**Definition 4.22** (Client O-configurations)**.** Let library trace $\tau$ be of the form $\tau_1\tau_2$, where $\tau_1$ is the portion of $\tau$ that has been played so far. We define the set of opponent configurations $\mathtt{Conf}_{\tau_2}$ that play the remainder trace $\tau_2$ of trace $\tau$ to be

$$(\mathcal{E}_{\tau_1}, R, S_{\tau_1}, \mathcal{P}_{\tau_1}, \mathcal{A}_{\tau_1}) \in \mathtt{Conf}_{\tau_2}$$

where

- $R$ is the initial repository obtained from client $C_\tau$;
- $S_{\tau_1}$ has the same domain as the initial store $S$ obtained from client $C_\tau$ and defines values $\texttt{cnt} \mapsto \mathit{len}(\tau_1)/2$ and $\texttt{ref}_i \mapsto m_i$ for all $m_i$ revealed in $\tau_1$;
- $\mathcal{P}_{\tau_1} = \mathcal{A} \uplus \{m_in \in \mathcal{A}' \mid m_i \in \tau_1\}$, for $\mathcal{A}, \mathcal{A}'$ as defined initially in $C_\tau$;
- $\mathcal{A}_{\tau_1} = \mathcal{P} \uplus \{m_in \in \mathcal{P}' \mid m_i \in \tau_1\}$, for $\mathcal{P}, \mathcal{P}'$ as defined initially in $C_\tau$;
- and $\mathcal{E}_{\tau_1} = f(\lceil \tau_1 \rceil)$ where $\lceil \tau \rceil$ removes all closed calls in $\tau$ as defined in

$$\lceil \tau \rceil = \begin{cases} \lceil \tau' \tau''' \rceil & \text{if } \tau \text{ is of the form } \tau'\texttt{call}(m,v)\tau''\mathit{ret}(m,v)\tau''' \\ \tau & \text{otherwise} \end{cases}$$

and

$$\begin{aligned}
&f(\tau'\texttt{call}(m,v)_o) = \\
&\quad (\texttt{let } x = \bullet \texttt{ in cnt++;meth} := 0; \texttt{val}_{\theta_2} := x; \texttt{oracle}(); !\texttt{val}_{\theta_2}, m) :: f(\tau') \\
&f(\texttt{call}(m,v)_o) = \\
&\quad (\texttt{let } x = \bullet \texttt{ in cnt++;meth} := 0; \texttt{val}_{\theta_2} := x; \texttt{oracle}(); !\texttt{val}_{\theta_2}, m) :: [] \\
&f(\tau'\texttt{call}(m,v)_p) = m :: f(\tau') \\
&f(\texttt{call}(m,v)_p) = m :: []
\end{aligned}$$

**Lemma 4.23.** Let library trace $\tau_L$ be of the form $\tau_1\tau_2$, such that $\tau_1$ is a prefix of $\tau_L$. For all configurations $\mathbb{C}_{\tau_2} \in \texttt{Conf}_{\tau_2}$, $\mathbb{C}_{\tau_2}$ produces $\tau_2$.

*Proof.* Let $\tau_L$ be a library trace of the form $\tau_p\tau$. We prove that $\mathbb{C}_\tau$ produces $\tau$ for all $\mathbb{C}_\tau \in \texttt{Conf}_\tau$ by induction on the length of $\tau$.

***Base Cases:***

- if $\tau = \texttt{call}(m,v)$, then we know $\mathbb{C}_\tau \to (m :: \mathcal{E}_{\tau_p}, mv, \dots)_p$ produces a valid OQ move since $m$ must have been revealed as an initial public name or in $\tau_p$ for it to appear as a call at this point in the trace.
- if $\tau = \texttt{ret}(m,v)$, then we know $\mathbb{C}_\tau \to (\mathcal{E}', v, \dots)_p$, where $\mathcal{E}_{\tau_p} = \texttt{call}(m,v') :: \mathcal{E}'$, produces a valid OA move since $m$ must appear at the top of the evaluation stack for a return to appear at this point in the trace.

We thus have base cases for *odd length suffixes.*

***Inductive Cases:***

- if $\tau = \texttt{call}(m,v)\texttt{call}(m',v')\tau'$, then we have the OQ move

$$\mathbb{C}_\tau \to (m :: \mathcal{E}_{\tau_p}, mv, \dots)_p \twoheadrightarrow (m :: \mathcal{E}_{\tau_p}, \texttt{oracle}(); !\texttt{val}_{\theta_2}, \dots)_p \to (\dots, E[m'v'], \dots)_p$$

where $E$ is $(E');!\texttt{val}_{\theta_2}$ and $E'$ is defined from line 26 to line 28 in the client code, which correctly updates the store. So far, $\mathbb{C}_\tau$ produces the same trace up to the next move. We then have the PQ move

$$(m :: \mathcal{E}_{\tau_p}, E[m'v'], \dots)_p \to ((E, m') :: m :: \mathcal{E}_{\tau_p}, \dots)_o$$

which produces the next valid move. At this point, we can observe that $((E, m') :: m :: \mathcal{E}_{\tau_p}, \dots)_o \in \texttt{Conf}'_\tau$, so we know $\tau'$ is produced by the inductive hypothesis. Thus, $\tau$ is produced.

- if $\tau = \texttt{call}(m, v)\texttt{ret}(m', v')\tau'$, since we have a return move as the second move this time, we have the OQ move

$$\mathbb{C}_\tau \to (m :: \mathcal{E}_{\tau_p}, mv, \dots)_p \twoheadrightarrow (m :: \mathcal{E}_{\tau_p}, \texttt{val}_{\theta_2} := v'; !\texttt{val}_{\theta_2}, \dots)_p \to (\dots, v', \dots)_p$$

which produces the first move. We then have the PA move

$$(\mathcal{E}_{\tau_p}, v', \dots)_p \to (\mathcal{E}', \dots)_o$$

which produces the second move since $\mathcal{E}_{\tau_p}$ must be of the form $m' :: \mathcal{E}'$. As before, since the store has been correctly updated by internal moves, $(\mathcal{E}', \dots)_o \in \texttt{Conf}'_\tau$, so we know $\tau'$ is produced by the inductive hypothesis. Thus, $\tau$ is produced.

- if $\tau = \texttt{ret}(m, v)\texttt{call}(m', v')\tau'$, then it must be the case that $\mathcal{E}_\tau = (\texttt{let}\ x = \bullet\ \texttt{in}\ \texttt{cnt++}; \texttt{meth} := 0; \texttt{val}_{\theta_2} := x; \texttt{oracle}(), m) :: \mathcal{E}'$. We have the OA move

$$\mathbb{C}_\tau \to (\mathcal{E}', \texttt{let}\ x = v\ \texttt{in}\ \dots, \dots)_p \twoheadrightarrow (\mathcal{E}', \texttt{oracle}(); !\texttt{val}_{\theta_2}, \dots)_p \to (\mathcal{E}', E[m'v'], \dots)_p$$

where $E$ is the context for `oracle`, which produces the first move. From here we have OQ move

$$(\mathcal{E}', E[m'v'], \dots)_p \to ((E, m') :: \mathcal{E}', \dots)_o$$

which produces the second move. Since the store is correctly updated internally, we know $((E, m') :: \mathcal{E}', \dots)_o \in \texttt{Config}'_\tau$, so $\mathbb{C}_{\tau'}$ produces $\tau'$ by the inductive hypothesis. Thus, $\tau$ is produced.

- if $\tau = \texttt{ret}(m, v)\texttt{ret}(m', v')\tau'$, we have the OA move

$$\mathbb{C}_\tau \to (\mathcal{E}', \texttt{let}\ x = v\ \texttt{in}\ \dots, \dots)_p \twoheadrightarrow (\mathcal{E}', !\texttt{val}_{\theta_2}, \dots)_p \to (\mathcal{E}', v', \dots)_p$$

which produces the first move. From here, we have PA move

$$(\mathcal{E}', v', \dots)_p \to (\mathcal{E}'', \dots)$$

since $\mathcal{E}'$ must have been of the form $m' :: \mathcal{E}''$ for a return to $m'$ to appear on the trace. Since the internal moves correctly update the store, we know that $(\mathcal{E}'', \dots) \in \texttt{Config}'_\tau$, so $\mathbb{C}_{\tau'}$ produces $\tau'$ by the inductive hypothesis. Thus $\tau$ is produced.

If $\tau'$ is empty, these serve as base cases for *even length suffixes.* With all cases proven (odd and even base cases, and the inductive cases), we have that $\tau$ is always possible to produce with any $\mathbb{C}_\tau \in \mathtt{Conf}_\tau$. □

***Theorem 4.13 (Definability)*** Let $L$ be a library and $(\tau, \rho) \in [\![L]\!]$. There is a good client compatible with $L$ such that $(\tau, \rho') \in [\![C]\!]$ for some $\rho'$.

*Proof.* Given a library $L$ and trace produced $\tau$, we construct client $C_\tau$. Since $C_\tau$ has a main method, we begin from a proponent configuration $(\mathtt{oracle}(), [], R, \mathcal{A}, \mathcal{P})_p$. Since the library cannot return without being called first, we know the next move is a call, so $\tau$ is of the form $\mathtt{call}(m, v)\tau'$. Thus, we have the following transitions

$$([], \mathtt{oracle}(), R, \mathcal{A}, \mathcal{P})_p \twoheadrightarrow ([], E[mv], R, \mathcal{A}, \mathcal{P})_p \rightarrow ((E, m) :: [], R, \mathcal{A}', \mathcal{P})_o$$

From this point, if $\tau'$ is empty, we have shown that $\tau$ can be produced by $C_\tau$. If $\tau'$ is not empty, we have a trace $\tau$ with suffix $\tau'$ and prefix $\mathtt{call}(m, v)$. By Lemma 4.23, we know that $\tau'$ can be produced by any configuration in $\mathtt{Config}_{\tau'}$. Since $((E, m) :: [], R, \mathcal{A}', \mathcal{P})_o \in \mathtt{Config}_{\tau'}$, we know that $((E, m) :: [], R, \mathcal{A}', \mathcal{P})_o$ is able to produce $\tau'$. We thus have that $C_\tau$ can produce $\tau$. □

## 4.5 Symbolic Semantics

So far, we have presented a concrete trace semantics for higher-order terms. However, checking libraries for errors using this trace semantics is infeasible, even when the traces are bounded in length, because ground values would be handled concretely. In particular, integer values provided by $O$ as arguments to calls or as return values range over infinitely many integers. One approach (symbolic execution) mitigates this limitation by executing the semantics symbolically using symbolic values for integers and a path condition to constrain the plausible concrete values that every symbolic value may take. We use this technique to devise a symbolic version of the trace semantics, corresponding to a symbolic execution for higher-order stateful libraries, which will enable us in the next sections to introduce a practical method (and implementation) to check libraries for errors. The symbolic semantics is fully formal, closely following the developments of the previous sections, and allows us to prove a strong form of correspondence between the concrete and symbolic semantics (a bisimulation).

For the symbolic semantics defined in this section, all names played by $O$ shall be fresh. Using fresh names for methods played by $O$ is sound because the effect of $O$ calling a higher-order public method with either an argument $m$ (where $m$ is another public method), or an argument $\lambda x.mx$, is equivalent as far as reachability of an error is concerned. In the latter case, the client semantics would create a fresh name $m'$, bind it to $\lambda x.mx$, and pass $m'$ as an argument. We therefore just focus on this latter, simpler, case. Formally speaking, the semantics lives in nominal sets [66].

The symbolic zsemantics involves terms that may contain symbolic values for integers. We thus extend the syntax for values and terms to include such values, and abuse notation by continuing to use $M$ to range over them. We let `SInts` be a set of symbolic integers ranged over by $\kappa$ and variants, and define:

$$\begin{aligned} \mathit{Sym.Values} \quad \tilde{v} &::= m \mid i \mid () \mid \kappa \mid \tilde{v} \oplus \tilde{v} \mid \langle \tilde{v}, \tilde{v} \rangle \\ \mathit{Sym.Terms} \quad M &::= \cdots \mid \kappa \end{aligned}$$

where, in $\tilde{v} \oplus \tilde{v}$, not both $\tilde{v}$ can be integers. We moreover use a symbolic environment to store symbolic values for references, but also to keep track of symbolic terms built from arithmetic performed with symbolic integers. More precisely, we let $\sigma$ be a finite partial map from the set $\texttt{SInts} \cup \texttt{Refs}$ to symbolic values. Finally, we use $pc$ to range over program conditions, which will be quantifier-free first-order formulas with variables taken from `SInts`, and with $\top, \bot$ denoting true and false respectively.

The semantics for closed symbolic terms involves configurations of the form $(M, R, \sigma, pc, k)$. Its rules include copies of those from Figure 2.1 (top) where the $pc$ and $\sigma$ are simply carried over. For example:

$$(E[\lambda x.M], R, \sigma, pc, k) \rightarrow_s (E[m], R \uplus \{m \mapsto \lambda x.M\}, \sigma, pc, k)$$

where $m$ is fresh. On the other hand, the following rules directly involve symbolic reasoning:

$$\begin{aligned}
&(E[\texttt{assert}(\kappa)], R, \sigma, pc, k) \rightarrow_s (E[\texttt{assert}(0)], \sigma, pc \wedge (\kappa = 0), k) \\
&(E[\texttt{assert}(\kappa)], R, \sigma, pc, k) \rightarrow_s (E[()], R, \sigma, pc \wedge (\kappa \neq 0), k) \\
&(E[!r], R, \sigma, pc, k) \rightarrow_s (E[\sigma(r)], R, \sigma, pc, k) \\
&(E[r := \tilde{v}], R, \sigma, pc, k) \rightarrow_s (E[()], R, \sigma[r \mapsto \tilde{v}], pc, k) \\
&(E[\tilde{v}_1 \oplus \tilde{v}_2], R, \sigma, pc, k) \rightarrow_s (E[\kappa], R, \sigma \uplus \{\kappa \mapsto \tilde{v}_1 \oplus \tilde{v}_2\}, pc, k) \quad \text{where } \kappa \text{ is fresh} \\
&(E[\texttt{if } \kappa \texttt{ then } M_1 \texttt{ else } M_0], R, \sigma, pc, k) \rightarrow_s (E[M_0], R, \sigma, pc \wedge (\kappa = 0), k) \\
&(E[\texttt{if } \kappa \texttt{ then } M_1 \texttt{ else } M_0], R, \sigma, pc, k) \rightarrow_s (E[M_1], R, \sigma, pc \wedge (\kappa \neq 0), k)
\end{aligned}$$

and where $\tilde{v}_1 \oplus \tilde{v}_2$ is a symbolic value (for $i_i \oplus i_2$ the rule from Figure 2.1 applies).

We now extend the symbolic setting to the trace semantics. We define symbolic configurations for $P$ and $O$ respectively as:

$$(\mathcal{E}, M, R, \mathcal{P}, \mathcal{A}, \sigma, pc, k)_p \qquad (\mathcal{E}, l, R, \mathcal{P}, \mathcal{A}, \sigma, pc, k)_o$$

with evaluation stack $\mathcal{E}$, proponent term $M$, counters $k, l \in \mathbb{N}$, method repository $R$,

$$(\widetilde{\text{INT}})\quad \frac{(M,R,\sigma,pc,k) \to_s (M',R',\sigma,pc',k')}{(\mathcal{E},M,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \to_s (\mathcal{E},M',R',\mathcal{P},\mathcal{A},\sigma',pc',k')_p}$$

$$(\widetilde{\text{PQ}})\quad (\mathcal{E},E[m\tilde{v}],R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \xrightarrow{\texttt{call}(m,\tilde{v})}_s ((m,E)::\mathcal{E},l_0,R,\mathcal{P}',\mathcal{A},\sigma,k)_o$$

$$(\widetilde{\text{OQ}})\quad (\mathcal{E},l,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_o \xrightarrow{\texttt{call}(m,\tilde{v})}_s ((m,l+1)::\mathcal{E},m\tilde{v},R,\mathcal{P},\mathcal{A}',\sigma,pc,k)_p$$

$$(\widetilde{\text{PA}})\quad ((m,l)::\mathcal{E},\tilde{v},R,\mathcal{P},\mathcal{A},\sigma,pc,k)_p \xrightarrow{\texttt{ret}(m,\tilde{v})}_s (\mathcal{E},l,R,\mathcal{P}',\mathcal{A},\sigma,pc,k)_o$$

$$(\widetilde{\text{OA}})\quad ((m,E)::\mathcal{E},l,R,\mathcal{P},\mathcal{A},\sigma,pc,k)_o \xrightarrow{\texttt{ret}(m,\tilde{v})}_s (\mathcal{E},E[\tilde{v}],R,\mathcal{P},\mathcal{A}',\sigma,pc,k)_p$$

$(\widetilde{\mathbf{PC}})$ $m \in \mathcal{A}$ and $\mathcal{P}' = \mathcal{P} \cup (\texttt{Meths}(\tilde{v}) \cap dom(R))$.

$(\widetilde{\mathbf{OC}})$ $m \in \mathcal{P}$ and $(\tilde{v}',\mathcal{A}') \in \mathsf{symval}(\theta,\mathcal{A})$ where $\theta$ is the expected type of $\tilde{v}$. Moreover:

$$\texttt{symval}(\theta,\mathcal{A}) = \begin{cases} \{((),\mathcal{A})\} & \text{if } \theta = \texttt{unit} \\ \{(\kappa,\mathcal{A} \uplus \{\kappa\}) \mid \ \kappa \text{ is fresh in } dom(\sigma) \uplus \mathcal{A}\} & \text{if } \theta = \texttt{int} \\ \{(m,\mathcal{A} \uplus \{m\}) \mid m \text{ is fresh in } dom(R) \uplus \mathcal{A}\} & \text{if } \theta = \theta_1 \to \theta_2 \\ \{(\langle \tilde{v}_1,\tilde{v}_2\rangle,\mathcal{A}_2) \mid (\tilde{v}_1,\mathcal{A}_1) \in \texttt{symval}(\theta_1,\mathcal{A}) & \text{if } \theta = \theta_1 \times \theta_2 \\ \qquad\qquad (\tilde{v}_2,\mathcal{A}_2) \in \texttt{symval}(\theta_2,\mathcal{A}_1)\} & \end{cases}$$

Rules $(\widetilde{\text{PQ}})$, $(\widetilde{\text{PA}})$ assume the condition $(\widetilde{\mathbf{PC}})$, and similarly for $(\widetilde{\text{OQ}})$,$(\widetilde{\text{OA}})$ and $(\widetilde{\mathbf{OC}})$.

Figure 4.3: Symbolic trace (game) semantics rules.

public method name set $\mathcal{P}$, $\sigma$ and $pc$ as previously. The abstract name set $\mathcal{A}$ is now a finite subset of $\texttt{Meths} \cup \texttt{SInts}$, as we also need to keep track of the symbolic integers introduced by $O$ (in order to be able to introduce fresh such names). The rules for the symbolic trace semantics are given in Figure 4.3. Note that $O$ always refreshes names it passes. This is a sound overapproximation of all names passed for the sake of analysis.

Similarly to Definition 4.2, we can define the symbolic semantics of libraries.

**Definition 4.24** (Symbolic Games)**.** Given library $L$, the symbolic semantics of $L$ is:

$$\begin{aligned}[\![L]\!]_s = \{(\tau,\rho) \mid & (L,\emptyset,\emptyset,\emptyset,\emptyset) \xrightarrow{bld}{}^* (\varepsilon,R,S,\mathcal{P},\mathcal{A}) \\ & \wedge (\varepsilon,0,R,\mathcal{P},\mathcal{A},S,\top,0)_o \xrightarrow{\tau}_s \rho \ \wedge \ \exists \mathcal{M}.\, \mathcal{M} \vDash \rho(\sigma)^\circ \wedge \rho(pc)\}\end{aligned}$$

where $\rho(\chi)$ is component $\chi$ in configuration $\rho$, and $\mathcal{M}$ is a model as defined in Section 4.7.1. We say that $[\![L]\!]_s$ *fails* if it contains some $(\tau,(\mathcal{E},E[\texttt{assert}(0)],\cdots))$.

The symbolic rules follow those of the concrete semantics, the biggest change being the treatment of symbolic values played by $O$. Condition $(\widetilde{\mathbf{OC}})$ stipulates that $O$ plays distinct fresh symbolic integers as well as fresh method names, in each appropriate position in $\tilde{v}$, and all these names are included in the set $\mathcal{A}$.

**Example 4.25.** As with Example 4.3, we consider the DAO attack. Running the symbolic semantics, we find the following minimal class of errors. We write $\sigma_{\tilde{v}}$ for a symbolic environment $[bal \mapsto \tilde{v}]$.

$$(\varepsilon,2,\sigma_{100},k_0)_o \xrightarrow{wdraw(\kappa_1)?} ((wdraw,1),wdraw(\kappa_1),\sigma_{100},2)_p$$

$$
\begin{aligned}
&\to^* ((wdraw,1), E[send(\kappa_1)], \sigma_{100}, 1)_p \xrightarrow{send(\kappa_1)?} ((send,E),2,\sigma_{100},1)_o \\
&\xrightarrow{wdraw(\kappa_2)?} ((wdraw,1), wdraw(\kappa_2), \sigma_{100}, 1)_p \\
&\to^* ((wdraw,1), E'[send(\kappa_2)], \sigma_{100}, 0)_p \xrightarrow{send(\kappa_2)?} ((send,E),2,\sigma_{100},0)_o \\
&\xrightarrow{send(())!} ((wdraw,1), E'[()], \sigma_{100}, 0)_p \\
&\to^* ((wdraw,1), (), \sigma_{100-\kappa_2}, 0)_p \xrightarrow{wdraw(())!} ((send,E),1,\sigma_{100-\kappa_2},0)_o \\
&\xrightarrow{send(())!} ((wdraw,1), E[()], \sigma_{100-\kappa_2}, 1)_p \\
&\to^* ((wdraw,1), E[\texttt{assert}(!bal \geq 0)], \sigma_{100-\kappa_2-\kappa_1}, 1)_p
\end{aligned}
$$

For this to be a valid error, we require $(\kappa_1, \kappa_2 \leq 100) \wedge (100 - \kappa_2 - \kappa_1 < 0)$ to be satisfiable. Taking assignment $\{\kappa_1 \mapsto 100, \kappa_2 \mapsto 1\}$, we show the path is valid. $\Diamond$

## 4.6 Bounded Analysis for Libraries

Definition 4.24 states how the symbolic trace semantics can be used to independently check libraries for errors. As with the trace semantics in Definition 4.2, this is strongly normalising when given an upper limit to the call counters. As such, $[\![L]\!]_s$ with counter bounds $k_0, l_0 \in \mathbb{N}$, for $k, l$ respectively, defines a finite set (modulo selecting of fresh names) of reachable valid configurations within $k \leq k_0, l \leq l_0$, where validity is defined by the satisfiability of the symbolic environment $\sigma$ and the path condition $pc$ of the configuration reached. By virtue of Theorems 4.29 and 4.11, every valid reachable configuration that is failed (i.e. evaluates an invalid assertion) is realisable by some client. And viceversa.

Given a library $L$, taking $\mathcal{F}[\![L]\!]_s$ to be all reachable final configurations, we have the exhaustive set of paths $L$ can reach—also called the computation tree of $L$. In $\mathcal{F}[\![L]\!]_s$, every failed configuration $(\tau, \rho)$, i.e. such that $\rho$ holds a term $E[\texttt{assert}(0)]$, defines a reachable assertion violation, where $\tau$ is a true counterexample. Hence, to check $L$ for assertion violations it suffices to produce a finite representation of the set $\mathcal{F}[\![L]\!]_s$. The approach we have been building to do this is that of bounding the depth of analysis by setting an upper bound to the call counters, using a name generator ensure deterministic creation of fresh names, to thus exhaustively search all final configurations for failed elements. This has the effect of finding all assertion violations in the depth-bounded computation tree of $L$. In Section 4.8. we implement this routine and test it.

## 4.7 Soundness of Symbolic Games

We shall now follow by proving soundness of our symbolic semantics with respect to errors found. This depends on two underlying results: the bisimilarity of concrete and symbolic game configurations (Lemma 4.29)—provided a unifying model for them exists—and the extensional equivalence between taking only $O$-refreshing moves (i.e. the opponent always refreshes names) and otherwise (Lemma 4.31). The main result of this section is thus establishing soundness: a trace and a specific configuration can be achieved symbolically iff they can be achieved concretely as well. As suggested before, we will need to quantify this statement as, by construction, the symbolic semantics requires $O$ to always place fresh method names, whereas in the symbolic semantics $O$ is given the freedom to play old names as well. Intuitively, we show that the symbolic semantics corresponds (via *bisimilarity*) to a restriction of the concrete semantics where $O$ plays fresh names only. We then show this restriction is sound in the sense that it does not affect our ability to identify whether a configuration can fail. We make this precise below.

**Theorem 4.26** (Soundness)**.** For any $L$, $[\![L]\!]$ fails iff $[\![L]\!]_s$ fails.

*Proof.* Lemma 4.29 implies that $[\![L]\!]_s$ fails iff $[\![L]\!]$ fails with $O$-refreshing transitions, which in turns occurs iff $[\![L]\!]$ fails, by Lemma 4.31. □

### 4.7.1 Bisimilarity of Concrete and Symbolic Configurations

We start with the bisimilarity argument. A ***model*** $\mathcal{M}$ is a finite partial map from symbolic integers to concrete integers. Given such an $\mathcal{M}$ and a formula $\phi$, we define $\mathcal{M} \models \phi$ using a standard first-order logic interpretation with integers and arithmetic operators (in particular, we require that all symbolic integers in $\phi$ are in the domain of $\mathcal{M}$). Moreover, for any symbolic term $M$ (or trace, move, etc.), we denote by $M\{\mathcal{M}\}$ the concrete term we obtain by substituting any symbolic integer $\kappa$ of $M$ with its corresponding concrete integer $\mathcal{M}(\kappa)$. Finally, given a symbolic environment $\sigma$, we define its formula representation $\sigma^\circ$ recursively by:

$$\emptyset^\circ = \top, \quad (\sigma \uplus \{r \mapsto v\})^\circ = \sigma^\circ, \quad (\sigma \uplus \{\kappa \mapsto v\})^\circ = \sigma^\circ \wedge (\kappa = v).$$

We now define notions for equivalence between symbolic and concrete configurations. The equivalence we require between concrete configurations and their symbolic counterparts is behavioural equivalence, modulo $O$ playing fresh names.

**Definition 4.27.** Let $\mathcal{M}$ be a model. For any concrete configuration $\rho = (\mathcal{E}, \chi, R, S, \mathcal{P}, \mathcal{A}, k)$ and symbolic configuration $\rho_s = (\mathcal{E}', \chi', R', \mathcal{P}', \mathcal{A}', \sigma, pc, k')$, we say they are ***equivalent in*** $\mathcal{M}$, written $\rho =_{\mathcal{M}} \rho_s$, if:

- $(\mathcal{E}, \chi, R) = (\mathcal{E}', \chi', R')\{\mathcal{M}\}, \mathcal{P} = \mathcal{P}', \mathcal{A} = \mathcal{A}' \cap \texttt{Meths}$ and $S = (\sigma \restriction \texttt{Refs})\{\mathcal{M}\}$;
- $\mathrm{dom}(\mathcal{M}) = (\mathcal{A}' \cup \mathrm{dom}(\sigma)) \cap \texttt{SInts}$ and $\mathcal{M} \vDash pc \wedge \sigma^\circ$.

More precisely, a transition $\rho \xrightarrow{\chi} \rho'$ is called *O-refreshing* if, when $\rho$ is an $O$-configuration and $\chi = \texttt{call/ret}(m, v)$ then all names in $v$ are fresh and distinct. To prove the correspondence between our symbolic and concrete semantics we shall be using a new notion of bisimulation that takes into account the existence of a model $\mathcal{M}$ that relates a symbolic configuration to a concrete one. We shall be abusing notation to use familiar syntax seen before in Definition 4.19. It should be noted that Definition 4.28 below is different from Definition 4.19 in that they handle different kinds of configurations, and Definition 4.28 additionally requires a model that closes the symbolic configuration.

**Definition 4.28.** A finite set $\mathcal{R}$ with elements of the form $(\rho, \mathcal{M}, \rho_s)$ is a ***bisimulation in*** $\mathcal{M}$ if, whenever $(\rho, \mathcal{M}, \rho_s) \in \mathcal{R}$, written $\rho \, \mathcal{R}_{\mathcal{M}} \, \rho_s$ then $\rho =_{\mathcal{M}} \rho_s$ and, using $\chi$ to range over moves and $\varepsilon$ (i.e. no move):

- if $\rho \xrightarrow{\chi} \rho'$ is $O$-refreshing then there exists $\mathcal{M}' \supseteq \mathcal{M}$ such that $\rho_s \xrightarrow{\chi_s}_s \rho'_s$, with $\chi = \chi_s\{\mathcal{M}'\}$, and $\rho' \mathcal{R}_{\mathcal{M}'} \rho'_s$;
- if $\rho_s \xrightarrow{\chi}_s \rho'_s$ then there exists $\mathcal{M}' \supseteq \mathcal{M}$ such that $\rho \xrightarrow{\chi\{\mathcal{M}'\}}_G \rho'$ and $\rho' \mathcal{R}_{\mathcal{M}'} \rho'_s$.

We let $\sim$ be the largest bisimulation relation: $\rho \sim_{\mathcal{M}} \rho_s$ iff there is bisimulation $\mathcal{R}$ such that $\rho \mathcal{R}_{\mathcal{M}} \rho_s$.

We now show that concrete and symbolic configurations are bisimilar.

**Lemma 4.29.** Given $\rho, \rho_s$ a concrete and symbolic configuration respectively, and $\mathcal{M}$ a model such that $\rho =_{\mathcal{M}} (\rho')$, we have $\rho \sim_{\mathcal{M}} \rho_s$.

*Proof.* We want show that $\mathcal{R} = \{(\rho, \mathcal{M}, \rho_s) \mid \rho =_{\mathcal{M}} \rho_s\}$ is a bisimulation. First, we show that if $\rho \to \rho'$, being $O$-refreshing, then $\rho_s \to_s \rho'_s$ such that $(\rho', \mathcal{M}', \rho'_s)$ is in $\mathcal{R}$ for some $\mathcal{M}' \supseteq \mathcal{M}$. By cases on the transition $\rho \to \rho'$:

1. If $\rho \to \rho'$ is one of the return moves, then we have the following possible transitions:
   (a) If $(\mathcal{E}, E[\texttt{assert}(0)], R, S, \mathcal{P}, \mathcal{A}, k)_p \not\to$, then we have the corresponding symbolic final configuration:

$$(\mathcal{E}, E'[\texttt{assert}(0)], R, \mathcal{P}, \mathcal{A}, \sigma, pc, k)_p$$

   From the assumptions, we know that $\mathcal{M} \vDash pc \wedge \sigma^\circ$. It is also the case that $E'\{\mathcal{M}\}$ is equivalent to $E$, and $\rho'$ and $\rho'_s$ are equivalent terminal configurations.
   (b) If $(\emptyset, v, R, S, \mathcal{P}, \mathcal{A}, k)_p \not\to$, the proof is similar to (a).
2. If $\rho \to \rho'$ is an (Int) move, we have that $\rho_s \to_s \rho'_s$ such that $\rho' \sim \rho'_s$ by soundness of the symbolic execution (Lemma 4.30).
3. If $\rho \to \rho'$ is a (Pq) move, then we have the following transition

$$(\mathcal{E}, E[mv], R, S, \mathcal{P}, \mathcal{A}, k)_p \xrightarrow{\texttt{call}(m,v)} ((m, E) :: \mathcal{E}, l_0, R', S, \mathcal{P}', \mathcal{A}, k)_o$$

   with its corresponding symbolic equivalent

$$(\mathcal{E}', E'[mv'], \ldots, \sigma, pc, k)_p \xrightarrow{\texttt{call}(m,v')}_s ((m, E') :: \mathcal{E}', l_0, \ldots, \sigma, pc, k)_o$$

From the assumptions, we know $\mathcal{M}(v') = v$. In addition, since $E'[mv'] = E[mv]$ under $\mathcal{M}$, we have that $(m, E') :: \mathcal{E}' = (m, E) :: \mathcal{E}$, and similarly for other components, so $\rho' =_{\mathcal{M}} \rho'_s$, meaning $(\rho', \mathcal{M}, \rho'_s) \in \mathcal{R}$.

4. If $\rho \to \rho'$ is a (PA) move, then we have the following transition

$$((m, l) :: \mathcal{E}, v, R, S, \mathcal{P}, \mathcal{A}, k)_p \xrightarrow{\mathtt{ret}(m,v)} (\mathcal{E}, l, R', S, \mathcal{P}', \mathcal{A}, k)_o$$

with its corresponding symbolic equivalent

$$((m, l) :: \mathcal{E}', v', \ldots, \sigma, pc, k)_p \xrightarrow{\mathtt{ret}(m,v')}_s (\mathcal{E}', l, \ldots, \sigma, pc, k)_o$$

From the assumptions, we know $\mathcal{M}(v') = v$. Since the original stacks are equivalent under $\mathcal{M}$, we have that $\mathcal{E} =_{\mathcal{M}} \mathcal{E}'$, and similarly for other components, so $\rho' =_{\mathcal{M}} \rho'_s$, meaning $(\rho', \mathcal{M}, \rho'_s) \in \mathcal{R}$.

5. If $\rho \to \rho'$ is an (OQ) move, $O$-refreshing, then we have the following transition

$$(\mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o \xrightarrow{\mathtt{call}(m,v)} ((m, l+1) :: \mathcal{E}, mv, R, S, \mathcal{P}, \mathcal{A}', k)_p$$

with its corresponding symbolic equivalent

$$(\mathcal{E}', l, \ldots, \sigma, pc, k)_o \xrightarrow{\mathtt{call}(m,v')}_s ((m, l+1) :: \mathcal{E}', mv', \ldots, \sigma, pc, k)_p$$

Let us choose $\mathcal{M}' = \mathcal{M}[v' \mapsto v]$. Since the original stacks are equivalent under $\mathcal{M}$, we have that $((m, l+1) :: \mathcal{E}) =_{\mathcal{M}} ((m, l+1) :: \mathcal{E}')$, and similarly for other components, so $\rho' =_{\mathcal{M}'} \rho'_s$, meaning $(\rho', \mathcal{M}', \rho'_s) \in \mathcal{R}$.

6. If $\rho \to \rho'$ is an (OA) move, $O$-refreshing, then we have the following transition

$$((m, E) :: \mathcal{E}, l, R, S, \mathcal{P}, \mathcal{A}, k)_o \xrightarrow{\mathtt{ret}(m,v)} (\mathcal{E}, E[v], R, S, \mathcal{P}, \mathcal{A}', k)_p$$

with its corresponding symbolic equivalent

$$((m, E') :: \mathcal{E}', l, \ldots, \sigma, pc, k)_o \xrightarrow{\mathtt{ret}(m,v')}_s (\mathcal{E}', E'[v'], \ldots, \sigma, pc, k)_p$$

Let us choose $\mathcal{M}' = \mathcal{M}[v' \mapsto v]$. Since the original stacks are equivalent under $\mathcal{M}$, we have that $\mathcal{E} =_{\mathcal{M}} \mathcal{E}$. Additionally, since $\mathcal{M}'$ extends $\mathcal{M}$, we know that $E[v] = E'[v']$ under $\mathcal{M}'$, and similarly for the remaining components, so $\rho' =_{\mathcal{M}'} \rho'_s$, meaning $(\rho', \mathcal{M}', \rho'_s) \in \mathcal{R}$.

The opposite direction is treated with similarly. □

**Lemma 4.30** (Soundness of symbolic execution)**.** For any concrete configuration $\eta = (M, R, S, k)$ and symbolic configuration $\eta' = (M', R', \sigma, pc, k)$, given an assignment $\mathcal{M} \vDash pc \wedge \sigma^\circ$ such that $M =_{\mathcal{M}} M'$, it is the case that $\eta \sim \eta'$.

*Proof.* Let $\mathcal{R} = \{(\eta, \mathcal{M}, \eta_s) \mid \eta =_{\mathcal{M}} \eta_s\}$ for any concrete configuration $\eta$ and symbolic configuration $\eta_s$. We want to show that $\mathcal{R}$ is a bisimulation. We now show that $\eta_s \rightarrow \eta'_s$ if $\eta \rightarrow \eta'$. By cases on $\eta \rightarrow \eta'$:

1. If we have a terminal rule, then we have the following cases.
   (a) for $(E[\texttt{assert}(0)], R, S, k) \not\rightarrow$ we have the equivalent final configuration

   $$(E'[\texttt{assert}(0)], R', \sigma, pc, k)$$

   Since $\eta =_{\mathcal{M}} \eta'$, and $\eta' =_{\mathcal{M}} \eta'_s$ since they are equivalent terminal configurations, it is the case that $(\eta', \mathcal{M}, \eta'_s) \in \mathcal{R}$.

   (b) for $(v, R, S, k) \not\rightarrow$ we have a similar proof to (a).
2. If $(E[\texttt{assert}(i)], R, S, k) \rightarrow (E[()], R, S, k)$ where $(i \neq 0)$, then we have the equivalent symbolic transition

   $$(E'[\texttt{assert}(i)], R', \sigma, pc, k) \rightarrow (E'[()], R', \sigma, pc, k)$$

   By assumption, we know $E =_{\mathcal{M}} E'$ and $R =_{\mathcal{M}} R'$, and similarly for other components, so $\eta' =_{\mathcal{M}} \eta'_s$. As such, we know $(\eta', \mathcal{M}, \eta'_s) \in \mathcal{R}$.
3. If $(E[!r], R, S, k) \rightarrow (E[S(r)], R, S, k)$, then we have the equivalent symbolic transition

   $$(E'[!r], R', \sigma, pc, k) \rightarrow (E'[\sigma(r)], R', \sigma, pc, k)$$

   Since $\eta =_{\mathcal{M}} \eta_s$, we know that $S =_{\mathcal{M}} \sigma$, meaning that $\sigma(r)\{\mathcal{M}\} = S(r)$. Thus, $(\eta', \mathcal{M}, \eta'_s) \in \mathcal{R}$.
4. If $(E[r := v], R, S, k) \rightarrow (E[()], R, S[r \mapsto v], k)$, then we have the equivalent symbolic transition

   $$(E'[r := v'], R', \sigma, pc, k) \rightarrow (E'[()], R', \sigma[r \mapsto \sigma(v')], pc, k)$$

   Since $\eta =_{\mathcal{M}} \eta_s$, we know that $S =_{\mathcal{M}} \sigma$ and $v' =_{\mathcal{M}} v$, meaning that $\sigma[r \mapsto v']\{\mathcal{M}\} = S[r \mapsto v]$. Thus, $(\eta', \mathcal{M}, \eta'_s) \in \mathcal{R}$.
5. If $(E[\pi_j\langle v_1, v_2\rangle], R, S, k) \rightarrow (E[v_j], R, S, k)$, then we have the equivalent symbolic transition

   $$(E'[\pi_j\langle v'_1, v'_2\rangle], R', \sigma, pc, k) \rightarrow (E'[v'_j], R', \sigma, pc, k)$$

   Since $\eta =_{\mathcal{M}} \eta_s$, we know that $\langle v_1, v_2\rangle =_{\mathcal{M}} \langle v'_1, v'_2\rangle$, so $v'_j\{\mathcal{M}\} = v_j$. Thus, $(\eta', \mathcal{M}, \eta'_s) \in \mathcal{R}$.
6. If $(E[i_1 \oplus i_2], R, S, k) \rightarrow (E[i], R, S, k)$ where $i = i_1 \oplus i_2$, prove as above.
7. If $(E[\lambda x.M], R, S, k) \rightarrow (E[m], R[m \mapsto \lambda x.M], S, k)$, then we have the equivalent symbolic transition

   $$(E'[\lambda x.M'], R', \sigma, pc, k) \rightarrow (E'[m], R'[m \mapsto \lambda x.M'], \sigma, pc, k)$$

   Since $\eta =_{\mathcal{M}} \eta_s$, we know that $E[m] =_{\mathcal{M}} E[m']$, so $v'_j\{\mathcal{M}\} = v_j$. Additionally, we know $M = M'\{\mathcal{M}\}$, so $R'[m \mapsto \lambda x.M'] =_{\mathcal{M}} R[m \mapsto \lambda x.M]$. Thus, $(\eta', \mathcal{M}, \eta'_s) \in \mathcal{R}$.

8. If $(E[\texttt{if}\ 0\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_0], R, S, k)$, then we have the equivalent symbolic transition

$$(E'[\texttt{if}\ 0\ \texttt{then}\ M_1'\ \texttt{else}\ M_0'], R', \sigma, pc, k) \to (E'[M_0'], R', \sigma, pc, k)$$

   Since $\eta =_{\mathcal{M}} \eta_s$, we know that $E[M_0] =_{\mathcal{M}} E[M_0']$. Thus, $(\eta', \mathcal{M}, \eta_s') \in \mathcal{R}$.

9. If $(E[\texttt{if}\ i\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, S, k) \to (E[M_1], R, S, k)$ where $i \neq 0$, prove as above.

10. If $(E[\texttt{let}\ x = v\ \texttt{in}\ M], R, S, k) \to (E[M\{v/x\}], R, S, k)$, then we have the equivalent symbolic transition

$$(E'[\texttt{let}\ x = v'\ \texttt{in}\ M'], R', \sigma, pc, k) \to (E'[M'\{v'/x\}], R', \sigma, pc, k)$$

   Since $\eta =_{\mathcal{M}} \eta_s$, we know that $E[M] =_{\mathcal{M}} E[M']$ and $v'\{\mathcal{M}\} = v$, so $E[M\{v/x\}] =_{\mathcal{M}} E[M'\{v'/x\}]$. Thus, $(\eta', \mathcal{M}, \eta_s') \in \mathcal{R}$.

11. If $(E[\texttt{letrec}\ f = \lambda x.M'\ \texttt{in}\ M], R, S, k)$

    $\to (E[M\{m/f\}], R[m \mapsto \lambda x.M'\{m/f\}], S, k)$

    prove by combining cases (7) and (10).

12. If $(E[mv], R, S, k) \to (E[(\!|M\{v/y\}|\!)], R, S, k+1)$, prove like (10).

13. If $(E[(\!|v|\!)], R, S, k) \to (E[v], R, S, k-1)$, then we have the equivalent symbolic transition

$$(E'[(\!|v'|\!)], R', \sigma, pc, k) \to (E'[v'], R', \sigma, pc, k-1)$$

   Since $v =_{\mathcal{M}} v'$, it is the case that $(\eta', \mathcal{M}, \eta_s') \in \mathcal{R}$.

In the opposite direction, all cases are treated similarly to the ones above, but we now additionally have symbolic branching cases not directly covered by the previous cases.

1. If $(E[\texttt{assert}(\kappa)], R, \sigma, pc, k) \to (E[\texttt{assert}(0)], \sigma, pc \wedge (\sigma(\kappa) = 0))$, then there exists $\mathcal{M}$ such that $E[\texttt{assert}(\kappa)]$ evaluates to $E[\texttt{assert}(0)]$, which requires it to satisfy $(\sigma(\kappa) = 0)$. As such, we know $\mathcal{M} \vDash \sigma(\kappa) = 0$, meaning that $0 =_{\mathcal{M}} \kappa$. We thus have the following equivalent concrete configuration

$$(E'[\texttt{assert}(0)], R', S, k) \not\to$$

   which holds since $\eta'$ and $\eta_s'$ are equivalent terminal configurations.

2. If $(E[\texttt{assert}(\kappa)], R, \sigma, pc, k) \to (E[()], \sigma, pc \wedge (\sigma(\kappa) \neq 0))$, prove as above.

3. If $(E[v_1 \oplus v_2], R, \sigma, pc, k) \to (E[\kappa], R, \sigma[\kappa \mapsto \sigma(v_1) \oplus \sigma(v_2)], pc, k)$, then we have the following equivalent concrete transition

$$(E'[i_1 \oplus i_2], R', S, k) \to (E'[i], R', S, k)$$

   From the assumption, we know $i_1 \oplus i_2 =_{\mathcal{M}} \sigma(v_1) \oplus \sigma(v_2)$, so by choosing $\mathcal{M}' = \mathcal{M}[\kappa \mapsto i]$, we have that $\eta'$ and $\eta_s'$ are equivalent under $\mathcal{M}$. As such, this case holds.

4. If $(E[\texttt{if}\ \kappa\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, \sigma, pc, k) \rightarrow (E[M_0], R, \sigma, pc \wedge (\sigma(\kappa) = 0), k)$, then there must exist a model $\mathcal{M} \vDash \kappa = 0$. We thus have the following equivalent concrete transition

$$(E'[\texttt{if}\ 0\ \texttt{then}\ M_1'\ \texttt{else}\ M_0'], R', S, k) \rightarrow (E'[M_0'], R', S, k)$$

From the assumption, we know $M_0 =_{\mathcal{M}} M_0'$, so $\eta'$ and $\eta_s'$ are equivalent under $\mathcal{M}$. As such, this case holds.

5. If $(E[\texttt{if}\ \kappa\ \texttt{then}\ M_1\ \texttt{else}\ M_0], R, \sigma, pc, k) \rightarrow (E[M_1], R, \sigma, pc \wedge (\sigma(\kappa) \neq 0), k)$, prove as above.

□

### 4.7.2 Extensional Equivalence of O-Refreshing Moves

Let us remember that the opponent in our symbolic games always refreshes names, i.e. only plays $O$-refreshing moves. As such, showing the correspondence between an $O$-refreshing semantics and one with a regular opponent is necessary to prove soundness. We thus argue in this section that $O$-refreshing transitions suffice for examining failure of concrete configurations. Indeed, suppose $\tau$ is a trace leading to fail where $O$ at some point plays an old name $m$ as an argument to some public name. Then, $\tau$ can be simulated by a trace $\tau'$ that uses a fresh $m'$ in place of $m$. We show this by case analysis on $m$. If $m$ is an $O$-name, then we can obtain $\tau'$ from $\tau$ by following exactly the same transitions, where some $P$-calls to $m$ can be safely replaced by calls to $m'$ (and accordingly for returns) without changing the semantics. If, on the other hand, $m$ is a $P$-name, then the simulation performed by $\tau'$ is somewhat more elaborate: some internal calls to $m$ will be replaced by $P$-calls to $m'$, immediately followed by the required calls to $m$ (and dually for returns).

**Lemma 4.31** (O-Refreshing)**.** Given a concrete configuration $\rho$, the following are equivalent:

1. $\rho$ fails using any kinds of transitions
2. $\rho$ fails using only $O$-refreshing transitions

*Proof.* Let us consider two games starting from $\rho$: (A) is allowed to play any kind of moves, while (B) is only allowed to play $O$-refreshing moves. We thus want to show that (A) and (B) are both allowed to reach an assertion violation.

***(2)*** $\Longrightarrow$ ***(1):*** Trivial. (A) is allowed to play all the moves that (B) can play.

***(1)*** $\Longrightarrow$ ***(2):*** By Lemma 4.32, we know any $\rho$ fails in (B) if it fails in (A). □

The above result requires the following lemma, which in turn requires some definitions. First, we call a name *phantom* if it is an opponent name created by refreshing a proponent name through an $O$-refreshing transition that has some equivalent original name in the non-refreshing semantics. We assume a method to identify phantom names by keeping track of them with regard to the non-refreshing semantics as computation progresses. We thus say that a configuration $\rho$ that is reached through $O$-refreshing transitions has a corresponding phantom names dictionary $\Phi$ that maps all phantom names $m$ in $\rho$ to their proponent-owned original names $\hat{m}$ in $\Phi(\rho)$. Let us also define a set $\mathcal{A}_\Phi \subseteq \mathcal{A}$ for all the phantom names in $\mathcal{A}$.

**Lemma 4.32.** Given a configuration $\rho$ with corresponding phantom names $\Phi$, it is the case that $\rho$ fails through $O$-refreshing transitions if $\Phi(\rho)$ fails.

*Proof.* Let (A) be a standard semantics where any moves are allowed. Let (B) be a semantics where only $O$-refreshing transitions are allowed. Suppose (B) starts from a configuration $\rho$ and has phantom names $\Phi$. We show this by induction on the number of steps to reach $\rho$. Let us consider proponent moves first, so $\rho = (\mathcal{E}, M, R, S, \mathcal{P}, \mathcal{A})_p$. Suppose $\Phi(\rho) \twoheadrightarrow \tau(\dots, \mathtt{assert}(0), \dots)$ in (A), by case analysis on $M$, we have the following.

1. $M$ is not of the form $E[mv]$ or is of the form $E[mv]$ where $m \in \mathcal{P}$:
   Let $\Phi(\rho) \to \hat{\rho}'$ via (A) semantics. Since $\rho$ is a proponent configuration, and the language features no name comparison, we know that the semantics are not affected by opponent names. Thus, we know $\hat{\rho}' = \Phi(\rho')$, so $\rho \to \rho'$ via (B). By the inductive hypothesis on $\hat{\rho}'$ and $\rho'$, we know (A) and (B) both fail.
2. $M$ is of the form $E[mv]$ and $m \in (\mathcal{A} \setminus \mathcal{A}_\Phi)$ ($m$ is not a phantom name):
   Let $\Phi(\rho) \xrightarrow{\mathtt{call}(m,\hat{v})} \hat{\rho}'$ in (A). It must be the case $\hat{\rho}' \xrightarrow{cr(\hat{m}',\hat{v}')} \hat{\rho}''$ for some call or return $cr$, since $\hat{\rho}'$ cannot fail without passing control to the proponent.
   With (B), we know $\rho \xrightarrow{\mathtt{call}(m,v)} \rho' \xrightarrow{cr(m',v')} \rho''$. Extending $\Phi$, we get $\Phi' = \Phi[m'_i \mapsto \hat{m}'_i]$ for every $m'_i, \hat{m}_i \in v', \hat{v}'$. Thus, we have $\Phi'(\rho'') = \hat{\rho}''$. By the inductive hypothesis on $\rho''$, $\hat{\rho}''$ and $\Phi'$, we know (A) and (B) fail.
3. $M$ is of the form $E[mv]$ where $m \in \mathcal{A}_\Phi$ ($m$ is a phantom name):
   Let $\Phi(m) = \hat{m}$. We have two cases on $\hat{m}$:
   (a) If $\hat{m} \in \mathcal{A}$, then we have the same situation as before.
   (b) If $\hat{m} \in \mathcal{P}$, then we know $\hat{\rho} \to (\dots, \hat{E}[(R(\hat{m}))\hat{v}], \dots)$ in (A). In (B), we have $\rho \xrightarrow{\mathtt{call}(m,v)} \rho'$. Since $\hat{m}$ must have been revealed to the opponent at some point in order for it to have been refreshed by (B), we have $\rho' \xrightarrow{\mathtt{call}(m,v)} (\dots, E[R(\hat{m})v'], \dots)$. Extending $\Phi$ to account for the indirect call of $\hat{m}$, we have $\Phi' = \Phi[m_i \mapsto \hat{m}_i]$ for every $m_i \in v'$ and $\hat{m}_i \in \Phi(v)$. Thus, we have $\Phi'(\dots, E[R(\hat{m})v'], \dots) = (\dots, \hat{E}[(R(\hat{m}))\hat{v}], \dots)$, so by the inductive hypothesis on them, we know (B) fails.

For the opponent moves, the cases are captured for every move $\hat{\rho} \xrightarrow{cr(m,\hat{v})} \hat{\rho}'$ in (A) and every move $\rho \xrightarrow{cr(m,v)} \rho'$ in (B) by extending $\Phi$ to be $\Phi' = \Phi[m_i \mapsto \hat{m}_i]$ for every name $m_i \in v$ and $\hat{m}_i \in \hat{v}$ introduced in the move. With this, by the inductive hypothesis on $\rho'$, $\hat{\rho}'$ and $\Phi'$, we know (B) fails in all the opponent cases. With this, we know (B) fails if (A) fails under $\Phi$. □

## 4.8 Implementation

We implemented the syntax and symbolic trace semantics (symbolic games) for HOLi in the 𝕂 semantic framework [71] as a proof of concept, and tested it on 70 sample libraries. The following sections will describe our implementation and the experiments in more detail. As already mentioned, we shall refer to this prototype implementation in 𝕂 as "HOLiK". The tool and its benchmark can be found at:

`https://github.com/LaifsV1/HOLiK`

### 4.8.1 The 𝕂 Framework

𝕂 is a semantic framework in which programming language semantics can be implemented and executed. Formal semantics for a language $\mathcal{L}$ are defined using configurations and rewrite rules, and compiled into an executable definition that can be used to run programs written in $\mathcal{L}$. The syntax of 𝕂 organises a semantics into program configurations and rewrite rules to transition from one configuration to another. The similarity between 𝕂's rewrite syntax and the notation used to formally present operational semantics makes 𝕂 convenient to implement analysis tools for programs for which a formal semantics has been defined. The idea is thus to implement the symbolic semantics of HOLi in 𝕂, and use it to symbolically execute HOLi programs.

We implemented our symbolic execution tool for HOLi by defining the symbolic transition rules in 𝕂. To illustrate this, following is the definition of proponent questions in 𝕂.

```
1 rule <k> M:KVar X:KVar ~> E => !OPPONENT! </k>
2       <eval> ES => M~>(Context E)~>ES </eval>
3       <repo> ... X |→ (fun (A:ATYPE) :(OTYPE) → T) ... </repo>
4       <abs> ... (M |→ _) ... </abs>
5       <pub> P => P[X <- (ATYPE→OTYPE)] </pub>
6       <lcount> I => L </lcount>
7       <lzero>  L </lzero>
8       <trace> TAU => TAU ~> call (M X) </trace>
9 [transition]
```

The keyword *rule* in line 1 is used to start the definition of a rewrite rule. In this case, the rule operates on a configuration with eight cells defined in a markup notation.

The cell <k>...</k> holds the main program term constructed by the built-in parser, which corresponds to $M$ in the game semantics. We then have <eval>...</eval> for the evaluation stack $\mathcal{E}$, <repo>...</repo> for the method repository $R$, <abs>...</abs> for abstract names $\mathcal{A}$, <pub>...</pub> for public names $\mathcal{P}$, <lcount>...</lcount> for the call counter $l$, <lzero>...</lzero> for the call counter bound $l_0$, and <trace>...</trace> for the trace produced $\tau$. A special term !OPPONENT! is used to tell $\mathbb{K}$ that the next configuration is an opponent configuration. In this rule, method application M X has no definition in repo, which requires the proponent to pass control to the opponent. The transition inside each cell is written using the operator =>, which states that the left hand side is to be rewritten into what is defined on the right hand side. Where omitted, configuration cells are left unmodified by the rule. The label [transition] tells $\mathbb{K}$ that the rule can be expanded non-deterministically. When provided the --search option, $\mathbb{K}$ runs a program by exhaustively expanding all non-deterministic transitions, which is useful for symbolic execution.

To ensure validity of transitions taken, we make external calls to Z3 [23], i.e., a transition is taken only if the path condition is satisfiable. To illustrate this, following is a non-deterministic symbolic rule for assertions.

```
rule <k>assert X:KVar => fail ... </k>
     <sig> SIG </sig>
     <pc>PC => PC (assert (= X 0))</pc>
     requires CheckSAT(SIG PC (assert (= X 0)))
[transition]
```

This rule states that an assertion assert(X) is able to fail (and be rewritten into a special term fail) if the symbolic environment conjoined with the path condition is satisfiable when the assertion is violated ($\texttt{X} = 0$). Here, CheckSAT is a custom function that parses an SMT-LIB 2 formula and them calls Z3 on it. To interface with Z3, a costume module was made that defines a data structure for SMT-LIB 2 and functions needed to call the solver. Specifically, the CheckSAT function flattens the SMT-LIB 2 data structure into a string, and then calls Z3 via a system call. The output of the system call is then parsed to know whether the formula passed is satisfiable or not.

Using $\mathbb{K}$'s option to exhaustively expand all transitions, we let $\mathbb{K}$ build a closure of all applicable rules. Providing $\mathbb{K}$ with a bound on the call counters, we produce a finite set of all reachable valid symbolic configurations up to a given depth of analysis—equivalent to finding every valid $\rho \in \mathcal{F}[\![L]\!]_s$, which thus implements our bounded symbolic execution.

### 4.8.2 Example Usage of HOLiK

HOLiK was implemented as two source files: holi.k and z3.k. The former contains an implementation of the syntax and game semantics for HOLi in $\mathbb{K}$, whereas the latter implements an interface between $\mathbb{K}$ and Z3. While $\mathbb{K}$ does have a built-in form of symbolic execution, we chose to implement symbolic execution manually to stay close to our definitions. This additionally required implementing our own interface with Z3. HOLiK is compiled from these source files using the 𝕂 command:

```
kompile -backend java holi.k
```

Despite the planned deprecation, we are using the Java backend because, to date, it is the only one providing both built-in substitution and exhaustive search.

**Example 4.33.** The file `reentrancy.holi` containing the simplification of the DAO smart contract introduced in Chapter 2 is defined as follows:

```
# set-bounds 2 1 #
import send :(int → unit)

public withdraw (m:int) :(unit) = {
  if (not (!funds < m))
  then send(m);
       funds := !funds - m;
       assert(not(!funds < 0))
  else ()};
int funds := 100;
```

The file is to be checked with bounds $k \leq 2$ and $l \leq 1$, as defined in its header. We can execute this file in HOLiK using the 𝕂 command:

```
krun reentrancy.holi
```

this would symbolically execute one path in the computation tree of `reentrancy.holi`, and output the final configuration reached along that path. To explore the entire computation tree, we can use the `--search` option. We can combine this with the `--pattern` option to filter the results and produce a more legible output. For instance:

```
krun --search --pattern "<trace> T </trace>" reentrancy.holi
```

produces the following list of 8 traces:

```
 T ==K ( call withdraw _1 ~> call send _1 ~> call withdraw _5
 ~> call send _5 ~> call withdraw _11 )
#Or
 T ==K ( call withdraw _1 ~> call send _1 ~> call withdraw _5
 ~> call send _5 ~> ret send () ~> ret withdraw () )
#Or
 T ==K ( call withdraw _1 ~> call send _1 ~> call withdraw _5
 ~> call send _5 ~> ret send () ~> ret withdraw ()
 ~> ret send () )
```

```
#Or
 T ==K ( call withdraw _1 ~> call send _1 ~> call withdraw _5
 ~> call send _5 ~> ret send () ~> ret withdraw ()
 ~> ret send () ~> ret withdraw () )
#Or
 T ==K ( call withdraw _1 ~> call send _1 ~> call withdraw _5
 ~> ret withdraw () )
#Or
 T ==K ( call withdraw _1 ~> call send _1 ~> call withdraw _5
 ~> ret withdraw () ~> ret send () ~> ret withdraw () )
#Or
 T ==K ( call withdraw _1 ~> call send _1 ~> ret send ()
 ~> ret withdraw () )
#Or
 T ==K ( call withdraw _1 ~> ret withdraw () )
```

By additionally filtering the configurations by whether an error was reached (implemented as a special term fail), we find all traces that violate an assertion:

```
$ krun --search --pattern "<k>fail</k><trace>T</trace>"
reentrancy.holi
T ==K ( call withdraw _1 ~> call send _1 ~> call withdraw _5
        ~> call send _5 ~> ret send () ~> ret withdraw ()
        ~> ret send () )
```

which corresponds to the example trace provided in Chapter 2:

$$\begin{aligned}&\mathit{call}\langle \mathit{withdraw}, x_1\rangle \cdot \mathit{call}\langle \mathit{send}, x_1\rangle \cdot \mathit{call}\langle \mathit{withdraw}, x_5\rangle\\ &\quad \cdot \mathit{call}\langle \mathit{send}, x_5\rangle \cdot \mathit{ret}\langle \mathit{send}, ()\rangle \cdot \mathit{ret}\langle \mathit{withdraw}, ()\rangle \cdot \mathit{ret}\langle \mathit{send}, ()\rangle\end{aligned}$$

where $x_1$ and $x_5$ are opponent values, with $x_5$ being the value used for the reentrant call. The path condition and symbolic environment generated for this trace contain 5 and 11 clauses respectively, for a total of 16 clauses over 14 variables.

$\Diamond$

## 4.9 Experiments

We wrote and adapted examples of coding errors into a set of 70 sample libraries written in HOLi, totalling 6,510 lines of code (LoC). Examples adapted from literature include: reentrancy bugs from smart contracts [8, 51]; variations of the “awkward example” [65]; various programs from the MoCHi benchmark [45]; and simple implementations related to concurrent programming (e.g. flat combining and race conditions) where errors may occur in a single thread due to higher-order behaviour. We also combined several libraries, by concatenating refactored method and reference definitions, to generate larger libraries

| | $l \leq 1$ | $l \leq 2$ | $l \leq 3$ |
|---|---|---|---|
| $k \leq 2$ | 226/70/45 (555s) | 5708/60/44 (4710s) | 9656/3/23 (12471s) |
| $k \leq 3$ | 1254/67/51 (1475s) | 4092/27/18 (13482s) | 4187/17/12 (16649s) |
| $k \leq 4$ | 3392/63/48 (3180s) | 3069/19/14 (15903s) | 1335/12/10 (17765s) |
| $k \leq 5$ | 3659/57/45 (4787s) | 895/15/10 (16757s) | 215/11/9 (17796s) |

$a/b/c$ ($d$) for $a$ traces found in $b$ successful runs taking $d$ seconds in total where $c$ out of 59 unsafe files were found to have bugs, per bound.
59 of 59 unsafe files found to have bugs over the various bounds checked

Table 4.1: Table recording performance of HOLiK on our benchmarks

that are harder to solve. In our benchmark, a filename `X-mo` describes a program `X` that has been extended, via simple concatenation, with all the method definitions inside our selection of MoCHi programs. These files are approximately 150 LoC each. The benchmark also contains filenames that include the term "`various`", which is used to label libraries built by combining various other files, with the largest being a combination of all files in the benchmark. Combined files range from 150 to 520 LoC.

We ran HOLiK on all sample libraries, lexicographically increasing the bounds from $k \leq 2, l \leq 1$ to $k \leq 5, l \leq 3$ (totalling 78,120 LoC checked), with a timeout set to five minutes per library. We start from $k \leq 2$ because it provides the minimum nesting needed to observe higher-order semantics. All experiments ran on an Ubuntu 19.04 machine with 16GB RAM, Intel Core i7 3.40GHz CPU, with intermediate calls to Z3 to prune invalid configurations. Per file, the number of error traces found and time taken can be seen in Figure 4.4. More compactly, per bound pair, the number of counterexamples found, the time taken in seconds, and the execution status (i.e. whether it terminated or not) is recorded in Table 4.1. Finally, Figure 4.5 records the frequency of errors per file size measured in lines of code.

### 4.9.1 Results and Evaluation

In Figure 4.4, we observe the raw data recorded from the experiments. The graphs show that bug-finding is, expectedly, very dependent on the nature of the specific program being checked, as seen by the wide distribution of errors found. However, a pattern can still be observed with both lower and higher $k, l$ bound pairs reporting errors more sparsely. This can explained by lower bounds not finding bugs existing deeper in the execution tree, as well as analyses not terminating when attempting to search too deeply. Table 4.1, which more succinctly expresses the results, shows this behaviour more clearly. We can observe that independently increasing the bounds for $k$ and $l$ causes exponential growth in the total time taken, which is expected from symbolic execution. Note that the time tends towards 21000 seconds because of the timeout set to 5 minutes for 70 programs. Similarly, the number of errors found initially grows exponentially with respect to the

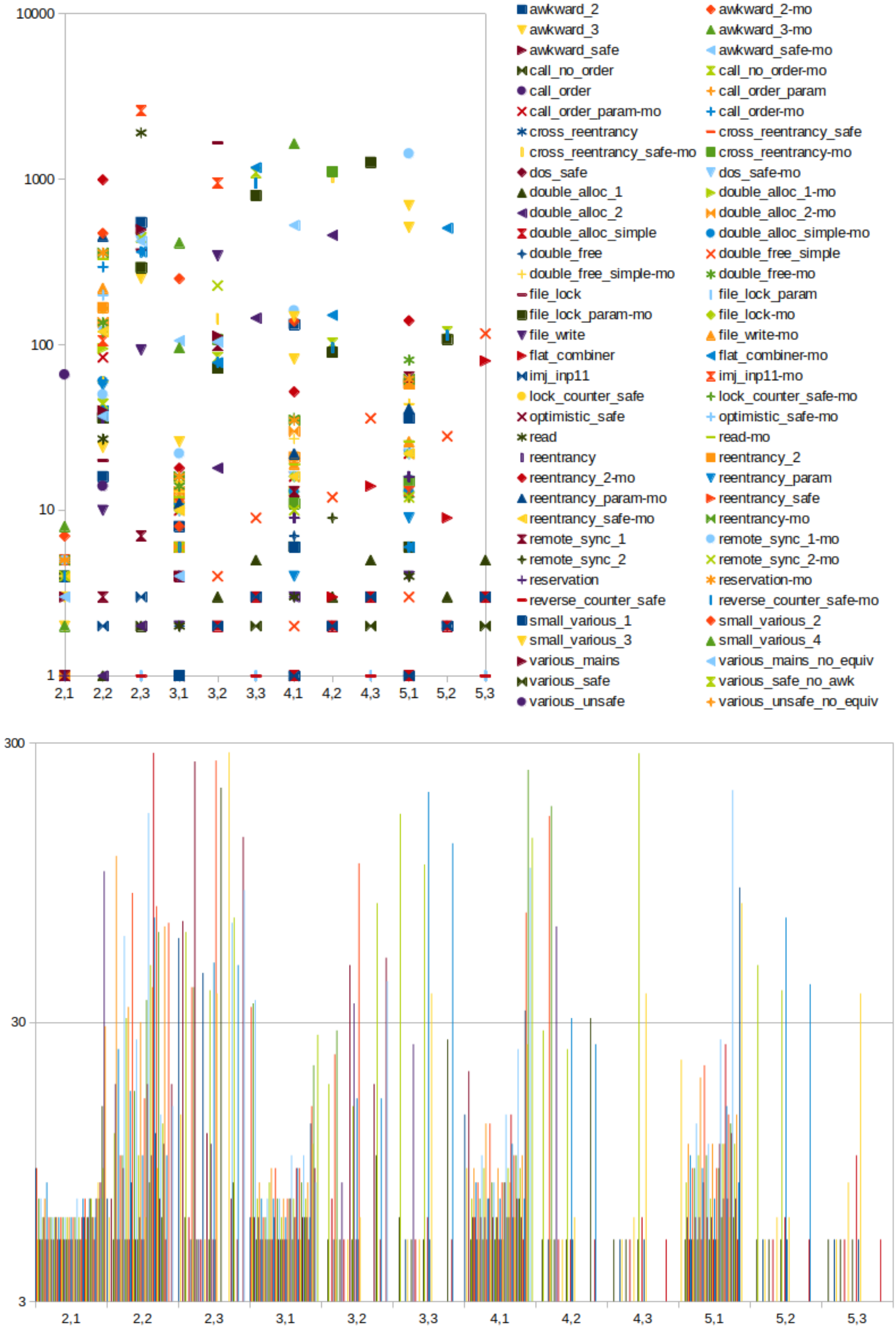


Figure 4.4: Errors (top) and time(s) (bottom) per file per $k, l$-bound in HOLiK

increase in bounds, which can be explained by the exponential growth in paths, but this trend does not continue indefinitely as the tool starts timing out without reporting any errors as the bounds grow. For the benchmark as a whole, with bounds $k \leq 2$ and $l \leq 1$, all 70 files in our benchmark were successfully analysed, meaning, all analyses terminated. Errors were found at different bounds for different files, which was expected seen as some programs had deeper bugs by design. Cumulatively, all unsafe programs in our benchmark were correctly identified. While the table may suggest that increasing the bound for $l$ is more beneficial than that for $k$, the number of errors reported does not imply every trace is useful. For instance, increasing the bound for $l$ can lead to errors re-merging in a higher-order version, which suggests potential gain from a partial order reduction. In general, the $k$ and $l$ counters are incomparable as they keep track of different behaviours.

As for scalability in program size rather than depth of search, we see in Figure 4.5 that, for our benchmark, the frequency of errors found in programs increases with the size of the program. We can thus tell that our tool is able to scale up to programs over 500 LoC, but the number of errors reported starts to plateau at this point. This is mainly due to the fact bigger programs cannot be searched as deeply as smaller programs can be, so, while more errors may exist in larger programs, less terminating deeper analyses are recorded. The tool appears to be most comfortable with programs between 100 and 400 LoC, as it records the widest range of errors, but this can be misleading as there were not many files over 500 LoC (<1000 LoC) compared to files in the 100 to 500 LoC range, which contains the bulk of benchmarks. The highest average for errors reported were in the 400 to 500 LoC range. Again, these results depend on the nature of the programs in our benchmark, and some files between 100 and 400 LoC were found to contain more errors than the larger programs. Bounded symbolic execution, however, is also theoretically less scalable in terms of compilation speed than other bounded techniques (such as BMC discussed in the previous chapter) and thus, it may so be an expected result to see scalability of SE drop off in terms of program size.

Considering theoretical results, and, experimentally, since HOLiK was able to handle every file and correctly identified all unsafe files in the benchmark, we conclude that HOLiK, as a proof of concept, captures the full range of behaviours in higher-order libraries in a practical sense. Results suggest that the tool scales up to at least medium-sized programs (<1000 LoC), which is promising because real-world medium-size higher-order programs have been proven infeasible to check with standard techniques (e.g. the DAO withdraw contract was approximately 100 LoC, but required the development of techniques specialised on reentrancy to check). Even if performance for larger programs appears to drop off, as a bounded technique, our tool still comfortably analysed all the large programs when given smaller bounds to work with. For instance, for $k, l \leq 2, 1$ the tool analysed all programs in less than 5 minutes each, which suggests the potential to check much larger programs if allowed to run for longer. Taking into consideration that $k, l \leq 2, 1$ is the minimal depth required to find intricate higher-order bugs, i.e. it is

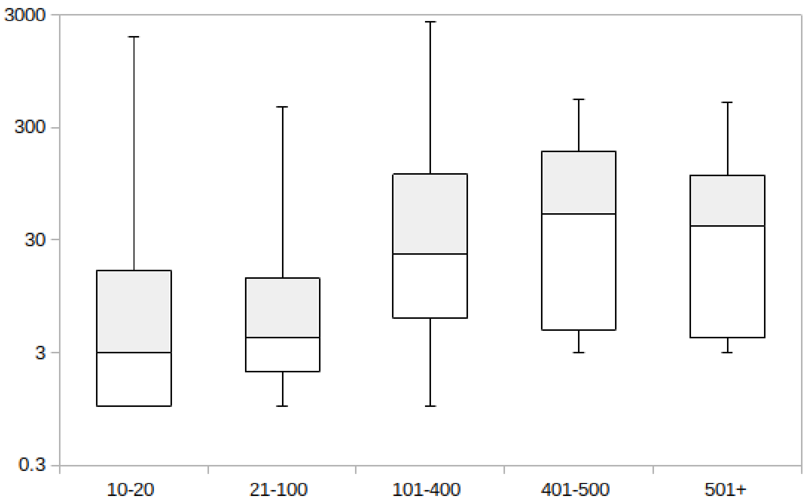


Figure 4.5: Error distribution vs ranges of file size (LoC) in HOLiK

sufficient to find higher-order interaction with the environment, the technique appears promising even on larger programs, as it is at least able to scan said large programs for superficial higher-order errors, which, as already mentioned, have been proven infeasible to check with standard techniques. Moreover, unlike tools that specialise on particular higher-order behaviours, our semantics models the complete range of higher-order errors definable by a client.

### 4.9.2 Comparison with Racket Contract Verification

In Section 1.5, we mentioned [58] and [57] as tools related to higher-order verification. In this section we compare our games-based approach to those in the area of Racket contract verification. Firstly, since [58] does not handle state, we shall only consider the latest version of the tool [57], which we shall refer to as SCV (for Soft Contract Verification). A small benchmark (19 programs based on the HOLiK and SCV benchmarks) was used for the comparison. Programs were manually translated between Racket and HOLi. Care was taken to translate programs while keeping the semantics as close as possible. However, since assertions and contracts are fundamentally different, some decisions had to be made about the terms to choose for the different features in contracts, and vice versa. For instance, contracts enforcing an input-output relation were translated into HOLi using wrapper functions that define the relation through an if statement. e.g.

```
(provide/contract
  [f (→i [n (and/c integer? (>/c 0))]
          (res (n) (and/c integer? (>/c n))))])
(define (f n) (...))
```

would be translated into

```
public fcontract (n:int) :(unit) = {
  if (n > 0) then assert(f n > n) else ()
};
private f (n:int) :(int) = {...};
```

In the opposite direction, since contracts do not directly access references in a term, stateful functions were translated to return any references we wish to reason about.

Table 4.2 records the comparison. The most obvious difference is in bug reporting and safety; which makes them incomparable in general. While HOLiK reported traces that always correspond to real errors, SCV reported several spurious counter examples—a third of all errors reported by SCV were spurious. Spurious errors were manually determined by whether the contract referenced is possible to violate or not. On the other hand, SCV was able to prove total correctness for 3 of the 7 programs designed to be bug free. The next obvious difference is that SCV scales much better than HOLiK with respect to program size, which it achieves in exchange for the loss in precision incurred in finitising the semantics. While the difference in time for small programs is mainly due to initalisation time, HOLiK would take close to an order of magnitude longer for some larger programs. The programs that took the longest for HOLiK (`dao-various` and `dao2-e`) consist of multiple repeated methods, which suggests a potential optimisation in partial order reduction. More subtle differences in the nature of each approach can also be observed in the results recorded. e.g., HOLiK reports 1 real error for `ack-simple-e`, whereas SCV reports 2 errors. The difference is because SCV takes into account constraints for integers (e.g. $> 0$ and $= 0$). More interestingly, for `various`, HOLiK reports 19 ways to reach assertion violations, whereas SCV reports only 6 real ways to violate contracts. The difference is because HOLiK reports paths through the execution tree that violate assertions, whereas SCV reports a set of term skeletons that may violate contracts. To illustrate this difference, take for instance two independently safe methods $A$ and $B$ that may call an unsafe method $C$. From testing, HOLiK reports three valid traces as there are three paths that reach $C$ ($call\langle A\rangle \cdot call\langle C\rangle$, $call\langle B\rangle \cdot call\langle C\rangle$ and $call\langle C\rangle$). In contrast, SCV reports a single contract violation directly blaming $C$ as the only error. This could be because SCV groups these paths into a single term description, whereas we explicitly list them. Finally, `ack` failed to run on SCV due to internal errors. Since `ack` is valid and can be executed manually, we cannot explain this result.

Overall, HOLiK is generally slower than SCV and appears to scale worse in terms of program size. This is expected of SCV since it benefits from a finitised abstract interpretation of its semantics. However, the trade-off in precision can be easily observed from the fact SCV is not able to distinguish between the safe and unsafe versions of the DAO example. In our own attempts at finitising our symbolic games, which we discuss as a potential further direction in Chapter 5, abstract interpretation also caused our games to become unable to distinguish the safe DAO from the unsafe one, which we deemed too much of a loss in precision. Moreover, as mentioned previously, bounding the semantics still allows games to scale for much larger programs, while still reporting only

| Program | LoC | Traces | Time (s) | LoC | Errors | Time (s) | False Errors |
|---|---|---|---|---|---|---|---|
| ack | 17 | 0 | 6.0 | 9 | N/A | 2.4 | N/A |
| ack-simple | 13 | 0 | 6.5 | 9 | 0 | 2.4 | 0 |
| ack-simple-e | 13 | 1 | 6.5 | 9 | 2 | 2.5 | 0 |
| dao | 10 | 0 | 5.0 | 15 | 1 | 2.6 | 1 |
| dao-e | 16 | 1 | 5.5 | 15 | 1 | 2.7 | 0 |
| dao-various | 85 | 5 | 22.5 | 122 | 10 | 3.0 | 5 |
| dao2-e | 85 | 10 | 23.5 | 122 | 10 | 2.9 | 0 |
| escape | 9 | 0 | 5.0 | 9 | 0 | 2.6 | 0 |
| escape-e | 9 | 2 | 5.0 | 10 | 1 | 2.7 | 0 |
| escape2-e | 10 | 14 | 6.0 | 10 | 1 | 2.7 | 0 |
| factorial | 10 | 0 | 5.0 | 9 | 0 | 2.2 | 0 |
| mc91 | 12 | 0 | 5.0 | 9 | 1 | 2.2 | 1 |
| mc91-e | 12 | 1 | 5.0 | 8 | 1 | 2.4 | 0 |
| mult | 14 | 0 | 5.0 | 11 | 2 | 2.7 | 2 |
| mult-e | 14 | 1 | 5.0 | 11 | 2 | 2.4 | 0 |
| succ | 7 | 0 | 5.0 | 7 | 1 | 2.5 | 1 |
| succ-e | 7 | 1 | 5.0 | 7 | 1 | 2.8 | 0 |
| various | 116 | 19 | 14.0 | 108 | 11 | 6.2 | 5 |
| total | 459 | 55 | 140.5 | 500 | 45 | 49.8 | 15 |

Table 4.2: Comparison of HOLiK (left) and SCV (right).

sound errors. This is, in addition to the fact that there is still room for optimisation since the approach presented herein is a foundational and has not been optimised, suggests the technique could find use in verifying real-world software. We thus conclude that, in terms of verification of higher-order programs, our games-based approach is a viable alternative to the line of work presented in soft contract verification.

# Chapter 5

# Conclusions

We conclude this thesis with a summary of the contributions and results presented herein, and a discussion of potential future work. We shall present concluding statements in two parts, one for each content chapter of the thesis, starting with Chapter 3.

## 5.1 BMC for Closed Higher-Order Programs

In Chapter 3 we presented a BMC approach for closed higher-order programs inspired by the widespread success of BMC in industry. The approach presented is based on a nominal interpretation of defunctionalisation at the semantics level, and a points-to analysis to optimise the flow of methods. Preliminary results were provided for the performance of our approach in a prototype tool named BMC-2, as well as a proof of soundness of the algorithm.

From testing, we concluded that our BMC technique is a practically viable alternative to existing approaches for higher-order (closed) terms when compared to HORS model checking (e.g. MoCHi) and higher-order SE (e.g. Rosette) in particular. For HORS model checking, we provide a bug-finding alternative to total correctness that is sound for errors. Compared to SE, results suggest that the theoretical speed-up in compilation, at the expense of a larger monolithic SAT instance, can be beneficial for large programs, even if comparatively detrimental for depth of analysis. Specifically, BMC-2 scales better than the internal implementation of SE in Rosette for program size when initialisation is considered. Large higher-order programs may find use in our approach as its bounded nature could make it feasible to scan these for superficial errors. We also briefly discussed a limitation of state merging in our approach due to the nature of the problem at hand. Specifically, a higher-order semantics that allows dynamic creation of methods does not have a bounded branching factor, and thus does not scale linearly with the bound as mainstream BMC might. While this partially detracts from the advantages in memory efficiency, its performance on larger programs show BMC is still worth investigating.

## 5.2 Symbolic Games for Open Higher-Order Programs

While able to handle internal higher-order behaviour, the monolithic nature of our BMC approach hinders its ability to handle external higher-order interaction in general, which requires a compositional reasoning. To address this, and practically motivated by the prominence of higher-order errors in real-world programs, in Chapter 4 we presented a game semantics to model the behaviour of an undefined environment and adapted it to symbolically execute open higher-order libraries. We implemented a prototype symbolic execution tool for higher-order libraries named HOLiK as a proof of concept. Our approach was also supported with theoretical results that prove soundness and completeness of our game semantics, which makes use of compositionality and definability results; prove that termination can be enforced by bounding the nesting of calls and *chattering* of the opponent; and prove that the resulting symbolic execution technique is sound with respect to errors reported.

Our experiments show that our technique has a practical application in soundly scanning programs for errors. In comparison to most other lines of work, our approach is distinguished by the ability to cope with the full range of higher-order interaction allowed by an undefined environment, whereas most extant tools focus on fragments of the behaviours we are interested in. In comparison to the line of work presented in [57], which does handle the complete range of higher-order behaviour, we found that game semantics is a different theoretical foundation that produced a viable alternative approach to analysis of higher-order programs. We showed [57] scales better in terms of program size, but sacrifices precision, whereas HOLiK always finds concrete errors in a bounded-complete manner. Fundamentally, the approaches have different motivations (bug-finding vs total correctness), and are thus incomparable. In their overlapping task, however, both tools can produce counterexamples, which, as stated, mainly differs in their soundness and scalability. Considering that the theory presented is foundational only and has not been optimised in its implementation, on top of the scalability added by the parametrisation of depth of search, we concluded that our games approach is practically promising and serves as a novel foundation for symbolic execution of open higher-order programs.

## 5.3 Limitations and Further Directions

In this section we primarily discuss possible future work for both content chapters of the thesis. We shall dedicate a section for each prospective endeavour for the content presented herein. We start with that for our BMC technique, and continue with those for our SE game semantics.

### 5.3.1 Further Developing and Optimising BMC-2

We begin this section with a discussion for our BMC technique. In particular, the experiments we ran on our tool show that BMC is a practically feasible direction to check higher-order programs. This agrees with the intuition that BMC is a practically successful technique at an industrial level. However, the tool we produced was a very preliminary proof of concept. There are various limitations for BMC-2 as a tool, which we shall discuss in the following paragraphs.

Firstly, HORef is a toy language for academic purposes as it is a universal stateful language that includes a minimal set of features required to exhibit higher-order behaviour internal to the term. Theoretically, selecting a specific industry language may present many complications in features that are not of interest. Implementationally, it makes developing a prototype under limited resources feasible. Practically, however, the tool would need a translation from source code into HORef in order to be useful. HORef could serve as an intermediate language due to its universality, but being fundamentally a simple low-level extension of lambda calculus makes impractical to translate the range of features seen in real-world languages. As such, a possible direction is to implement our approach for one such real-world language. Particularly, as it is, HORef already serves as a subset of features for ML-like languages, and could even be almost directly implemented for C. These would require extensions of the theory to cover features such as pattern matching for functional languages and pointer arithmetic for C. Of particular interest may be object oriented features. The current nominal model for higher-order terms could potentially be used to reason about objects as values.

Secondly, although some effort was made in optimising the translation, particularly with regards to flow of names to prune spurious paths, this was done primarily for overall feasibility rather than as a fine-tuned optimisation. As is, the encoding could be further optimised for size and complexity of the formula produced, for instance, by merging states more optimally. Our translation currently adds a so-called $\Phi$ function after each branch, meaning that states are always merged wherever merging is possible. While this keeps memory usage low in mainstream BMC, this has two severe drawbacks that affect the complexity of our approach. Firstly, total state merging means that symbolic names (i.e. those appearing in variables) are never concretised. This means that names, wherever encountered, are more likely to cause branching as each corresponds to the largest possible set of names that may flow into any given term. This leads to more spurious paths explored, which takes longer to compile and produces formulas that are harder to solve. Our points-to analysis tackles this issue by reducing the number of names to consider. However, concretisation of names is still desirable as it involves less operations than data flow of a single name. Secondly, state merging is an expensive operation, particularly where references are involved. When a $\Phi$ function is inserted, all references in every branch need to be merged. This is an expensive operation that adds to the formula at least $n$ many clauses, where $n$ is the number of references in the term. Additionally, the $\Phi$ function requires a guard for each path merged within it. This adds, for every branch, a disjunction nested in the clauses (i.e. a nested implication) that makes the formula harder to solve. As such, minimising the number of $\Phi$ functions whilst ensuring static single assignment is desirable.

As mentioned in Chapter 1, state merging in BMC is a result of the SSA transformation. As such, the problem of efficient state merging has already been studied in optimising SSA transformations. A general solution to inserting $\Phi$ functions efficiently has been provided using the concept of "minimal" SSA with *dominance frontiers*. These tell us precisely where $\Phi$ functions should be inserted to maintain the SSA form. Moreover, with liveness analysis and dead-code elimination, one can construct a so-called "pruned" SSA form [19], which tackles the problem of having to merge every reference appearing in the term by pruning all references that become unreachable from any point onwards. The added overhead in computing the information required for optimal $\Phi$ function placement is non-trivial, however, and a parameter to specify the level of precision needed is often useful. For instance, a "semi-pruned" SSA form [14] approximates the effect of a fully "pruned" SSA without the expense of full liveness analysis. Instead, it computes the relatively cheap set of "block-local"—meaning variables that are never live outside a block—that require no $\Phi$ function.

### 5.3.2 Theoretical and Practical Directions for Symbolic Games

In this section we discuss different prospective developments for our symbolic games. The future work presented in this sectional shall be divided into two directions: optimisations further implementational directions for the current theory, and theoretical developments in the form of full compositionality and total verification.

#### 5.3.2.1 Practical Developments and Optimisation

According to the experiments, most of the errors reported when increasing the depth of analysis were replicas of shallow errors resurfacing deeper in the trace. These occurred in the form of lower order errors appearing nested within some trace of higher order, and, as such, suggest the possibility of defining a partial order for our semantics. A notion of partial order could define configurations that subsume the behaviour of lower order behaviours already observed, and thus eliminate paths that involve known errors [63, 80].

Additionally, from our theoretical results we observed that a lot of effort was placed in carefully manipulating $k$ and $l$ to prove soundness. Moreover, attempting to show the complexity of our games in terms of $k$ and $l$ proved to be a harder task than expected. With the addition of $k$ and $l$ not being intuitively useful when attempting to understand the practical impact of increasing the bound with respect to verification feasibility, it seems that our bounding mechanism as it currently is, while sensible as a direct bound to sources of infinite behaviour, is more of an arbitrary choice with respect to the overarching goal of verifying programs. Considering that the theory does not prevent the generalisation of $k$ and $l$ as a monotonic cost function, allowing instead the option of a user-specified bounding mechanism that satisfies the property of monotonicity could provide versatility for the user to define a mechanism that most intuitively satisfies their expected behaviour. For instance, while $k$ is intended to bound method calls, it only bounds proponent calls—i.e. the bound is not symmetric, which makes it more difficult to reason about the semantics both theoretically and intuitively. Another example of this disadvantage can be seen by how different programs may differ immensely in feasibility even when given the same bounds; one may desire a more intuitive bounding strategy where bounds determine how the analysis is expected to behave. Additionally, a generalisation of the bounds could improve the practical feasibility of our technique by more precisely filtering the state-space. The idea is that, similarly to a search heuristic, specifying a generalised bound could have the effect of guiding the search towards certain patterns of errors—or guarantee the lack thereof—that would otherwise be difficult to find. For instance, the minimal error trace for the “flat combiner" example in Chapter 2 requires a minimum call depth of $k = 4$ and for the opponent to be able to make at least $l = 2$ consecutive calls at each layer. However, the error is caused by a specific pattern that has an initial preference for shallow moves that are followed by a series of calls that increases the depth of the game. Finding this specific pattern among the entire computation tree of depth $k = 4$ and environment size $l = 2$ could be difficult.

Another issue observed is that the language supported is again a toy language. The choice to use HOLi as our vehicle of study was made for the sake of developing a theoretical foundation. Support for a real-world language would have required a substantial amount of effort in the formalisation of specific language features that are not relevant to the theory of stateful higher-order verification. Moreover, doing so may have still required the translation of programs from one language to another, especially when taking into consideration that our benchmarks were obtained from various languages. However, it remains a limitation that all our benchmarks had to be translated into HOLi, which involves an additional step where the semantics of the source program and the target program may deviate, which is exacerbated by the low-level nature of HOLi. For instance, while our DAO example captures the essence of the reentrant attack, as would any simplification used for illustration, the actual attack was additionally a result of the semantics of Solidity, which would have obfuscated the fact that an external function could be called within the withdraw contract. Additionally, the semantics of the Ethereum virtual machine has not been taken into account in the naive simplification. Again, our focus was never smart contract verification, but this leads to a forced simplification of some benchmarks, since properties such as resource consumption when making calls were not modelled in the benchmarks inspired by smart contracts. Lastly, the use of a toy language also imposes a limitation in the size, relevance and eloquence of programs we are allowed to examine. In particular, translating real-world features into HOLi is not trivial as it involves reconciliation between different operational semantics, and thus prevents us from using larger real-world representatives in our benchmarks. For these reasons, a possible development is to formalise a real-world language, or extend an existing formalisation or SE tool with our game-semantic theory.

Finally, before moving on with theoretical developments, our experiments revealed a potential problem in the architecture of our implementation. Namely, our semantics was implemented in $\mathbb{K}$ as it makes the implementing formal rules a relatively simple process. However, we noticed potential issues in reliability and performance as a result of depending on $\mathbb{K}$. During development, we encountered an issue with HOLiK after revision 01ba5140[1] (July 30, 2019) of the $\mathbb{K}$ platform-independent binaries, after which the interaction between $\mathbb{K}$ and Z3 became unreliable. The update changed pattern matching, and caused $\mathbb{K}$ to not recognise some responses from the solver. This was eventually fixed, but dependence on external tools could affect long-term support and development of the tool. More importantly, several limitations were imposed by using $\mathbb{K}$. For instance, we did not have control over the search strategy, and lacked features necessary to implement certain optimisations (e.g. keeping a set of explored configurations across all paths is not possible). Worse still, the current plan is to deprecate the Java backend which we depend on. Currently, the other backends parse definitions slightly differently and do not feature the options we need for our symbolic execution—from communication with the developers, we learned that they have a different direction in mind for symbolic execution and support for implementation of functional languages is not a priority. The eventual lack of support may be a problem for continued development of a reliable tool. Putting reliability aside, a potential issue in the performance of $\mathbb{K}$ was also observed in its performance; for instance, long initialisation times. While the complexity of our semantics is the primary factor in the performance observed, results suggest that some amount of time is added to all executions simply by virtue of using $\mathbb{K}$. Considering these limitations, a native implementation of our semantics is desirable. This could, as in the previous paragraph, be by extension of an existing SE tool, or built from scratch.

#### 5.3.2.2 Full Compositionality and Total Verification

The first theoretical direction is that of full compositionality. The current semantics only allows for compositionality between two components. Compositionality between $n$-many components (full compositionality) would allow for fully modular verification. Given the compositional design of games, it is possible full compositionality is already a property of our games that just needs to be proven. For our purpose, a two-part compositionality was sufficient for verification. However, a full compositionality result may lead to a fully compositional technique. More intuitively, as it is, our semantics guarantees that composing the result of independently analysing two components (a library and a client) is equivalent to analysing an already fully composed system. This has the useful property of allowing one to either analyse each component in isolation

[1] `https://github.com/kframework/k/releases/tag/nightly-01ba514c0`

first to then modularly analyse various compatible clients for correctness, or, by splitting any program into two components with disjoint name sets, allowing one to decompose the analysis of programs into smaller instances. The latter is of particular interest in the field of modular verification, as it allows tools like BLITZ [18] to split programs into smaller components at various levels of granularity. This is useful practice especially when verifying large programs that would otherwise not fit in memory. As such, the development of a theoretical result for full compositionality may indicate to us a method to use games for modular analysis that decomposes programs into arbitrarily many libraries, also adjustable for a specified level of granularity.

Finally, to finish this section we shall discuss the possibility of removing the bounds entirely for unbounded verification. A limitation of our technique when compared to [57] is that we are not sound when the termination cannot be determined within the bound. This means we are generally unable to show total correctness of programs, which is theoretically interesting and practically desirable. For this, we looked into three possible directions to finitise the game semantics.

**Abstract Interpretation** The first is to go for a standard abstract interpretation technique [22]. The plan would be to transform our bounded semantics to an abstract interpretation semantics by using abstract domains in place of symbolic values. A bounded execution of this new semantics would produce a chain of abstract values for each assertion in the source code, which could then be used to produce a fixpoint for the range of possible values that may occur in each assertion. While experimenting with preliminary definitions of possible abstract semantics, however, we noticed it incurred a substantial loss in precision, which we deemed unacceptable. Particularly, being able to differentiate between safe programs and unsafe programs—e.g. the safe and unsafe DAO—turned out to require a substantial amount of additional infrastructure.

**Fresh-Register Pushdown Automata** Another direction, closely related to Coneqct [53], is to define a pushdown system that finitely captures the unbounded game semantics of any program we want to verify. The approach in [53] is based on the decidability of reachability in fresh-register pushdown automata (FPDRA)—even over infinite alphabets. This requires, however, a finite branching factor for the semantics, which in our case amounts to overapproximating methods and integers. As with abstract interpretation, this may require defining an abstract domain for integers to enforce finite branching. Methods could be overapproximated using a finite set of names as done in $k$-CFA [75]. Assuming that sufficient precision is maintained, the approach would proceed by building an FPDRA that captures the unbounded game semantics of a target program, to then apply an existing decision procedure [54] for reachability of errors. As a result of thr overapproximation, unreachability of errors in the automaton would imply errors are similarly unreachable in the unbounded computation tree.

**Counter-Example Guided Refinement** Lastly, from MoCHi [45] we know that it is possible to use a CEGAR loop in tandem with higher-order recursion schemes for unbounded verification of higher-order programs. The main disadvantage is that the MoCHi approach does not handle state or undefined functions. From private communication with the authors, we found that extensions to MoCHi may provide limited support for state, but would not handle open code. An extension of MoCHi that combines it with our games to support open code is not obvious, but remains a potential direction to follow to produce a total correctness technique.

# Bibliography

[1] S. Abramsky, D. R. Ghica, L. Ong, and A. Murawski. Algorithmic game semantics and component-based verification. In *Proceedings of SAVBCS 2003: Specification and Verification of Component-Based Systems, Workshop at ESEC/FASE 2003*, pages 66–74, 2003. published as Technical Report 03-11, Department of Computer Science, Iowa State University.

[2] Samson Abramsky. Semantics of interaction (abstract). In Hélène Kirchner, editor, *Trees in Algebra and Programming - CAAP'96, 21st International Colloquium, Linköping, Sweden, April, 22-24, 1996, Proceedings*, volume 1059 of *Lecture Notes in Computer Science*, page 1. Springer, 1996.

[3] Samson Abramsky, Kohei Honda, and Guy McCusker. A fully abstract game semantics for general references. In *Thirteenth Annual IEEE Symposium on Logic in Computer Science, Indianapolis, Indiana, USA, June 21-24, 1998*, pages 334–344. IEEE Computer Society, 1998.

[4] Samson Abramsky, Pasquale Malacaria, and Radha Jagadeesan. Full abstraction for PCF. In Masami Hagiya and John C. Mitchell, editors, *Theoretical Aspects of Computer Software, International Conference TACS '94, Sendai, Japan, April 19-22, 1994, Proceedings*, volume 789 of *Lecture Notes in Computer Science*, pages 1–15. Springer, 1994.

[5] Samson Abramsky and Guy McCusker. Game semantics. In Ulrich Berger and Helmut Schwichtenberg, editors, *Computational Logic*, pages 1–55, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[6] Nina Amla, Robert P. Kurshan, Kenneth L. McMillan, and Ricardo Medel. Experimental analysis of different techniques for bounded model checking. In Hubert Garavel and John Hatcliff, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 9th International Conference, TACAS 2003, Proceedings*, volume 2619 of *Lecture Notes in Computer Science*, pages 34–48. Springer, 2003.

[7] L. O. Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts sok. In *Proceedings of the 6th International Conference on Principles of Security and Trust - Volume 10204*, pages 164–186, New York, NY, USA, 2017. Springer-Verlag New York, Inc.

[9] Christel Baier and Joost-Pieter Katoen. *Principles of model checking*. MIT Press, 2008.

[10] Adam Bakewell and Dan R. Ghica. Game-based safety checking with mage. In Arnd Poetzsch-Heffter, editor, *Proceedings of the 2007 Conference Specification and Verification of Component-Based Systems, SAVCBS 2007, Dubrovnik, Croatia, September 3-4, 2007*, pages 85–87. ACM, 2007.

[11] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[12] Robert Boyer, Bernard Elspas, and Karl Levitt. SelectâĂŤa formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10:234–245, 06 1975.

[13] Robert S. Boyer, Bernard Elspas, and Karl Levitt. SELECTâĂŤa formal system for testing and debugging programs by symbolic execution. *ACM SIGPLAN Notices*, 10:234–245, 06 1975.

[14] Preston Briggs, Keith D. Cooper, Timothy J. Harvey, and L. Taylor Simpson. Practical improvements to the construction and destruction of static single assignment form. *Softw., Pract. Exper.*, 28(8):859–881, 1998.

[15] Toby Cathcart Burn, C.-H. Luke Ong, and Steven J. Ramsay. Higher-order constrained horn clauses for verification. *PACMPL*, 2(POPL):11:1–11:28, 2018.

[16] Cristian Cadar, Daniel Dunbar, and Dawson Engler. Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*, OSDI'08, pages 209–224, Berkeley, CA, USA, 2008. USENIX Association.

[17] Cristian Cadar and Koushik Sen. Symbolic execution for software testing: three decades later. *Commun. ACM*, 56(2):82–90, 2013.

[18] Chia Yuan Cho, Vijay D'Silva, and Dawn Song. BLITZ: compositional bounded model checking for real-world programs. In Ewen Denney, Tevfik Bultan, and Andreas Zeller, editors, *2013 28th IEEE/ACM International Conference on Automated Software Engineering, ASE 2013, Silicon Valley, CA, USA, November 11-15, 2013*, pages 136–146. IEEE, 2013.

[19] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In David S. Wise, editor, *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages, Orlando, Florida, USA, January 21-23, 1991*, pages 55–66. ACM Press, 1991.

[20] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In Kurt Jensen and Andreas Podelski, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 10th International Conference, TACAS 2004, Proceedings*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176. Springer, 2004.

[21] Lori A. Clarke. A program testing system. In John A. Gosden and Olin G. Johnson, editors, *Proceedings of the 1976 Annual Conference, Houston, Texas, USA, October 20-22, 1976*, pages 488–491. ACM, 1976.

[22] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In Robert M. Graham, Michael A. Harrison, and Ravi Sethi, editors, *Conference Record of the Fourth ACM Symposium on Principles of Programming Languages, Los Angeles, California, USA, January 1977*, pages 238–252. ACM, 1977.

[23] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[24] Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. Modular verification of code with SAT. In Lori L. Pollock and Mauro Pezzè, editors, *Proceedings of the ACM/SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2006, Portland, Maine, USA, July 17-20, 2006*, pages 109–120. ACM, 2006.

[25] Aleksandar S. Dimovski. Program verification using symbolic game semantics. *Theor. Comput. Sci.*, 560:364–379, 2014.

[26] Julian Dolby, Mandana Vaziri, and Frank Tip. Finding bugs efficiently with a SAT solver. In Ivica Crnkovic and Antonia Bertolino, editors, *Proceedings of the 6th joint meeting of the European Software Engineering Conference and the ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2007, Dubrovnik, Croatia, September 3-7, 2007*, pages 195–204. ACM, 2007.

[27] Vijay D'Silva, Daniel Kroening, and Georg Weissenbacher. A survey of automated techniques for formal software verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 27(7):1165–1178, 2008.

[28] Quinn Dupont. *Experiments in Algorithmic Governance: A history and ethnography of " The DAO, " a failed Decentralized Autonomous Organization*, chapter 8. Routledge, 01 2017.

[29] Juan P. Galeotti, Nicolás Rosner, Carlos Gustavo López Pombo, and Marcelo F. Frias. TACO: efficient sat-based bounded verification using symmetry breaking and tight bounds. *IEEE Trans. Software Eng.*, 39(9):1283–1307, 2013.

[30] D. R. Ghica. Applications of game semantics: From program analysis to hardware synthesis. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*, pages 17–26, 2009.

[31] Dan R. Ghica. Applications of game semantics: From program analysis to hardware synthesis. In *Proceedings of the 24th Annual IEEE Symposium on Logic in Computer Science, LICS 2009, 11-14 August 2009, Los Angeles, CA, USA*, pages 17–26. IEEE Computer Society, 2009.

[32] Dan R. Ghica and Guy McCusker. Reasoning about idealized ALGOL using regular languages. In Ugo Montanari, José D. P. Rolim, and Emo Welzl, editors, *Automata, Languages and Programming, 27th International Colloquium, ICALP 2000, Geneva, Switzerland, July 9-15, 2000, Proceedings*, volume 1853 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 2000.

[33] Dan R. Ghica and Nikos Tzevelekos. A system-level game semantics. *Electr. Notes Theor. Comput. Sci.*, 286:191–211, 2012.

[34] Ilya Grishchenko, Matteo Maffei, and Clara Schneidewind. Foundations and tools for the static analysis of ethereum smart contracts. In Hana Chockler and Georg Weissenbacher, editors, *Computer Aided Verification - 30th International Conference, CAV 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 14-17, 2018, Proceedings, Part I*, volume 10981 of *Lecture Notes in Computer Science*, pages 51–78. Springer, 2018.

[35] Michael Hind. Pointer analysis: haven't we solved this problem yet? In John Field and Gregor Snelting, editors, *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis For Software Tools and Engineering, PASTE'01*, pages 54–61. ACM, 2001.

[36] David Hopkins, Andrzej S. Murawski, and C.-H Luke Ong. Hector: An equivalence checker for a higher-order fragment of ml. In *CAV*, 2012.

[37] David Van Horn and Matthew Might. Abstracting abstract machines. In Paul Hudak and Stephanie Weirich, editors, *Proceeding of the 15th ACM SIGPLAN international conference on Functional programming, ICFP 2010, Baltimore, Maryland, USA, September 27-29, 2010*, pages 51–62. ACM, 2010.

[38] William E. Howden. Symbolic testing and the DISSECT symbolic evaluation system. *Software Engineering, IEEE Transactions on*, SE-3:266– 278, 08 1977.

[39] John Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, 1989.

[40] J. M. E. Hyland and C.-H. Luke Ong. On full abstraction for PCF: i, ii, and III. *Inf. Comput.*, 163(2):285–408, 2000.

[41] A. Jeffrey and J. Rathke. A fully abstract may testing semantics for concurrent objects. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 101–112, July 2002.

[42] James C. King. A new approach to program testing. *SIGPLAN Not.*, 10(6):228–233, April 1975.

[43] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[44] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 416–428, New York, NY, USA, 2009. ACM.

[45] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011*, pages 222–233. ACM, 2011.

[46] Daniel Kroening. The cbmc homepage. `http://www.cprover.org/cbmc/`, 2017. [Online; accessed 13-Jun-2017].

[47] James Laird. A fully abstract trace semantics for general references. In Lars Arge, Christian Cachin, Tomasz Jurdzinski, and Andrzej Tarlecki, editors, *Automata, Languages and Programming, 34th International Colloquium, ICALP 2007, Wroclaw, Poland, July 9-13, 2007, Proceedings*, volume 4596 of *Lecture Notes in Computer Science*, pages 667–679. Springer, 2007.

[48] Yu-Yang Lin and Nikos Tzevelekos. A bounded model checking technique for higher-order programs. In Nan Guan, Joost-Pieter Katoen, and Jun Sun, editors, *Dependable Software Engineering. Theories, Tools, and Applications - 5th International Symposium, SETTA 2019, Shanghai, China, November 27-29, 2019, Proceedings*, volume 11951 of *Lecture Notes in Computer Science*, pages 1–18. Springer, 2019.

[49] Yu-Yang Lin and Nikos Tzevelekos. Symbolic execution game semantics. In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 27:1–27:24. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[50] C. Liu, H. Liu, Z. Cao, Z. Chen, B. Chen, and B. Roscoe. Reguard: Finding reentrancy bugs in smart contracts. In *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)*, pages 65–68, May 2018.

[51] Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, and Aquinas Hobor. Making smart contracts smarter. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 254–269, New York, NY, USA, 2016. ACM.

[52] Jeremy Morse, Mikhail Ramalho, Lucas C. Cordeiro, Denis Nicole, and Bernd Fischer. ESBMC 1.22 - (competition contribution). In Erika Ábrahám and Klaus Havelund, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 20th International Conference, TACAS 2014, Proceedings*, volume 8413 of *Lecture Notes in Computer Science*, pages 405–407. Springer, 2014.

[53] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. A contextual equivalence checker for IMJ*. pages 234–240, 11 2015.

[54] Andrzej S. Murawski, Steven J. Ramsay, and Nikos Tzevelekos. Reachability in pushdown register automata. *J. Comput. Syst. Sci.*, 87:58–83, 2017.

[55] Andrzej S. Murawski and Nikos Tzevelekos. Higher-order linearisability. In Roland Meyer and Uwe Nestmann, editors, *28th International Conference on Concurrency Theory, CONCUR 2017, September 5-8, 2017, Berlin, Germany*, volume 85 of *LIPIcs*, pages 34:1–34:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2017.

[56] Joseph P. Near and Daniel Jackson. Rubicon: bounded verification of web applications. In Will Tracz, Martin P. Robillard, and Tevfik Bultan, editors, *20th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-20), SIGSOFT/FSE'12, Cary, NC, USA - November 11 - 16, 2012*, page 60. ACM, 2012.

[57] Phuc C. Nguyen, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. *PACMPL*, 2(POPL):51:1–51:30, 2018.

[58] Phuc C. Nguyen and David Van Horn. Relatively complete counterexamples for higher-order programs. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 446–456. ACM, 2015.

[59] Hanno Nickau. Hereditarily sequential functionals. In Anil Nerode and Yuri V. Matiyasevich, editors, *Logical Foundations of Computer Science, Third International Symposium, LFCS'94, St. Petersburg, Russia, July 11-14, 1994, Proceedings*, volume 813 of *Lecture Notes in Computer Science*, pages 253–264. Springer, 1994.

[60] Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of program analysis.* Springer, 1999.

[61] Ivica Nikolić, Aashish Kolluri, Ilya Sergey, Prateek Saxena, and Aquinas Hobor. Finding the greedy, prodigal, and suicidal contracts at scale. In *Proceedings of the 34th Annual Computer Security Applications Conference*, ACSAC '18, pages 653–663, New York, NY, USA, 2018. ACM.

[62] C.-H. L. Ong. On model-checking trees generated by higher-order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*, pages 81–90, Aug 2006.

[63] Doron A. Peled. All from one, one for all: on model checking using representatives. In Costas Courcoubetis, editor, *Computer Aided Verification, 5th International Conference, CAV '93, Elounda, Greece, June 28 - July 1, 1993, Proceedings*, volume 697 of *Lecture Notes in Computer Science*, pages 409–423. Springer, 1993.

[64] Quoc-Sang Phan, Pasquale Malacaria, and Corina S. Pasareanu. Concurrent bounded model checking. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–5, 2015.

[65] Andrew Pitts and Ian Stark. Operational reasoning for functions with local state. In *Higher Order Operational Techniques in Semantics*, pages 227–273. Cambridge University Press, 1998.

[66] Andrew M. Pitts. *Nominal Sets: Names and Symmetry in Computer Science*. Cambridge University Press, New York, NY, USA, 2013.

[67] Gordon D. Plotkin. LCF considered as a programming language. *Theor. Comput. Sci.*, 5(3):223–255, 1977.

[68] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. In Kousha Etessami and Sriram K. Rajamani, editors, *Computer Aided Verification*, pages 82–97, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[69] John C. Reynolds. Definitional interpreters for higher-order programming languages. *Higher-Order and Symbolic Computation*, 11(4):363–397, 1998.

[70] Williame Rocha, Herbert Rocha, Hussama Ismail, Lucas Cordeiro, and Bernd Fischer. Depthk: A k-induction verifier based on invariant inference for c programs. In *Proceedings, Part II, of the 23rd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 10206*, pages 360–364, Berlin, Heidelberg, 2017. Springer-Verlag.

[71] Grigore Rosu and Traian-Florin Serbanuta. An overview of the K semantic framework. *J. Log. Algebraic Methods Program.*, 79(6):397–434, 2010.

[72] Ryosuke Sato and Naoki Kobayashi. Modular verification of higher-order functional programs. In Hongseok Yang, editor, *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017*, volume 10201 of *Lecture Notes in Computer Science*, pages 831–854. Springer, 2017.

[73] Ryosuke Sato, Hiroshi Unno, and Naoki Kobayashi. Towards a scalable software model checker for higher-order programs. In Elvira Albert and Shin-Cheng Mu, editors, *Proceedings of the ACM SIGPLAN 2013 Workshop on Partial Evaluation and Program Manipulation, PEPM 2013*, pages 53–62. ACM, 2013.

[74] Koushik Sen and Gul Agha. Cute and jcute: Concolic unit testing and explicit path model-checking tools. In Thomas Ball and Robert B. Jones, editors, *Computer Aided Verification*, pages 419–423, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[75] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages.* PhD thesis, Carnegie-Mellon University, 1991.

[76] Bjarne Steensgaard. Points-to analysis in almost linear time. In Hans-Juergen Boehm and Guy L. Steele Jr., editors, *POPL'96: The 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 32–41. ACM Press, 1996.

[77] Taku Terao and Naoki Kobayashi. A zdd-based efficient higher-order model checking algorithm. In Jacques Garrigue, editor, *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, volume 8858 of *Lecture Notes in Computer Science*, pages 354–371. Springer, 2014.

[78] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In Gary T. Leavens and Matthew B. Dwyer, editors, *Proceedings of the 27th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2012, part of SPLASH 2012, Tucson, AZ, USA, October 21-25, 2012*, pages 537–554. ACM, 2012.

[79] Emina Torlak and Rastislav Bodík. A lightweight symbolic virtual machine for solver-aided host languages. In Michael F. P. O'Boyle and Keshav Pingali, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, pages 530–541. ACM, 2014.

[80] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[81] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. QSYM : A practical concolic execution engine tailored for hybrid fuzzing. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 745–761. USENIX Association, 2018.