ABSTRACT

Title of dissertation: HIGHER-ORDER SYMBOLIC EXECUTION

Phúc C. Nguyễn
Doctor of Philosophy, 2019

Dissertation directed by: Professor David Van Horn
Department of Computer Science

There are multiple challenges in designing a static verification system for an existing programming language. There is the technical challenge of achieving soundness and precision in the presence of expressive language features such as dynamic typing, higher-order functions, mutable state, control operators, and their idiomatic usage. There is also the practical challenge of allowing gradual adoption of verification in the face of large code bases in the real world. Failure to achieve this gradual property hinders the system's adoption in practice: Existing correct but unverifiable components due to the lack of annotations, the unavailability or the source code, or the inherent incompleteness of static checking would require tedious modifications to a program to make it safe and executable again.

This dissertation shows a simple framework for integrating rich static reasoning into an existing expressive programming language by leveraging dynamic checks and a novel form of symbolic execution. *Higher-order symbolic execution enables gradual verification that is sound, precise, and modular for expressive programming languages.* First, symbolic execution can be generalized to higher-order

programming languages: Symbolic functions do not make bug-finding and counterexample generation fundamentally more difficult, and the counterexample search is relatively complete with respect to the underlying first-order SMT solver. Next, finitized symbolic execution can be viewed as verification, where dynamic contracts are the specifications, and the lack of run-time errors signifies correctness. A further refinement to the semantics of applying symbolic functions yields a verifier that soundly handles higher-order imperative programs with challenging patterns such as higher-order stateful call-backs and aliases to mutable state. Finally, with a novel formulation of termination as a run-time contract, symbolic execution can also verify total-correctness.

Using symbolic execution to statically verify dynamic checks has important consequences in scaling the tool in practice. Because symbolic execution closely models the standard semantics, dynamic language features and idioms such as first-class contracts and run-time type tests do not introduce new challenges to verification and bug-finding. Moreover, the method allows gradual addition and strengthening of specifications into an existing program without requiring a global re-factorization: The programmer can decide to stop adding contracts at any point and still have an executable and safe program. Properties that are unverifiable statically can be left as residual checks with the existing familiar semantics. Programmers benefit from static verification as much as possible without compromising language features that may not fit in a particular static discipline. In particular, this dissertation lays out the path to adding gradual theorem proving into an existing untyped, higher-order, imperative programming language.

HIGHER-ORDER SYMBOLIC EXECUTION

by

Phúc C. Nguyễn

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2019

Advisory Committee:
Professor David Van Horn, Chair/Advisor
Professor Rance Cleaveland
Professor Tudor Dumitras
Professor Michael Hicks
Professor Sam Tobin-Hochstadt

© Copyright by
Phúc C. Nguyễn
2019

Preface

Much of the material in this thesis has previously appeared in the following peer-reviewed publications, authored jointly with David Van Horn, Sam Tobin-Hochstadt, and Thomas Gilray:

- Phúc C. Nguyễn, and David Van Horn. Relatively complete counterexamples for higher-order programs. In *Programming Language Design and Implementation (PLDI)*. ACM, Portland, OR, USA, 2015

- Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification for higher-order stateful programs. In *Principles of Programming Languages (POPL)*. ACM, Los Angeles, CA, USA, 2018

- Phúc C. Nguyễn, Thomas Gilray, Sam Tobin-Hochstadt, and David Van Horn. Size-change termination as a contract. In *Programming Language Design and Implementation (PLDI)*. ACM, Phoenix, AZ, USA, 2019

Acknowledgments

First and foremost, I would like to thank my advisor David Van Horn, for without him, none of my work would have happened. David taught me how to write and present effectively, and saw value in my work on termination contracts when even I was skeptical. Thanks Sam Tobin-Hochstadt for his helpful discussions and constructive feed-backs throughout the years. Thanks Matthias Felleisen for delivering inspirational lectures and introducing me to David and Sam, starting all of this. Thanks everyone on my committee for detailed feed-backs on my work and writing.

Thanks to everybody in the PLUM lab for helpful pointers and feed-backs on my work. Thanks Nick Labich for being a supportive lab-mate and roommate, making me coffee and foods. Thanks Thomas Gilray for helping me with writing and dealing with most of LaTeX magics for me. Thanks Clay Mentzer, Venkata Gaddam, William Kunkel, and Cameron Moy for their various work supporting my project.

Thanks to my cousins Kha and Thoa for letting me stay at their place during my early days in the States. Thanks to my parents, brother, uncles and aunts, and cousins for their support throughout the years.

Thanks to anonymous reviewers of my papers for their constructive feed-backs and pointers.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

SCT	Size-chage Termination
PCF	Programming Computable Functions
CPCF	Contracted PCF
SPCF	Symbolic PCF
SMT	Satisfiability Modulo Theories

Chapter 1: Introduction

Researchers constantly want to enrich existing programming languages with more static checking, such as adding static types to untyped languages [1–3], or adding refinement types to coarsely typed languages [4–7]. One recurring problem is that support for expressive language features tends to be non-trivial, with soundness hard to establish due to subtle mismatches between the static and dynamic semantics. For example, refinement type checking for eager languages was discovered unsound when applied as-is to lazy languages due to the difference between values and thunks [6], and set-based analysis for higher-order contracts [8] was discovered unsound due to the difference between mathematical mappings and higher-order functions in programming languages [9]. In addition, soundness for a verified component sometimes is only guaranteed under the assumption that all components it interacts with have also been verified. Lifting this assumption often requires additional work devising dynamic checks at component boundaries and proving that these checks imply the desired static properties.

The challenges seem inherent for real-world programming languages, and compromises seem unavoidable. First, there is a natural trade-off between a language's expressiveness and its amenability to static verification. For example, first-class

functions prevent program flows from being straightforwardly computed from syntax. Dynamic typing obscures basic invariants about the data shapes, which is only worsened by dynamic typing idioms that rely on path-sensitive, inter-procedural reasoning to justify the use of partial functions. Mutable states introduce implicit communication channels that can invalidate previously established invariants. Control operators destroy assumptions about "well-bracketed" program flows. The interaction of these features only introduce more opportunities for unexpected behavior, challenging manual as well as automatic reasoning. Second, many static properties, such as totality of functions, do not have any obvious corresponding dynamic checks.

This dissertation presents higher-order symbolic execution as a framework for gradually enriching an existing programming language with more static reasoning tools. Symbolic execution turns out an effective tool for turning dynamic checks into static verification. When a specification is formulated as a dynamic check, it can either be turned into static checks, giving programmers early feed-backs without incurring any run-time cost, or be deferred to run-time with the existing familiar semantics. A verification system based on this framework has multiple advantages: It is relatively simple to ensure soundness, because symbolic execution (and its finite abstraction) is an extension to the standard semantics that often proceed in lock-step with it. It is also straightforward to employ a path-condition and state-of-the-art SMT solvers to achieve good precision and accommodate many language idioms. Higher-order symbolic execution is also modular, hence significantly mitigates the common problem of path-explosion. Finally, as specifications are originally run-time checks, unverifiable properties due to the inherent limitation of static checking can

be left as-is, allowing a flexible and safe interaction between verified and unverified components.

In the rest of this document, chapter 2 generalizes symbolic execution to the higher-order case, and shows that symbolic functions do not fundamentally complicate the search space. Chapter 3 shows the use of symbolic execution as a static verification, turning first-class higher-order contracts into expressive specifications, most of which can be statically verified and discharged. Chapter 4 shows a novel formulation of termination as a dynamically checkable property, which in turn allows enforcing total correctness at run-time, which are also eligible for static verification through symbolic execution. The sections below give an overview of the next chapters.

1.1 Relatively complete counterexamples for higher-order programs

The Problem Demonstrating bugs with concrete counterexamples is extremely useful in helping programmers understand defects in their code [10–14]. Symbolic execution has proven effective in achieving this goal for first-order programs [10,11]. The gist of symbolic execution is simple: We simply run programs under a modified semantics that allows symbolic values, which may stand for multiple concrete values. Symbolic execution then accumulates a path-condition remembering invariants about symbolic values that are gained from taking conditional branches. The path-conditions are helpful for eliminating spurious branches and generating fully concrete counterexamples demonstrating reachable errors when they are found. Ex-

tending symbolic execution for higher-order programming languages would mean that symbolic values can also stand for higher-order functions. The situation may seem hopeless: A symbolic function represents unknown code, so how should execution proceed when one is applied, and how do we synthesize higher-order functions to reproduce errors? The obvious approach of modeling functions as mappings has multiple flaws. First, higher-order functions interact with their contexts not only through the values they return, but also through the values they supply to their functional arguments; modeling higher-order functions as mappings would miss the second way functions interact with their contexts. Moreover, that functions can be inputs of mappings means it would be non-trivial to decide if some keys should be the same or distinct, because equality between executable functions is not trivial to define.

Main Ideas We solve the problem of symbolically executing higher-order programs by noticing that even though the space of higher-order functions is huge, there are only a few canonical ways in which unknown code interact with the known code under symbolic execution to cause an error that the known code is responsible for. In particular, there is no need to consider unknown code that introduces its own errors (because verifying unknown code is not a very interesting problem). In addition, because any error trace is finite, there is no need for the unknown code to be recursive (because any counterexample with recursive functions could have been unrolled to behave equivalently without). Finally, also due to error traces being finite, there is no need to search for counterexamples with primitive functions

such as addition or multiplication in their code: They can be approximated as finite maps over the finite error traces. In the end, higher-order unknown functions only need to either ignore their arguments, or applying their arguments to a lower-order function and recursively interact with the result. We use the operational semantics to precisely define the notion of "who's responsible for error" and prove soundness and relative completeness of counterexample generation for a symbolic extension of the PCF language [15].

Evaluation To evaluate the effectiveness of higher-order symbolic execution for generating concrete counterexamples, we integrate this work into an existing contract verification system for a subset of Racket programs. The language supports first-class contracts, rich base values, and a simple module system. Benchmarks are taken from prior lines of work on verification [1, 16–18] along with their modified counterparts introducing bugs, as well as anonymous submissions to our online tool. The tool quickly finds bugs in most benchmarks, some of which proven challenging for more lightweight techniques such as random testing [19].

1.2 Soft-contract verification for higher-order stateful programs

The Problem Higher-order contracts generalize the traditional pre-and-post conditions and enable dynamically enforcing invariants on higher-order functions. Contracts can be composed using the full expressiveness of the host programming language, and allow all correct programs to execute, as opposed to conservatively rejecting unverifiable programs as in more static approaches such as type systems. Despite

the advantages, contracts have obvious downsides: They delay error discovery until run-time, and can introduce significant execution overhead. Verifying contracts statically as much as possible, while leaving unverifiable ones as residual run-time checks would bring in the benefits of both dynamic checking and static verification: Programmers can state rich specifications without worrying about overhead from most or all contracts, receive early warnings about violations, and always have correct and executable programs. Contract verification, however, is challenging, due to the highly dynamic nature of contracts: They can be composed from arbitrary expressions capable of crashing, diverging, and modifying states, and first-class contracts are computed at run-time. This prevents straightforward verification methods that translate contracts into logical formulas in some decidable theories. In addition, contracts are customarily used as boundary enforcement, and programmers tend to not write contracts for private functions, relying on invariants established by inter-procedural and path-sensitive reasoning. A compositional analysis is unlikely to succeed in verifying idiomatic contracted programs without requiring heavy annotations. Modularity, however, is crucial in scaling any analysis to a realistic code base. Previous approaches to contract verification place restrictions on the language and contracts that limit applicability to a realistic programming environment [18, 20–23].

Main Ideas We solve the problem of sound, precise, and modular contract verification of idiomatic programs in a higher-order, untyped, imperative language using higher-order symbolic execution. Symbolic execution provides a natural frame-

work for path-sensitive reasoning, which is crucial in effectively handling idiomatic untyped programs with dynamic type tests. Operational aspects in the language such as higher-order functions or divergence are handled on the execution side, and the solver is only integrated to do what it is best at: solving complex properties on base values such as arithmetic. There is no heavyweight translation of the general-purpose programming language into the restricted logical formulas. By letting the unknown code remember (behavioral) values that have escaped to unknown functions, symbolic execution soundly approximates possible interactions in higher-order stateful programs and can cause the code under verification to violate contracts. That symbolic values can be functions makes the analysis modular, allowing omitting arbitrary components in the program and approximating them with (contracted) symbolic values. To ensure termination of symbolic execution, we abstract it using well-studied methods for computing a sound, finite approximation of an existing semantics [24, 25].

Evaluation We implement the verifier for a significant subset of core Racket, and evaluate it on benchmarks from multiple lines of work [1, 9, 16, 18, 26], as well as realistic libraries collected from different sources. Different benchmark suites demonstrate different challenges for verification. In total, benchmarks consist of 86.7% imperative and 13.2% pure functional (by lines of code). The tool is shown to not only verify almost all contracts, but also works for many interesting programming patterns, with reasonable analysis time even for large programs. It also uncover genuine bugs in some programs, confirmed by manual inspection.

1.3 Size-change termination as a contract

The Problem Using dynamic checks as specifications seemingly has a flaw: They cannot directly check for liveness properties such as termination, which would imply that contracts cannot enforce total correctness. It seems that adding termination checking to an existing language would require either intrusively adding a static type system with inductive data types, or sub-optimally applying a separate termination analysis that would not support gradual enforcement.

Main Ideas We solve the problem of supporting gradual enforcement of termination (and therefore total contracts) by expressing termination checking as a contract. The key to achieving this goal is to approximate termination using a safety property that implies it. Size-change termination [27,28] has proven effective in checking for a wide range of terminating patterns in first-order and higher-order programs without user annotations. The original analysis is presented as a size-change principle for detecting a large class of terminating programs, and an analysis that checks if the statically approximated program follows that principle. In our work, we transplant this principle into the context of run-time checking, and propose an operational semantics supporting a mix of terminating and non-terminating functions. Termination is ultimately broken down into simple "less-than" properties that symbolic execution can readily check. A program enforced with termination contract terminates either by adhering to the size-change principle, or by violating it and being aborted. Termination contracts bring in new expressive styles of programming.

First, dynamic termination checking can enforce termination in programs that are challenging to check statically, such as Turing-complete language interpreters interpreting terminating programs. Second, programmers can specify termination that can be gradually verified just as other properties, and the program is always safe and executable. Finally, they now can use total contracts as propositions stating various properties about the program, and the terms that satisfy these contracts are proofs. Even when the terms cannot be statically checked, they are useful for random testing.

Evaluation We evaluate termination contracts' precision and run-time overhead on terminating programs, and time to detect non-terminating programs. We also evaluate the contract verifier's effectiveness in verifying total contracts statically, and whether it can work well as a theorem prover. To evaluate our termination contract approach, we use benchmarks from different lines of work on termination checking for functional programs [6,27–30], as well as multiple larger Scheme benchmarks, and find dynamic checks robust against challenges such as dynamic types and higher-order functions. To evaluate the contract verifier, in addition to these benchmarks, we also test it on basic proofs of properties on lists. The contract verifier is competitive, though not dominant, when compared to other tools, and can be used to check inductive properties where the proofs need only describe the high-level structure, analogous to F* [31] and Liquid Haskell [32].

Chapter 2: Relatively Complete Counterexamples for Higher-Order Programs

In this chapter, we study the problem of generating inputs to a higher-order program causing it to error. We first approach the problem in the setting of PCF, a typed, core functional language and contribute the first relatively complete method for constructing counterexamples for PCF programs. The method is relatively complete with respect to a first-order solver over the base types of PCF. In practice, this means an SMT solver can be used for the effective, automated generation of higher-order counterexamples for a large class of programs.

We achieve this result by employing a novel form of symbolic execution for higher-order programs. The remarkable aspect of this symbolic execution is that even though symbolic higher-order inputs and values are considered, the path condition remains a first-order formula. Our handling of symbolic function application enables the reconstruction of higher-order counterexamples from this first-order formula.

After establishing our main theoretical results, we sketch how to apply the approach to untyped, higher-order, stateful languages with first-class contracts and show how counterexample generation can be used to detect contract violations in this

setting. To validate our approach, we implement a tool generating counterexamples for erroneous modules written in Racket.

2.1 Introduction

Generating inputs that crash first-order programs is a well-studied problem in the literature on symbolic execution [10, 11], type systems [12], flow analysis [13], and software model checking [14]. However, in the setting of higher-order languages, those that treat computations as first-class values, research has largely focused on the verification of programs without investigating how to effectively report counterexamples as concrete inputs when verification fails (e.g., [5, 9, 18, 20, 22, 33]).

There are, however, a few notable exceptions which tackle the problem of counterexamples for higher-order programs. Perhaps the most successful has been the approach of random testing found in tools such as *QuickCheck* [19, 34]. While testing works well, it is not a complete method and often fails to generate inputs for which a little symbolic reasoning could go further. Symbolic execution aims to overcome this hurdle, but previous approaches to higher-order symbolic execution can only generate *symbolic* inputs, which are not only less useful to programmers, but may represent infeasible paths in the program execution [9, 18]. Higher-order model checking [35] offers a complete decision procedure for typed, higher-order programs with finite base types, and can generate inputs for programs with potential errors. Unfortunately, only first-order inputs are allowed. This assumption is reasonable for whole programs, but not suitable for testing higher-order *components*, which

often consume and produce behavioral values (e.g., functions, objects). [36] gives an approach to dependent type inference for ML that relies on counterexample refinement. This approach can be used to generate higher-order counterexamples; however no measure of completeness is considered.

In this chapter, we solve the problem of generating potentially higher-order inputs to functional programs. We give the first relatively complete approach to generating counterexamples for PCF programs. Our approach uses a novel form of symbolic execution for PCF that accumulates a path condition as a symbolic heap. The semantics is an adaptation of [18], where the critical technical distinction is our semantics maintains a *complete* path condition during execution. The key insight of this work is that although the space of higher-order values is huge, it is only necessary to search for counterexamples from a subset of functions of specific shapes. Symbolic function application can be leveraged to decompose unknown functions to lower-order unknown values. By the point at which an error is witnessed, there are sufficient *first-order* constraints to reconstruct the potentially higher-order inputs needed to crash the program. The completeness of generating counterexamples reduces to the completeness of solving this first-order constraint, and in this way is relatively complete [37].

Beyond PCF, we show the technique is not dependent on assumptions of the core PCF model such as type safety and purity. We sketch how the approach scales to handle untyped, higher-order, imperative programs. We also show the approach seamlessly scales to handle first-class behavioral contracts [38] by incorporating existing semantics for contract monitoring [39] with no further work. The semantic

decomposition of higher-order contracts into lower-order functions naturally composes with our model of unknown functions to yield a contract counterexample generator for Contract PCF (CPCF) [9].

Contributions We make the following contributions:

1. We give a novel symbolic execution semantics for PCF that gradually refines unknown values and maintains a complete path condition.

2. We give a method of integrating a first-order solver to simultaneously obtain a precise execution of symbolic programs and enable the construction of higher-order counterexamples in case of errors.

3. We prove our method of finding counterexamples is sound and relatively complete.

4. We discuss extensions to our method to handle untyped, higher-order, imperative programs with contracts.

5. We implement our approach as an improvement to a previous contract verification system, distinguishing definite program errors from potentially false positives.

Outline The remainder of the chapter is organized as follows. We first step through a worked example of a higher-order program that consumes functional inputs (§ 2.2). Stepping through the example illustrates the key ideas of how the path condition is accumulated as a heap of potentially symbolic values with refinements

and how this heap can be translated to a first-order formula suitable for an SMT solver. Generating a model for the path condition at the point of an error reconstructs the higher-order input needed to witness the error. Next, we develop the core model of Symbolic PCF (§ 2.3) as a heap-based reduction semantics. We prove that the semantics is sound and relatively complete, our main theoretical contribution. We then show how to scale the approach beyond PCF to untyped, higher-order, imperative languages with contracts (§ 2.5). We use these extensions as the basis of a tool for finding contract violations in Racket code to validate our approach (§ 2.6). Finally, we relate our work to the literature (§ 2.7) and conclude (§ 2.8).

2.2 Worked Examples

We illustrate our idea using an *incomplete* OCaml program. The basic idea is that we give a semantics to incomplete programs using a heap of refinements that constrain all possible completions of the program. When an error is reached, the heap is given to an SMT solver, which constructs a model that represents a counterexample.

As our example we use a function `f` that takes as its arguments a function `g` and a number `n` and performs a division whose denominator involves the application of `g` to `n`. We write $\bullet^T$ to denote an unknown value of the appropriate type (and omit the type when it is clear from context). This example, though contrived, is small and conveys the heart of our method.

```
let f (g : int → int) (n : int) : int = 1 / (100 - g n)
in (• f)
```

Now let us consider the possible errors that can arise from running this code for any interpretation of the unknown value.

Although the application of unknown function • is an arbitrary computation that can result in any error, we restrict our attention to possible errors stemming from misbehavior of the visible part of the above code and assume function • is bug-free. Through symbolic execution and incremental refinement of unknown values, we reveal one implementation of • that triggers a division error in `f`'s implementation.

```
Error: Division_by_zero
Breaking context:
   • = fun f → (f (fun n → 100) 0)
```

To find a counterexample, we first seek a possible error by running the program under an extended reduction semantics allowing *unknown*, or *opaque*, values. When execution follows different branches, it remembers assumptions associated with each path, and opaque values become partially *known*, or *transparent*. To keep track of incremental refinements throughout execution, we allocate all values in a heap and maintain an upper bound to the behavior of each unknown value.

The semantics takes the form of a reduction relation on pairs of expressions and heaps, written $(E, \Sigma) \longmapsto (E', \Sigma')$. In our example, the first step of computation is to allocate a fresh location to hold the unknown function being applied.

$$(\texttt{(• f)}, \emptyset) \;\longmapsto\; (\texttt{(}\mathtt{L}_1\ \texttt{f)}, [\mathtt{L}_1 \mapsto \bullet])$$

Allocating values in the heap this way gives us a means to refine values and to communicate these refinements to later parts of the computation.

At this point, we can partially solve for the unknown value. Since it is applied to `f`, it must be a function of one argument. But how can we solve for the body of the function? The key observation is that while many possible solutions for the function body may exist, if the function can reach an error state, then it can reach that error state by immediately applying the input to some arguments, *without loss of generality*. Since the input function takes two arguments, we can partially solve for the body of the function as "apply the input to two unknown values." By allocating these two unknowns and refining `f`, we arrive at the state:

$$\langle \texttt{(f } \mathtt{L}_2 \texttt{ } \mathtt{L}_3\texttt{)}, [\mathtt{L}_1 \mapsto \texttt{fun f} \to \texttt{(f } \mathtt{L}_2 \texttt{ } \mathtt{L}_3\texttt{)},$$
$$\mathtt{L}_2 \mapsto \bullet^{\texttt{int}\to\texttt{int}},$$
$$\mathtt{L}_3 \mapsto \bullet^{\texttt{int}}]$$

The program then executes `f`'s body, substituting `g` with $\mathtt{L}_2$ and `n` with $\mathtt{L}_3$. The next sub-expression to reduce is `(g n)`, which is $\texttt{(}\mathtt{L}_2\ \mathtt{L}_3\texttt{)}$ after substitution, which is yet another unknown function application, so the next step is to partially solve for $\mathtt{L}_2$. Unlike in the higher-order case, there is no interaction with the input value that needs to be considered (since it is not behavioral), so the function can

simply return a new, unknown output, $\texttt{L}_4$, giving us the following transition:

$$\begin{aligned}
&\langle (\texttt{L}_2\ \texttt{L}_3), [\texttt{L}_1 \mapsto \texttt{fun f} \rightarrow \texttt{(f L}_2\ \texttt{L}_3\texttt{)}, \\
&\qquad \texttt{L}_2 \mapsto \bullet^{\texttt{int}\rightarrow\texttt{int}}, \\
&\qquad \texttt{L}_3 \mapsto \bullet^{\texttt{int}}]\rangle \\
\longmapsto \quad &\langle \texttt{L}_4, [\texttt{L}_1 \mapsto \texttt{fun f} \rightarrow \texttt{(f L}_2\ \texttt{L}_3\texttt{)}, \\
&\qquad \texttt{L}_2 \mapsto \texttt{fun n} \rightarrow \texttt{L}_4, \\
&\qquad \texttt{L}_3 \mapsto \bullet^{\texttt{int}}, \\
&\qquad \texttt{L}_4 \mapsto \bullet^{\texttt{int}}]\rangle
\end{aligned}$$

At this point, we need to compute `100 - L`$_4$, i.e. subtract an unknown integer from 100. The solution is simple, we extend the primitive arithmetic operations to produce new unknown values and annotate the unknown result with a predicate to embed the knowledge that it is equal to `100 - L`$_4$:

$$\begin{aligned}
&\langle \texttt{100 - L}_4, [\texttt{L}_1 \mapsto \texttt{fun f} \rightarrow \texttt{(f L}_2\ \texttt{L}_3\texttt{)}, \\
&\qquad \texttt{L}_2 \mapsto \texttt{fun n} \rightarrow \texttt{L}_4, \\
&\qquad \texttt{L}_3 \mapsto \bullet^{\texttt{int}}, \\
&\qquad \texttt{L}_4 \mapsto \bullet^{\texttt{int}}]\rangle \\
\longmapsto \quad &\langle \texttt{L}_5, [\texttt{L}_1 \mapsto \texttt{fun f} \rightarrow \texttt{(f L}_2\ \texttt{L}_3\texttt{)}, \\
&\qquad \texttt{L}_2 \mapsto \texttt{fun n} \rightarrow \texttt{L}_4, \\
&\qquad \texttt{L}_3 \mapsto \bullet^{\texttt{int}}, \\
&\qquad \texttt{L}_4 \mapsto \bullet^{\texttt{int}}, \\
&\qquad \texttt{L}_5 \mapsto \bullet^{\texttt{int,fun x}\rightarrow\texttt{x=(100-L}_4\texttt{)}}]\rangle
\end{aligned}$$

We finally arrive at the point of computing `1 / L5`. At this point the semantics branches non-deterministically since `L5` may represent a zero or non-zero value. In the case of an error, we refine `L5` to be zero, giving us the final state:

$$\langle \texttt{error}, [\texttt{L}_1 \mapsto \texttt{fun f} \rightarrow \texttt{(f L}_2\ \texttt{L}_3\texttt{)},$$
$$\texttt{L}_2 \mapsto \texttt{fun n} \rightarrow \texttt{L}_4,$$
$$\texttt{L}_3 \mapsto \bullet^{\texttt{int}},$$
$$\texttt{L}_4 \mapsto \bullet^{\texttt{int}},$$
$$\texttt{L}_5 \mapsto \bullet^{\texttt{int,fun x}\rightarrow\texttt{x=0,fun x}\rightarrow\texttt{x=(100-L}_4\texttt{)}}] \rangle$$

At this point, the program has reached an error state and has accumulated a heap of invariants that constrain the unknown values. But notice that since functions have been partially solved for as they've been applied, there are only first-order unknowns in the heap. At this point, translation of refinements on integers into first-order assertions is straightforward:

```
(declare-const L3 Int)
(declare-const L4 Int)
(declare-const L5 Int)
(assert (= L5 (- 100 L4)))
(assert (= 0 L5))
```

A solver such as Z3 [40] can easily solve such constraints and yield ($\texttt{L}_3 = \texttt{0}$, $\texttt{L}_4 = \texttt{100}$, $\texttt{L}_5 = \texttt{0}$) as a model. We then plug these values into the current heap and straightforwardly obtain the counterexample shown at the start.

In summary, we use execution to incrementally construct the shape of each function, query a first-order solver for a model for base values, and combine these first-order values to construct a higher-order counterexample.

2.3 Formal Model with Symbolic PCF

This section presents a reduction semantics illustrating the core of our approach. Symbolic PCF (SPCF) [9] extends the PCF language [15] with *incomplete* programs containing *symbolic values* that can be higher-order.

We present the language's syntax and semantics, describe its integration with an external solver, and show how the semantics enables the generation of a counterexample when an error occurs. Finally, we prove that our counterexample construction is sound and complete relative to the underlying solver. The key technical challenge in designing such a semantics is to make sure not to over-constrain unknowns, which would be unsound, while also not under-constraining unknowns, which would be incomplete.

2.3.1 Syntax of SPCF

Figure 2.1 presents the syntax of SPCF. We write $\overrightarrow{E}$ to mean a sequence of expressions and treat it as a set where convenient. The language is simply typed with typical expression forms for conditionals, applications, primitive applications, recursion, and values such as natural numbers and lambdas. The evaluation context $\mathcal{E}$ is standard for a call-by-value semantics. We highlight non-standard forms in gray. The key extension of SPCF compared to PCF is the notion of *symbolic*, or *opaque* values. We write $\bullet^T$ to mean an unknown but fixed and syntactically closed value[1] of type T. The system automatically annotates each opaque value with a unique

[1]For example, $\bullet$ does not approximate `(λ (x) y)`

label to identify its source location. It also uniquely labels each source location that could have a potential run-time failure. In SPCF, such failures can only occur with the application of partial, primitive operations.

When evaluating an SPCF expression, we allocate all values and maintain a heap to keep track of their constraints. When execution proceeds through conditional branches and primitive operations, we refine the heap at appropriate locations with stronger assumptions taken at each branch. As figure 2.1 shows, a heap is a finite function mapping each location L to a stored value S as an upper bound of the value's run-time behavior. A stored value S is similar to a syntactic value, but a stored unknown value can be further refined by arbitrary program predicates. For example, $\bullet^{\{\texttt{nat},\ (\lambda\ (\texttt{x})\ (\texttt{even?}\ \texttt{x}))\}}$ denotes an unknown even natural number.

In addition, we use $\texttt{case}^T\ \overrightarrow{[L \mapsto L]}$ to denote a mapping approximating an unknown function of type $\texttt{nat} \to T$. We clarify the role of this construct later when discussing the semantics of applying opaque functions, but the intuition is that this form is used to constrain unknown functions (of base type input) to always produce the same result when given the same input; it is critical for achieving completeness and is not present in the original SCPCF semantics of [9].

Syntax for answers A is internal and unavailable to programmers. An answer is either a location L^T pointing to a value of type T on the heap, or an error message $\texttt{err}_O^L$ *blaming* source location L for violating primitive O's precondition. A source location in an error message is not just for precise blaming, but is important in defining what it means to have a sound symbolic execution, as we will discuss in detail in section 2.3.2.

[Expressions]	$E ::= \mathrm{A} \mid V \mid X \mid (\mathsf{if}\ E\ E\ E) \mid (E\ E) \mid (O\ \overrightarrow{E})^{\ L}$
[Contexts]	$\mathcal{E} ::= [\,] \mid (\mathsf{if}\ \mathcal{E}\ E\ E) \mid (\mathcal{E}\ E) \mid (L\ \mathcal{E}) \mid (O\ \overrightarrow{L}\ \mathcal{E}\ \overrightarrow{E})$
[Values]	$V ::= \bullet_L^T \mid (\lambda\ (X : T)\ E) \mid N$
[Answers]	$\mathrm{A} ::= L^T \mid \mathtt{err}_O^L$
[Operations]	$O ::= \mathtt{zero?} \mid \mathtt{add1} \mid \mathtt{div} \mid \ldots$
[Predicates]	$P ::= (\lambda\ (X : T)\ E)$
[Types]	$T ::= \mathtt{nat} \mid T \to T$
[Heaps]	$\Sigma ::= \emptyset \mid \Sigma, L \mapsto S$
[Storeables]	$S ::= \bullet^{\{T\overrightarrow{P}\}} \mid (\lambda\ (X{:}T)\ E) \mid N \mid \mathtt{case}^T\ \overrightarrow{L \mapsto L}$
[Variables]	$X, L \in \textit{identifier}$

Figure 2.1: Syntax of SPCF

We omit straightforward type-checking rules for SPCF and assume all considered programs are well-typed. In addition, we omit showing types and labels for constructs such as locations and lambdas when they are irrelevant or clear from context.

In the following, we use the term *unknown program portion* to refer to all unknown values in the original (incomplete) program, and *known program portion* to refer to the rest of it.

2.3.2 Semantics of SPCF

We present the semantics of SPCF as a binary relation ($\longmapsto$) between states of the form (E, Σ). Key extensions to the straightforward concrete semantics include generalization of primitives to operate on symbolic values and reduction rules

for opaque applications. Intuitively, reduction on abstract states approximates reduction on concrete states, accounting for all possible instantiations of symbolic values. Figure 2.2 presents the reduction semantics of SPCF. The semantics is also mechanized as a Redex model and available online.[2]

Each value is allocated on the heap and reduces to a location as shown in rules *Opq* and *Conc*. Because an opaque value stands for an arbitrary but fixed and closed value, we reuse a location if it has been previously allocated.

Rule *Prim* shows the reduction of a primitive application. We use δ to relate primitive operators and values to results. Typically, δ is a function, but here it is a relation because primitive operations may behave non-deterministically on unknown values. In addition, the relation includes a heap to remember assumptions in each taken branch. Rules for conditionals are straightforward, except we also rely on δ to determine the truth of the value branched on instead of replicating the logic. We use `0` to indicate falsehood and any non-zero number for truth (as in PCF). Application of a λ-abstraction follows standard β-reduction.

Application of an unknown value to a function argument results in a range of possibilities to consider. This space, however, can be partitioned into a few cases. First, the unknown program portion can have bugs of its own regardless of the argument, but our concern is only to find bugs in the known program portion so the possibility of these errors is ignored. Second, the function argument escapes into an unknown context and can be invoked in an arbitrary way. However, any invocation

[2] `https://github.com/philnguyen/soft-contract/tree/pldi-2015/soft-contract/ce-redex`

triggering an error can be reduced to a chain of function applications. Alternatively, the unknown function may not explore its argument's behavior directly during the execution of its body, but delay that in a returned closure. Finally, the unknown function may completely ignore its argument and fail to reveal any hidden bug, allowing the program to proceed to other parts. These four cases result in specific shapes a function can have. Therefore, upon opaque function application, we refine the opaque function's shape accordingly.

Consider this example:

$$\bullet^{(\texttt{nat}\to\texttt{nat})\to T}_{\texttt{L}_1}\ (\lambda\ (\texttt{x} : \texttt{nat})\ (/\ 1\ \texttt{x})^{\texttt{L}})$$

and the following possible instantiations of $\texttt{L}_1$:

1. `(λ (f) (/ 1 0))`
2. `(λ (f) (f 0))`
3. `(λ (f) (λ (x) (add1 (f x))))`
4. `(λ (f) (λ (x) 42))`

Completion (1) raises an error from within the unknown function blaming $\texttt{L}_1$ itself, (2) triggers the division error blaming `L`, (3) delays the exploration of its argument's behavior by returning a closure referencing the argument, and (4) is a constant function ignoring its argument. As we are only interested in errors in the known program portion, we ignore behavior such as (1). Rules *AppOpq1*, *AppOpq2*, *AppOpq3* and *AppHavoc* model the remaining possibilities.

$(\bullet_L^T, \Sigma) \longmapsto (L^T, \Sigma')$ where $\Sigma' = \Sigma[L \mapsto \bullet_L^T]$ if $L \notin dom(\Sigma)$, or Σ otherwise [*Opq*]

$(V, \Sigma) \longmapsto (L, \Sigma[L \mapsto V])$ where $L \notin dom(\Sigma)$ and $V \neq \bullet$ [*Conc*]

$((\mathsf{if}\ L\ E_1\ E_2), \Sigma) \longmapsto (E_1, \Sigma')$ $\delta(\Sigma, \mathtt{zero?}, L) \ni (0, \Sigma')$ [*IfTrue*]

$((\mathsf{if}\ L\ E_1\ E_2), \Sigma) \longmapsto (E_2, \Sigma')$ $\delta(\Sigma, \mathtt{zero?}, L) \ni (1, \Sigma')$ [*IfFalse*]

$((O\ \vec{L}), \Sigma) \longmapsto (L', \Sigma'[L' \mapsto V])$ if $\delta(\Sigma, O, \vec{L}) \ni (V, \Sigma')$ and $L' \notin dom(\Sigma')$ [*Prim*]

$((L\ L_x), \Sigma) \longmapsto ([L_x/X]E, \Sigma)$ if $(\Sigma\,(L)) = (\lambda\ (X)\ E)$ [*AppLam*]

$((L\ L_x), \Sigma) \longmapsto$
$(L_a, \Sigma[L_a \mapsto \bullet^T, L \mapsto \mathtt{case}^T\ [L_x \mapsto L_a]])$ if $(\Sigma\,(L)) = \bullet^{\mathtt{nat} \to T}$ and $L_a \notin dom(\Sigma)$ [*AppOpq1*]

$((L\ L_x), \Sigma) \longmapsto (L_a, \Sigma[L_a \mapsto \bullet^T, L \mapsto (\lambda\ (\mathtt{x} : T')\ L_a)])$ if $(\Sigma\,(L)) = \bullet^{T' \to T}$, $T' = T_1 \to T_2$ and $L_a \notin dom(\Sigma)$ [*AppOpq2*]

$((L\ L_x), \Sigma) \longmapsto ([L_x/\mathtt{x}]V, \Sigma')$ if $(\Sigma\,(L)) = \bullet^{T' \to T}$, $T' = T_1 \to T_2$, $T = T_3 \to T_4$ [*AppOpq3*]
where $\Sigma' = \Sigma[L \mapsto (\lambda\ (\mathtt{x} : T')\ V), L_1 \mapsto \bullet^{T' \to T}]$
$L_1 \notin dom(\Sigma)$, and $V = (\lambda\ (\mathtt{y} : T_3)\ ((L_1\ \mathtt{x})\ \mathtt{y}))$

$((L\ L_x), \Sigma) \longmapsto$
$((L_2\ (L_x\ L_1)), \Sigma[L \mapsto V, L_1 \mapsto \bullet^{T_1}, L_2 \mapsto \bullet^{T_2 \to T}])$ if $(\Sigma\,(L)) = \bullet^{T' \to T}$, $T' = T_1 \to T_2$, $L_1, L_2, L_a \notin dom(\Sigma)$, and $V = (\lambda\ (\mathtt{x} : T')\ (L_2\ (\mathtt{x}\ L_1)))$ [*AppHavoc*]

$((L\ L_x), \Sigma) \longmapsto (L_a, \Sigma)$ if $(\Sigma\,(L)) = \mathtt{case}\ \ldots\ [L_x \mapsto L_a]\ \ldots$ [*AppCase1*]

$((L\ L_x), \Sigma) \longmapsto$
$(L_a, \Sigma[L \mapsto \mathtt{case}\ [L_z \mapsto L_b] \ldots\ [L_x \mapsto L_a]])$ if $(\Sigma\,(L)) = \mathtt{case}\ [L_z \mapsto L_b] \ldots$and $L_x \notin \{L_z \ldots\}$ and $L_a \notin dom(\Sigma)$ [*AppCase2*]

$(\mathcal{E}[E], \Sigma) \longmapsto (\mathcal{E}[E'], \Sigma')$ if $(E, \Sigma) \longmapsto (E', \Sigma')$ [*Close*]

$(\mathcal{E}[\mathtt{err}_O^L], \Sigma) \longmapsto (\mathtt{err}_O^L, \Sigma)$ if $\mathcal{E} \neq [\,]$ [*Error*]

Figure 2.2: Semantics of SPCF

Rule *AppOpq1* shows a simple case where the argument is a first-order value with no behavior. In this case, we approximate the application's result with a symbolic value of appropriate type, and refine the opaque function to be of the form $\texttt{case}^T\ \overrightarrow{[L \mapsto L]}$ to remember this mapping. Any future application of this function to an equal argument gives an equal result.

Applying a higher-order opaque function results in multiple distinct possibilities. Rule *AppOpq2* considers the case where the function ignores its argument (i.e. it is a constant function). Any future application of this unknown function gives the same result. Rule *AppOpq3* considers the case where the unknown context does not immediately explore its argument's behavior but delays that work by wrapping the argument inside another function. The context using this result may or may not reveal a potential error. Finally, rule *AppHavoc* considers the case where the unknown context explores its argument's behavior by supplying an unknown value to its argument and putting the result back into another unknown context.

When the argument is higher-order, we do not use a simple dispatch as in rule *AppOpq1* because there is no mechanism for comparing functions for equality (without applying them as in rule *AppHavoc*).

Application rules for mappings are straightforward. Rule *AppCase1* reuses the result's location for a previously seen application, whereas rule *AppCase2* allocates a fresh location for the result of a newly seen application.

These rules for opaque application collectively model the *demonic* context in previous work on higher-order symbolic execution [9], but they unroll the unknown context incrementally and remember its shape to enable counterexample construc-

$$\delta(\Sigma, \texttt{zero?}, L) \ni (\texttt{1}, \Sigma) \qquad \text{if } \Sigma \vdash L : \texttt{zero?}\checkmark$$
$$\delta(\Sigma, \texttt{zero?}, L) \ni (\texttt{0}, \Sigma) \qquad \text{if } \Sigma \vdash L : \texttt{zero?}\text{✗}$$
$$\delta(\Sigma, \texttt{zero?}, L) \supseteq \{(\texttt{1}, \Sigma[L \mapsto \texttt{0}]), (\texttt{0}, \Sigma[L \mapsto \neg\texttt{zero?}])\} \qquad \text{if } \Sigma \vdash L : \texttt{zero?}\,\texttt{?}$$
$$\delta(\Sigma, \texttt{div}, L_1, L_2) \ni (m/n, \Sigma) \qquad \text{if } \Sigma(L_1) = m \text{ and } \Sigma(L_2) = n, n \neq \texttt{0}$$
$$\delta(\Sigma, \texttt{div}, L_1, L_2) \ni (\bullet^{\texttt{nat}}, \texttt{(}\equiv\ L_1\ \texttt{/}\ L_2\texttt{)}, \Sigma') \qquad \text{if } \Sigma(L_2) \neq n \text{ and } \delta(\Sigma, \texttt{zero?}, L_2) \ni (\texttt{0}, \Sigma')$$
$$\delta(\Sigma, \texttt{div}, L_1, L_2) \ni (\texttt{err}_{\texttt{div}}, \Sigma') \qquad \text{if } \Sigma(L_2) \neq n \text{ and } \delta(\Sigma, \texttt{zero?}, L_2) \ni (\texttt{1}, \Sigma')$$

Figure 2.3: Selected Primitive Operations

tion when execution finishes.

Finally, we define the semantics to be the contextual closure of all the above reductions (rule *Close*). Errors halt the program and discard the context (rule *Error*).

2.3.3 Primitive Operations

We rely on relation δ to interpret primitive operations. The rules straightforwardly extend standard operators to work on symbolic values. In particular, division by an unknown denominator non-deterministically either returns another integer or raises an error. The relation also remembers appropriate refinements to arguments and results at each branch. Figure 2.3 presents a selection of representative rules for primitive operations `zero?` and `div`. We abbreviate `(`λ `(`X`)` `(=`XE`))` as `(`$\equiv E$`)`. Rules for primitive predicates such as `zero?` utilize a proof relation between the heap, the value, and a predicate, which we present next.

2.3.4 Proof Relation

We define a proof relation deciding whether a value satisfies a predicate. We write $\Sigma \vdash L : P \checkmark$ to mean the value at location L definitely satisfies predicate P, which implies that all possible instantiations of L satisfy P. In the same way, $\Sigma \vdash L : P \times$ means all instantiations of L definitely fail P. Finally, $\Sigma \vdash L : P ?$ is a conservative answer when we cannot draw a conclusion given information from existing refinements on the heap.

Precision of our execution relies on this proof relation. (A trivial relation answering "neither" for all queries would make the execution sound though highly imprecise.) Instead of implementing our own proof system, we rely on an SMT solver for sophisticated reasoning on base values.

Figure A.4 shows the translation $\{\!|\cdot|\!\}$ of run-time constructs into logical formulas. The translation of a heap is the conjunction of formulas obtained from each location and its value, and the translation of each location and value is straightforward. In particular, a location pointing to a concrete number translates to the obvious assertion of equality, and a mapping ($\texttt{case}\ \overrightarrow{L \mapsto L}$) adds constraints asserting that equal inputs imply equal outputs. Since outputs of maps may be functions, it might appear as though we need function equality. However, we do not need general equality on functions, but just a specialized equality that can handle those opaque functions generated by *AppOpq1*, *AppOpq2*, *AppOpq3* and *AppHavoc*. Equality on similar function forms proceeds structurally, while equality on different function forms translate trivially to `False` (not shown).

Notice that the proof system only needs to handle predicates of simple forms and not their arbitrary compositions. We rely on execution itself to break down complex predicates to smaller ones and take care of issues such as divergence and errors in the predicate itself. For example, if the proof system can prove that a value satisfies predicate P, it automatically allows the execution to prove that the value also satisfies `(λ (x) (or (`P `x) `E`))` for an arbitrarily expression E. By the time we have $[L \mapsto \bullet^{\overrightarrow{P}}]$, we can assume all predicates $\overrightarrow{P}$ have terminated with true on L. Further, because many solvers do not support uninterpreted higher-order functions, we do not assume such a feature, and the translation only produces queries on first-order values. Nevertheless, the symbolic execution itself can reason about higher-order unknown values. Handling higher-order functions on the semantics side and not relying on the theory of uninterpreted functions also potentially allows the method to scale to more realistic language features such as side effects.

For each query between heap Σ, location L and predicate P, we translate known assumptions from the heap to obtain formula ϕ, and the relationship $(L : P)$ to obtain formula ψ. We then consult the solver to obtain an answer. As figure 2.5 shows, validity of $(\phi \Rightarrow \psi)$ implies that value L definitely satisfies predicate P, and unsatisfiability of $(\phi \wedge \psi)$ means value L definitely refutes P. If neither can be determined, we return the conservative answer.

$$\{\!\{\overrightarrow{L \mapsto S}\}\!\} = \bigwedge \overrightarrow{\{\!\{L \mapsto S\}\!\}}$$
$$\{\!\{L \mapsto n\}\!\} = (L = n)$$
$$\{\!\{L \mapsto \bullet^{\mathtt{nat}\ \overrightarrow{P}}\}\!\} = \bigwedge \overrightarrow{\{\!\{L : P\}\!\}}$$
$$\{\!\{L \mapsto \mathtt{case}\ \_ \ldots [L_1 \mapsto L_2]\ \_ \ldots [L_3 \mapsto L_4]\ \_ \ldots\}\!\} = (\bigwedge\ ((L_1 = L_3) \Rightarrow (L_2 = L_4)) \ldots)$$
$$\{\!\{L : ((\lambda\ (X)\ (\mathtt{zero?}\ X)))\}\!\} = (L = 0)$$
$$\{\!\{L : ((\lambda\ (X)\ (= X (+\ L_1\ L_2))))\}\!\} = (L = (L_1 + L_2))$$
$$\{\!\{L_1 = L_2\}\!\}_{\mathtt{nat}} = (L_1 = L_2)$$
$$\{\!\{L_1 = L_2\}\!\}_{T \to T'} = \{\!\{\Sigma(L_1) = \Sigma(L_2)\}\!\}$$
$$\{\!\{(\mathtt{case}^T\ [L_1 \mapsto L_2] \ldots = \mathtt{case}^T\ [L_3 \mapsto L_4] \ldots)\}\!\} = (\bigwedge \{\!\{(L_1 = L_3) \Rightarrow (L_2 = L_4)\}\!\} \ldots)$$
$$\{\!\{(\lambda\ (X)\ L_1) = (\lambda\ (X)\ L_2)\}\!\} = \{\!\{L_1 = L_2\}\!\}$$
$$\{\!\{(\lambda\ (X)\ (L_1\ (X\ L_2))) = (\lambda\ (X)\ (L_3\ (X\ L_4)))\}\!\} = (\wedge\ \{\!\{L_1 = L_3\}\!\}\ \{\!\{L_2 = L_4\}\!\})$$
$$\{\!\{(\lambda\ (X)\ (\lambda\ (Y)\ ((L_1\ X)\ Y))) = (\lambda\ (X)\ (\lambda\ (Y)\ ((L_2\ X)\ Y)))\}\!\} = \{\!\{L_1 = L_2\}\!\}$$

Figure 2.4: Heap translation

$$\textsc{Proved}\ \frac{\{\!\{\Sigma\}\!\} \Rightarrow \{\!\{L : P\}\!\} \text{ is valid}}{\Sigma \vdash L : P\ \checkmark} \qquad \textsc{Refuted}\ \frac{\{\!\{\Sigma\}\!\} \wedge \{\!\{L : P\}\!\} \text{ is unsat}}{\Sigma \vdash L : P\ \text{✗}}$$

$$\textsc{Ambig}\ \frac{\{\!\{\Sigma\}\!\} \Rightarrow \{\!\{L : P\}\!\} \text{ is invalid and } \{\!\{\Sigma\}\!\} \wedge \{\!\{L : P\}\!\} \text{ is sat}}{\Sigma \vdash L : P\ \mathbf{?}}$$

Figure 2.5: Proof rules

2.3.5 Constructing Counterexamples

For each answer reached by evaluation, the heap contains refinements to symbolic values in order to reach such results. In particular, refinements on the heap in an error case describe the condition under which the program goes wrong.

Specifically, at the end of evaluation, refinements on the heap are nearly concrete: higher-order symbolic values are broken down into a chain of argument deconstruction and mappings, and first-order symbolic values have precise constraints that identify the execution path. Indeed, a model to the first-order constraints on the heap yields a counterexample to the program. We simply plug first-order concrete values back into the heap.

The reader may wonder if this process always generates an actual counterexample witnessing a real program bug (soundness), and if it always finds counterexample when a bug exists (completeness). The next section clarifies these points.

2.4 Soundness and Completeness of Counterexamples

We show that our method of finding counterexamples in a higher-order program is sound and relatively complete. Soundness means that the system only gives an actual counterexample triggering a bug (not a false positive). Relative completeness means that if the program actually contains a bug and the underlying solver can answer all queries on first order data, the system constructs a concrete counterexample witnessing that bug, even when it involves complex interactions between higher-order values.

The statements and proofs of soundness and completeness revolve around a notion of *approximation*, which we first describe before stating our main theorems.

Approximation Relation We define an approximation relation between *concrete* and *abstract* states. A concrete state contains no unknown values, while an abstract state may contain unknowns. We write $(E', \Sigma') \sqsubseteq (E, \Sigma)$ to mean "(E, Σ) *approximates* (E', Σ')," or conversely, "(E', Σ') *instantiates* (E, Σ)," where (E', Σ') is a concrete state and (E, Σ) is an abstract state. We make two remarks about the relation before defining it.

First, as discussed in section 2.3.2, when analyzing an incomplete program, we are only concerned with errors coming from known code. Therefore, we parameterize the approximation relation with a set of labels $\vec{L}$ denoting application sites from the known program portion. Figure 2.6 presents the straightforward definition of meta-function *lab* for computing a program's labels identifying application sites. The function takes a heap to compute labels for intermediate states, where a function may be referenced indirectly through its location. For the purpose of analyzing program E, the set of labels is $lab_{\emptyset}[\![E]\!]$. As an example, expression E below has an instantiation E', but when analyzing E, we are only interested in the potential division error at `L` and not `L'`.

$$E = \texttt{(div 1 (}\bullet^{\texttt{int}\to\texttt{int}}\texttt{ 1))}^{\texttt{L}}$$

$$E' = \texttt{(div 1 ((}\lambda\texttt{ (x) (div 1 x)}^{\texttt{L'}}\texttt{) 1))}^{\texttt{L}}$$

Second, we enforce that each location in the abstract state unambiguously approximates one location in the concrete state by parameterizing the approximation

relation with a function F mapping each label in the abstract state to one in the concrete state. For example, we do not want the following concrete state (E', Σ') to instantiate the abstract state (E, Σ), even though $\bullet^{\texttt{int}}$ intuitively approximates each number `1`, `2`, and `3` individually.

$$(E, \Sigma) \quad = (\texttt{(if L L L)} \quad , \{\texttt{L} \mapsto \bullet^{\texttt{int}}\})$$

$$(E', \Sigma') = (\texttt{(if } \texttt{L}_1 \texttt{ L}_2 \texttt{ L}_3\texttt{)}, \{\texttt{L}_1 \mapsto 1, \ \texttt{L}_2 \mapsto 2, \ \texttt{L}_3 \mapsto 3\})$$

Instead, the following concrete state (E'', Σ'') properly instantiates (E, Σ) with function $F = \{\texttt{L} \mapsto \texttt{L}_1\}$:

$$(E'', \Sigma'') = (\texttt{(if } \texttt{L}_1 \texttt{ L}_1 \texttt{ L}_1\texttt{)}, \{\texttt{L}_1 \mapsto 1\})$$

Figure 2.7 defines the approximation parameterized by label set $\vec{L}$ and function F. We present important, non-structural rules for the approximation relation between heaps, values, and states. We omit displaying parameters when they are unimportant or can be inferred from context. We defer the full definition to the appendix.

Rule *Heap-Ext* states that if a heap approximates a concrete heap, it approximates any extension of that concrete heap. This rule is necessary for ignoring irrelevant computations in instantiation of an opaque function. Next, rules *Heap-Int*, *Heap-Lam*, *Heap-Opq-1*, and *Heap-Opq-2* show straightforward extensions to the approximation when the heaps on both sides are extended. First, any concrete value of the right type instantiates the opaque value $\bullet^T$ as long as the instantiating value does not contain source locations from the known program portion. Second, refining an abstract value with a predicate known to be satisfied by the concrete

value preserves the approximation relation. Because a predicate can contain locations, we substitute labels appropriately as indicated by function F. We omit the straightforward definition of this substitution.

Rules *Heap-Case-1* and *Heap-Case-2* establish the approximation between functions on natural numbers. First, a fully opaque mapping approximates all functions. In addition, if there exists an execution trace witnessing that applying the concrete function yields a value approximated by an opaque value, then refining the mapping preserves the approximation.

Rule *Loc* states that location L approximates L' if the pair agrees with function F.

Rule *Err-Opq* reflects our decision of ignoring errors blaming source locations from unknown code. Otherwise, rule *Err* says that an error with a known label approximates another when they are the same error.

Finally, rule *Opq-App* states that we ignore irrelevant computation from a concrete function that instantiates an unknown function. Specifically, if we can establish that an opaque application approximates a concrete application by the structural rule, then the opaque application continues to approximate each non-answer state reachable from the concrete application. There are similar rules for approximation by applying other forms of opaque functions, which we defer to the appendix.

Theorem 1 states that every constructed counterexample from an error case actually reproduces the same error. Notice that the theorem is conditioned on $\Sigma' \sqsubseteq \Sigma_2$ and does not imply that all errors in the abstract execution are real. In

$$
\begin{array}{rcl}
lab_\Sigma[\![((O\,E))^L]\!] & = & \{L\} \cup lab_\Sigma[\![E]\!] \\
lab_\Sigma[\![(E_1\,E_2)]\!] & = & lab_\Sigma[\![E_1]\!] \cup lab_\Sigma[\![E_2]\!] \\
lab_\Sigma[\![(\mathsf{if}\ E\ E_1\ E_2)]\!] & = & lab_\Sigma[\![E]\!] \cup lab_\Sigma[\![E_1]\!] \cup lab_\Sigma[\![E_2]\!] \\
lab_\Sigma[\![(\lambda\ (X)\ E)]\!] & = & lab_\Sigma[\![E]\!] \\
lab_\Sigma[\![L]\!] & = & lab_\Sigma[\![\Sigma(L)]\!] \\
lab_\Sigma[\![\_]\!] & = & \emptyset
\end{array}
$$

Figure 2.6: Computing concrete labels

particular, if a path is spurious, its final heap has no instantiation.

Theorem 1 (Soundness of Counterexamples)**.**

If $(E, \Sigma_1) \longmapsto\!\!\!\!\rightarrow (\mathtt{err}^L_O, \Sigma_2)$ *and* $\Sigma' \sqsubseteq \Sigma_2$*, then* $(E, \Sigma') \longmapsto\!\!\!\!\rightarrow (\mathtt{err}^L_O, \Sigma'')$*.*

Theorem 2 states that we can discover every potential bug and construct a counterexample for it, assuming the underlying solver is complete for queries on first-order data.

Theorem 2 (Relative Completeness of Counterexamples)**.**

If $(E', \Sigma'_1) \longmapsto\!\!\!\!\rightarrow (\mathtt{err}^L_O, \Sigma'_2)$ *and* $(E', \Sigma'_1) \sqsubseteq_{\overrightarrow{L}} (E, \Sigma_1)$ *and* $L \in \overrightarrow{L}$*, then* $(E', \Sigma_1) \longmapsto\!\!\!\!\rightarrow$ $(\mathtt{err}^L_O, \Sigma_2)$ *such that there is an instantiation* Σ' *to* Σ_2*.*

We defer the proofs of theorems 1 and 2 to the appendix.

2.5 Extensions

We discuss important extensions to our system for a more practical programming language with dynamic typing, data structures, and contracts. In addition, we address the issue with termination. Our end goal is apply the method to realistic Racket [41] programs.

$$\textsc{Heap-Empty}\quad \emptyset \sqsubseteq^{\{\}}_{\overrightarrow{L}} \emptyset \qquad\qquad \textsc{Heap-Ext}\quad \frac{\Sigma' \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma}{\Sigma'[L' \mapsto S'] \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma}$$

$$\textsc{Heap-Int}\quad \frac{\Sigma' \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma}{\Sigma'[L' \mapsto N] \sqsubseteq^{F[L \mapsto L']}_{\overrightarrow{L}} \Sigma[L \mapsto N]}$$

$$\textsc{Heap-Lam}\quad \frac{\Sigma' \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma \qquad (E', \Sigma') \sqsubseteq^{F}_{\overrightarrow{L}} (E, \Sigma)}{\Sigma'[L' \mapsto (\lambda\ (X)\ E')] \sqsubseteq^{F[L \mapsto L']}_{\overrightarrow{L}} \Sigma[L \mapsto (\lambda\ (X)\ E)]}$$

$$\textsc{Heap-Opq-1}\quad \frac{\Sigma' \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma \qquad lab_{\Sigma'}[\![V']\!] \cap \overrightarrow{L} = \emptyset}{\Sigma'[L' \mapsto V'] \sqsubseteq^{F[L \mapsto L']}_{\overrightarrow{L}} \Sigma[L \mapsto \bullet^{T}]}$$

$$\textsc{Heap-Opq-2}\quad \frac{\Sigma'[L' \mapsto V'] \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma[L \mapsto \bullet^{TP\cdots}] \qquad \Sigma' \vdash V' : F(P_1)\ \checkmark}{\Sigma'[L' \mapsto V'] \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma[L \mapsto \bullet^{TP\cdots P_1}]}$$

$$\textsc{Heap-Case-1}\quad \frac{\Sigma' \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma \qquad lab_{\Sigma'}[\![E']\!] \cap \overrightarrow{L} = \emptyset}{\Sigma'[L' \mapsto (\lambda\ (X)\ E')] \sqsubseteq^{F[L \mapsto L']}_{\overrightarrow{L}} \Sigma[L \mapsto \mathtt{case}^{T}\ [\,]]}$$

$$\textsc{Heap-Case-2}\quad \frac{\begin{array}{c}\Sigma''[L' \mapsto (\lambda\ (X)\ E')] \sqsubseteq^{F}_{\overrightarrow{L}} \Sigma[L \mapsto \mathtt{case}^{T}\ [\ldots]] \\ F(L_x) = L'_x \qquad ([X/L'_x]E', \Sigma'') \longmapsto\!\!\!\!\rightarrow (V', \Sigma') \qquad (V', \Sigma') \sqsubseteq^{F}_{\overrightarrow{L}} (V, \Sigma)\end{array}}{\Sigma'[L' \mapsto (\lambda\ (X)\ E')] \sqsubseteq^{F[L \mapsto L']}_{\overrightarrow{L}} \Sigma[L \mapsto \mathtt{case}^{T}\ [\ldots L_x \mapsto V]]}$$

$$\textsc{Loc}\quad \frac{F(L) = L'}{(L', \Sigma') \sqsubseteq^{F}_{\overrightarrow{L}} (L, \Sigma)} \qquad \textsc{Err-Opq}\quad \frac{L' \notin \overrightarrow{L}}{(\mathtt{err}^{L'}_{O}, \Sigma') \sqsubseteq^{F}_{\overrightarrow{L}} (E, \Sigma)} \qquad \textsc{Err}\quad \frac{L' \in \overrightarrow{L}}{(\mathtt{err}^{L}_{O}, \Sigma') \sqsubseteq^{F}_{\overrightarrow{L}} (\mathtt{err}^{L}_{O}, \Sigma)}$$

$$\textsc{Opq-App}\quad \frac{\begin{array}{c}E' \neq \mathrm{A} \qquad lab_{\Sigma'}[\![E'']\!] \cap \overrightarrow{L} = \emptyset \qquad ((L'_f\ L'_x), \Sigma'') \sqsubseteq^{F}_{\overrightarrow{L}} ((L_f\ L_x), \Sigma) \\ \Sigma''(L'_f) = (\lambda\ (X)\ E'') \qquad \Sigma(L_f) = \bullet^{T \to T'} \qquad ([X/L'_x]E'', \Sigma'') \longmapsto\!\!\!\!\rightarrow (E', \Sigma')\end{array}}{(E', \Sigma') \sqsubseteq^{F}_{\overrightarrow{L}} (((L_f\ L_x)), \Sigma)}$$

Figure 2.7: Approximation

2.5.1 Dynamic typing

Dynamically typed languages defer safety checks to run-time to avoid conservative rejection of correct programs. Such languages have mechanisms for run-time inspection of data's type tag. We model this feature by extending primitive predicates with run-time type tests such as `integer?` or `procedure?`, which operate in the same manner as `zero?` in the typed language. Changes to the semantics are straightforward: we insert a run-time check into each application to ensure a function is begin applied, and into each primitive application to ensure arguments have the right tags. We also modify the rules for applying unknown functions, where previous static distinction in function types are turned to corresponding dynamic checks.

2.5.2 User-defined data structures

We extend the semantics to allow user-defined data structures, enabling programmers to express rich data such as lists and trees. Below is an example definition of a binary tree's node:

```
(struct node (left content right))
```

Each field in a data structure may itself be another data structure, function, or base value. Following the same treatment as functions, we do not encode data structures in the solver. Instead, we rely on execution to incrementally refine an unknown value's shape when knowing that it has a specific tag. For example, an unknown `node` has the shape of `(node` $\mathtt{L}_1$ $\mathtt{L}_2$ $\mathtt{L}_3$`)` where each of the fields $\mathtt{L}_1$, $\mathtt{L}_2$ and

L_3 is an unknown and refinable value. As before, we only need to encode constraints on base values at the leaves of data structures.

2.5.3 Contracts

Contracts generalize pre-and-post conditions to higher-order specifications [38], allowing programmers to express rich invariants using arbitrary code. They can either refine an existing type system [42] or ensure safety in an untyped language.

The following Racket [41] program illustrates the use of a higher-order contract. Function `argmin` requires a number-producing function as its first argument and a non-empty list as its second, and returns the list's element that minimizes the function. Here, contract `any/c` accepts any value, and combinator `and/c` returns the conjunction of its arguments.

```
; argmin : (any/c → number?) (and/c pair? list?) → any/c
(define (argmin f xs)
  (argmin/acc f (car xs) (f (car xs)) (cdr xs)))

(define (argmin/acc f b a xs)
  (cond
   [(null? xs) a]
   [(< b (f (car xs))) (argmin/acc f a b (cdr xs))]
   [else (argmin/acc f (car xs) (f (car xs)) (cdr xs))]))
```

Although the semantics of contract checking can be complex [43, 44], it introduces no new challenges in our system. We simply rely on the semantics of contract checking itself to break down complex and higher-order contracts into simple predicates. In addition, opaque flat contracts can be modeled soundly and precisely by rules for opaque application. Extension to the contract checking semantics enables our system to construct counterexamples to violated higher-order contracts.

2.5.4 Termination

The semantics presented so far does not guarantee termination. We can either accelerate (but not guarantee) termination by detecting recursion and widen values accordingly [18], or guarantee termination through systematic transformation of the semantics into a finite state or pushdown analysis of itself [24]. These techniques introduce spurious paths as over-approximations to actual execution branches. This affects both soundness and completeness of counterexamples. First, it requires more work to guarantee soundness. Because multiple concrete traces may be approximated to the same abstract trace, running the program with one instantiation of a constraint set may steer the program's flow to a different concrete trace that has the same abstraction. To ensure an instantiation corresponds to a real counterexample, it is necessary to first run the program with the concrete value set before reporting it as a counterexample. Second, relative completeness is also lost in practice. Even though execution still reveals every possible error, approximation results in a less precise constraint set for each trace, and the system may repeatedly query the solver for the wrong model before timing out. For example, a simplistic solver trying to refute that "`factorial(n + 4)` $\geq$ `10`" with no constraint may keep producing non-negative values for `n`.

Nevertheless, for our specific need of counterexample generation to refine an existing verification system (discussed next in section 2.6), we perform no abstraction. We rely on the previous system to prove the lack of counterexamples for a large set of correct programs [18] (therefore, many correct programs without coun-

terexamples do terminate). When used in combination with a verification system, abstracting the state space for counterexample generation is of little value, and makes it difficult to later concretize values to obtain a counterexample.

2.6 Implementation and evaluation

To evaluate our approach, we integrate counterexample generation into an existing contract verification system for programs written in a subset of Racket [41]. The system previously either successfully verified correct programs or conservatively reported probable contract violations and did not distinguish definite program errors from potentially false positives. With the new enhancement, the tool identifies a subset of reported errors as definite bugs with concrete counterexamples. Below, we describe implementation extensions, discuss promising experiment results, and address current difficulties.

2.6.1 Implementation

Our implementation and benchmarks can be found at

`https://github.com/philnguyen/soft-contract/tree/pldi-2015`

The prototype handles a much more realistic set of language features beyond SPCF. First, our implementation supports dynamic typing with user-defined structures and first class contracts as discussed in section 2.5. We also support more contract combinators such as conjunction, disjunction, and recursion. Second, we extend the set of base values and primitive operations, such as pairs, strings and Racket's full

numeric tower, which introduces more error sources and interesting counterexamples. Finally, we employ a module system to let users organize code. A module can export multiple values and define private ones for internal use.

Apart from being implemented as a command line tool, we also made a prototype available as a web REPL, accepting programs from anonymous users. The system attempts to verify correct programs and refute erroneous programs with concrete counterexamples. In some cases, it reports a probable contract violation without giving any counterexample due to limitations of the underlying solver, or the server simply times out after 10 seconds.

2.6.2 Evaluation

We collect benchmarks for our analysis from two sources: (1) prior published work and (2) submissions to the web REPL we built.

Benchmarks from prior work are drawn from research on higher-order model checking [16], dependent type checking [17], occurrence type checking [1], and our own work on contract verification [18]. Since these prior works focus on verification, the benchmarks are largely correct programs. In order to evaluate our counterexample generation technique, we modify each of the programs to introduce errors. To do so, we weakened preconditions and omitted checks before performing partial operations. For example, a resulting program may deconstruct a potentially empty list or compare potentially non-real numbers. We believe these changes are representative of common mistakes. A complete listing of the modifications is available.[3]

[3] `https://github.com/philnguyen/soft-contract/blob/pldi-2015/soft-contract/`

Benchmarks from our web service are submitted (anonymously) by users experimenting with the verification system. Many of these programs are buggy and we test how effective at discovering counterexamples.

In total, the evaluation is run on 282 programs consisting of 4050 lines of Racket code, excluding empty lines and comments. The largest programs are three student video games of the order of 250 lines. The test suite includes correct programs the system tries to verify as well as incorrect programs the system tries to generate counterexamples for.

We summarize our benchmark results in table 2.1. Each row shows the program's size (column 1), its highest function order (column 2), the time taken to verify the correct version of the program (column 3, if applicable), and the time taken to generate a counterexample refuting an incorrect variant of the program (column 4, if applicable). We compute each program's order by inspecting its contract's syntax (which is an under-approximation, because a contract may be dynamically computed). The last 3 rows "others", "others-e" and "others-w" summarize many small programs from our own benchmark suite as well as those collected from the server; we report their total, minimum and maximum line counts, total verification time, and highest function orders. With the exception of 5 programs in the last row "others-w", the system gives a counterexample for each incorrect program in a reasonable amount of time: the most complicated error takes 7 seconds to detect, and most errors in typical higher-order programs take less than 2 seconds. The last row shows benchmarks (all contributed by anonymous users) that reveal the limitation

`benchmark-verification/diff.txt`

of our counterexample generation in practice. In each of these cases, the system soundly reports a probable contract violation, but is unable to generate a counterexample confirming it. We discuss current shortcomings and language features known to thwart the tool in section 2.6.3.

The overall result is promising. First, there are specific examples where our prototype proves to be a good complement to random testing. For example, the tool finds a counterexample to the following program quickly and automatically:

```
f n = (/ 1 (- 100 n))
```

By default, QuickCheck does not find this error as it only considers integers from `-99` to `99`. Because QuickCheck treats a program as a black box, this conservative choice is reasonable for fear that the integer may be a loop variable causing the test case to run for a long time [45]. In contrast, our method explores the program's semantics symbolically and discovers `100` as a good test case.

Second, the resulting higher-order counterexamples suggest that the analysis can produce useful feedback. For example, it is easy for programmers to forget that Racket supports the full numeric tower [46] and that the predicate `number?` accepts complex numbers. The contract on `argmin` in section 2.5.3 is in fact too weak to protect the function. The system proves `argmin` unsafe by applying it to a specific combination of arguments. First, `f` is given a function that produces a non-real number, which causes `<` to signal an error. Second, `xs` is given a list of length 2, which is the minimum length to trigger a use of `<`.

```
f = (λ (x) 0+1i); xs = (list 0 0)
```

Finally, the tool analyzes the functional encoding of object-oriented programs effectively. Zombie is one such example with extensive use of higher-order functions to encode objects and classes, and the analysis can reveal errors buried in delayed function calls. We believe this is a promising first step for generating classes and objects as counterexamples. In the example below, we define interface `posn/c` that accepts two messages `x` and `y`, and function `first-quadrant?` that tests whether a position is in the first quadrant.

```
(define posn/c
  ([msg : (one-of/c 'x 'y)]
   → (match msg ['x number?] ['y number?])))

; posn/c → boolean?
(define (first-quadrant? p)
  (and (≥ (p 'x) 0) (≥ (p 'y) 0)))
```

The counterexample reveals one conforming implementation to interface `posn/c` that causes error in the module.

```
(λ (msg) (case msg [(x) 0+1i] [(y) 0]))
```

2.6.3 Difficulties

We discuss current difficulties to our approach and solutions in mitigating them.

First, the analysis is prone to combinatorial explosion as inherent in symbolic execution. Our tool finds bugs by performing a simple breadth-first search on the execution graph, then stops and reports on the first error encountered with a fully concrete counterexample. In practice, most conditionals come from case analyses

instead of independent alternatives, and we rely on a precise proof system to eliminate spurious paths. In addition, the modularity mitigates the problem further, as modules tend to be small, and contracts at boundaries help recovering necessary precision.

One major source of slowdown in our system is complex preconditions, where each input is guarded against a deep, inductively defined property. Execution follows different branches before being able to generate a valid input to continue verifying the module. A naive breadth-first search is bogged down by a large frontier resulting from different attempts to generate inputs, most of which are eventually found invalid. To mitigate this slow-down, we identify a class of expressions as likely to lead to counterexamples and prioritize their execution. Specifically, an expression whose innermost contract monitoring is of a first-order property on a concrete module is likely to reveal a bug.[4] In contrast, expressions in the middle of input generation do not have this form, because the inner-most contract monitoring is on the opaque input source. Once the system successfully instantiates a concrete input and turns the program into this "suspect" form, it focuses on exploring this branch with that input instead of trying numerous other inputs in parallel. Using this simple heuristic, we are able to cut the execution time of a module violating the "braun-tree" invariant from non-terminating after 1 hour down to 2 seconds.

Second, there is a mismatch in the data-types between Z3's data-type and Racket's rich numeric tower. In particular, Racket supports mixed arithmetic be-

[4]In a symbolic program, the monitored value in this position is usually abstract and covers all values the module produces.

tween different types of numbers up to complex numbers [46], while Z3's treatment of numbers resembles that from most statically typed languages, and the solver does not perform well in generating models involving a dynamic restriction of a number's type. Below is an example in the last row in table 2.1 where the tool fails to generate a counterexample:

```
; (integer? → integer?)
(define (f n) (/ 1 (+ 1 (* n n))))
```

In Racket, division is defined on the full numeric tower, and the result of `(/ 1 (+ 1 (* n n)))` may not be an integer. In the generated query, this result is an unknown number `L` of type `Real`, and the solver cannot give a model to a constraint set asserting "`(not (is_int L))`". In addition, Racket distinguishes between exact and inexact numbers, where inexact numbers are floating point approximations. Because Z3 does not reason about floating points, we currently do not soundly model inexact arithmetic.

2.7 Related work

We relate our work to four main lines of research: symbolic execution, counterexample guided abstraction refinement for dependent type inference, random testing, and contract verification.

First-order Symbolic Execution Symbolic execution on first-order programs is mature and has been used to find bugs in real-world programs [10,47]. [10] presents a symbolic execution engine for C that generates counterexamples of the form of

mappings from addresses to bit-vectors. Later work extends the technique to generate comprehensive test cases that discover bugs in large programs interacting with the environment [47].

Counterexample-guided Abstraction Refinement CEGAR has been used in model checking and dependent type inference [5, 16, 36], where the inference algorithm iteratively uses a counterexample given by the solver to refine preconditions attached to functions and values. In case the algorithm fails to infer a specification, the counterexample serves as a witness to a breaking input. Our work finds higher-order counterexamples only by integrating a first-order solver, and is applicable to both typed and untyped languages. In contrast, dependent type inference relies on an extension to ML. In addition, work on higher-order model checking [16, 35] analyzes complete programs with first-order unknown inputs, while we analyze partial programs with potentially higher-order unknown values at the boundaries.

Random Testing Random testing is a lightweight technique for finding counterexamples to program specifications through randomly generated inputs. QuickCheck for Haskell [19] proves the approach highly practical in finding bugs for functional programs. Later works extend random testing to improve code coverage and scale the technique to more language features such as states and class systems. [48] use contracts to guide random testing for Javascript, allowing users to annotate inputs to combine different analyses for increasing the probability of hitting branches with highly constrained preconditions. [34] also extend random testing to work on

higher-order stateful programs, discovering many bugs in object-oriented programs in Racket. [49] use refinement types as generators for tests, significantly improving code coverage.

Our approach is a complement to random testing. By combining symbolic execution with an SMT solver, the method takes advantage of conditions generated by ordinary program code and not just user-annotated contracts. In addition, the approach works well with highly constrained preconditions without further help from users. In contrast, random testing systems typically require programmers to implement custom generators [19] or require user annotations to incorporate a specific analysis collecting all literals in the program to guide input construction [48]. Type-targeted testing [49] is more lightweight and does not necessitate an extension to the existing semantics, but gives no guarantee about completeness, as inherent in random testing. Even though the tool rules out test cases that fail the pre-conditions, regular code and post-conditions do not help the test generation process. Our system makes use of both contracts and regular code to guide the execution to seek inputs that both satisfy pre-conditions and fail post-conditions. Exploring possible combination of symbolic execution and random testing for more efficient bug-finding in higher-order programs is our future work.

Contract Verification and Refinement Type Checking Contracts and refinement types are mechanisms for specifying much richer program invariants than those allowed in a typical type system. Verification systems either restrict the language of refinements to be decidable [5] or allow arbitrary enforcement but leave

unverifiable invariants as residual run-time checks [7, 9, 21, 23]. While verification proves the absence of errors but may give false positives, our tool aims to discover concrete, real counterexamples to faulty programs. Our work is a direct extension to previous work on symbolic execution of higher-order programs [9] and can be viewed as a complement to contract verification.

2.8 Conclusion

We have presented a symbolic execution semantics for finding concrete counterexamples in higher-order programs and proved it to be sound and relatively complete. An early prototype shows that the approach can scale to realistically sized functional programs with practical features such as first-class contracts. From the programmer's perspective, the approach is lightweight and requires no custom annotation to get started. However, if contracts are present, they can help guide the search for counterexamples. Combined with previous work on contract verification, it is possible to construct a tool that can statically guarantee contract correctness of programs and simultaneously ease the understanding of faulty programs, speeding up the development of reliable software.

Program	Lines	Order	Correct (ms)	Incorrect (ms)
Kobayashi et al. 2011 benchmarks				
fhnhn	18	2	38	50
fold-div	18	2	321	160
fold-fun-list	20	3	92	442
hors	25	2	49	34
hrec	9	2	52	143
intro1	13	2	24	128
intro2	13	2	25	127
intro3	13	2	25	23
isnil	9	1	13	9
max	14	2	32	135
mem	12	1	22	254
mult	9	1	61	147
nth0	15	1	19	296
r-file	50	1	74	123
r-lock	17	1	56	49
reverse	11	1	15	205
Terauchi 2010 benchmarks				
boolflip	10	1	10	22
mult-all	10	1	9	225
mult-cps	12	1	253	35
mult	10	1	72	21
sum-acm	10	1	33	833
sum-all	9	1	8	186
sum	8	1	44	19
Tobin-Hochstadt and Felleisen 2010 benchmarks				
occurrence (14)	116	1	99	226
Nguyên et al. 2014 benchmarks (video games)				
snake	164	1	37,350	2,476
tetris	267	2	11,809	2,188
zombie	249	3	19,239	954
Nguyên et al. 2014 other benchmarks and anonymous web submissions				
others (73)	(2 - 51) 818	3	20,465	-
others-e (124)	(3 - 23) 972	3	-	19,588
others-w (5)	(4 - 4) 20	1	-	431*

Table 2.1: Program verification and refutation time

Chapter 3: Soft Contract Verification for Higher-Order Stateful Programs

Software contracts allow programmers to state rich program properties using the full expressive power of an object language. However, since they are enforced at run-time, monitoring contracts imposes significant overhead and delays error discovery. *Soft contract verification* aims to guarantee all or most of these properties ahead of time, enabling valuable optimizations and yielding a more general assurance of correctness. Existing methods for static contract verification satisfy the needs of more restricted target languages, but fail to address the challenges unique to those conjoining untyped, dynamic programming, higher-order functions, modularity, and statefulness. Our approach tackles all these features at once, in the context of the full Racket system—a mature environment for stateful, higher-order, multi-paradigm programming with or without types. Evaluating our method using a set of both pure and stateful benchmarks, we are able to verify 99.94% of checks statically (all but 28 of $49,861$).

Stateful, higher-order functions pose significant challenges for static contract verification in particular. In the presence of these features, a modular analysis must permit code from the current module to escape permanently to an opaque

context (unspecified code from outside the current module) that may be stateful and therefore store a reference to the escaped closure. Also, contracts themselves, being predicates written in unrestricted Racket, may exhibit stateful behavior; a sound approach must be robust to contracts which are arbitrarily expressive and interwoven with the code they monitor. In this chapter, we present and evaluate our solution based on higher-order symbolic execution, explain the techniques we used to address such thorny issues, formalize a notion of behavioral approximation, and use it to provide a mechanized proof of soundness.

3.1 Static Contract Verification in a Stateful, Higher-order Setting

Software contracts [38, 50] allow programmers to provide rich specifications, using the full expressiveness of the host programming language, that are enforced dynamically. They have become a common mechanism for documenting and enforcing invariants in many dynamically typed and higher-order languages [51–55].

```
(define x 0)
(define/contract (f n)
  ((and/c int? (≥/c 0)) → (λ (n) (and/c int? (≥/c n))))
  (set! x (max x n))
  x)
```

To illustrate their use, consider the above function `f` written in Racket [41] that returns the largest natural number ever provided to it. Function `f` has a *dependent* contract enforcing that the function receives a single argument which must be a natural number and returns an integer no less than this argument. Here, the range "maker" `(λ (n) ...)` computes a range for each specific argument `n`, and contract

`(≥/c n)` accepts values no smaller than `n`.

While programmers may appreciate contracts for expressiveness and ease of use, as contracts are first-class values composable from arbitrary expressions, they have clear downsides: being checked dynamically delays error discovery and introduces non-trivial run-time overhead [54]. Static verification of contracts eliminates both disadvantages, verifying program components, discovering errors up front, and turning previously expensive dynamic checks into strong static guarantees with no run-time overhead. It aids programmer confidence in software correctness while justifying the removal of run-time monitors which, in turn, can enable further optimizations that the presence of interposed branches and calls prevented. *Soft* contract verification aims to soundly over-approximate program behavior, verifying contracts where possible and gracefully degrading by allowing them to be enforced dynamically otherwise.

Verification of contracts in a stateful, higher-order, and dynamically-typed language presents unique challenges:

- The idioms of dynamic languages thwart simple verification methods such as type inference [26], where programmers rely on dynamic type tests to justify uses of partial operations, with such tests being composed arbitrarily.

- Contracts are customarily used as enforcements at boundaries to ensure proper interaction between different components. Programmers tend not to write contracts for private functions, and rely on invariants established by inter-procedural and path-sensitive reasoning. A compositional analysis is unlikely

to succeed in verifying idiomatic dynamic programs without requiring heavy annotation from the programmer. Modularity, however, is crucial in scaling any analysis to a realistic code-base.

- Side effects complicate interactions between components through implicit communication channels. In particular, impure functions can escape the target module to be invoked an indeterminate number of times from an opaque context, possibly invalidating previously established invariants and triggering errors via interactions not possible in pure languages. Verification of effectful functions must soundly approximate such arbitrary interactions.

- Contracts themselves are arbitrary expressions capable of crashing, diverging, and modifying state, which prevents direct translation into pure functions and logical formulae for existing solvers.

Previous approaches to contract verification place greater restrictions on the language and contracts that limits applicability to a realistic programming environment. For example, they either rely on a static type system with contracts as function-level refinements [20–23], or assume the language is pure [18].

To advance this state of the art, we extend previous work on soft contract verification via higher-order symbolic execution, allowing mutable state in both concrete and symbolic functions. There were several crucial ingredients that made this possible. By employing a *path-condition* as standard in symbolic execution, our verification is inherently path-sensitive and capable of reasoning precisely on many idiomatic dynamically typed programs. By allowing symbolic values to be

higher-order and effectful, we achieve modularity, enabling the omission of arbitrary program components to trade between precision and complexity of analysis, while internally incurring no precision loss from programming abstractions such as private functions and sub-modules. By tracking values that have escaped to unknown contexts and reasoning conservatively about locations that may be mutated at arbitrary points, we make the verification robust against unknown effectful functions. Finally, by relying on an extended operational semantics to encompass the behavior of higher-order and stateful program features, and employing an SMT solver for proving only first-order properties of restricted run-time structures, verification scales to a realistic programming language with provable soundness while remaining capable of taking advantage of SMT solvers for sophisticated theorem proving.

The primary technical delta from prior work on soft contract verification [18] is the tracking of mutable functions and reasoning soundly about their possible effects. This required two extensions to our abstract semantics, we 1) maintained an over-approximation of the functions from transparent code that may have escaped to an unknown context, and then 2) extended our havoc semantics to simulate any sequence (of arbitrary order and length) of applying such functions at every point where control may escape to any unknown context.

Contributions

In this chapter, we make the following four contributions:

1. We give an extended operational semantics of a higher-order, stateful language that can be used for modular symbolic execution.

2. We use a tunable abstraction process, enforcing termination of the analysis at some cost to precision, coupled with a formal notion of behavioral over-approximation in the presence of unknown higher-order, stateful values and a mechanically verified proof of soundness.

3. We give a method for translating the symbolic execution history of a higher-order, stateful program into a pure, first-order formula, allowing the integration of a first-order SMT solver.

4. We implement and evaluate a practical contract verifier for a significant subset of Racket.

3.2 Examples

This section explains the essential ingredients of our verification approach using examples. The approach is based on symbolic execution, which extends an existing language with *symbolic values*, each standing for an unknown, but fixed, concrete value. Symbolic execution explores a set of paths through a program, maintaining a *path condition* along each to remember facts which must be true about symbolic values on that specific path. Each path condition is a formula, characterizing a particular path, that is strengthened incrementally as execution passes through the conditional branches that separates this path from all others.

By proving that some path conditions are contradictory and infeasible, the analysis is able to show that certain paths, and the errors along them, are unreachable.

If a symbolic execution were to explore all possible paths and terminate showing all errors to be unreachable, it would have performed a successful and complete verification that the program is statically free from run-time errors. In any real program, however, the number of distinct paths that may be explored is unbounded; in a traditional symbolic execution meant for finding bugs, imperfect coverage is just fine, for our purposes of static verification however, this unbounded set of paths must be soundly over-approximated. In addition, symbolic execution of higher-order functions requires simulating a program with unknown (opaque) functional values, i.e., we must reason about what happens when an unknown function is applied and the control path itself is unknown. In this section, we begin by discussing how symbolic execution can be used for program verification on a simple example, and then show its macro-expansion and a detailed view of our method on the program expanded to core forms (a smaller intermediate language). We then use further examples to discuss mutable state, effectful callbacks, and finitizing abstraction.

3.2.1 Path-sensitivity, SMT Solving, and Elaboration to Core Forms

Our first example demonstrates a function we can show is safe using path-sensitive reasoning over conditional control flow and first-order data—the basic building block of our verification. Being able to handle such programs is not unique to our process, but this example will allow us to illustrate the concepts underlying

our approach before addressing higher-order values and mutable state. Function `f` on the left column of Figure 3.1 expects a pair of a real number and a string, and promises to return a real number. The function pattern-matches on its argument before applying partial operations such as `str-len` and division. A complete verification assures that not only is `f` correct with respect to its explicit contracts, but that the function's uses of partial operations (such as `/` and `str-len`) are also safe, and that pattern matching covers all cases.

Verification of `f` via symbolic execution begins by applying `f` to a fresh symbolic value for its argument. All paths through `f` start with a path condition already constrained so that `f`'s argument matches the contract and is assumed to be a pair of a real number and string. Executing `f`'s body would then non-deterministically follow each branch of the `match` form, because all of them could be possible. At each branch, symbolic execution of `f` proceeds with a strengthened path-condition, remembering the constraints on data that may be assumed down this particular branch. For example, all paths reaching the second `match` clause will record in their path conditions that `x` cannot be a pair where the first value is less than or equal to 1. In this way, path-conditions encode the invariants that ensure the absence of run-time errors, allowing us to prove that `str-len` is only applied on strings or that `/` is given a non-zero divisor. By calling out to a dedicated SMT solver, the analysis proves that, in the second clause, `r` must be both a real number and also not a real number less than or equal to `1`, so therefore a non-zero real number. Down the implicit failure branch where Racket's `match` form would report an incomplete-pattern-match error, the path condition would report that `x` must be a

```
;; Original program
(define/contract (f x)
  ((cons/c real? str?) → real?)
  (match x
    [(cons r s)
     #:when (≤ r 1)
     (str-len s)]
    [(cons r s)
     (/ (str-len s) r)]))
```

```
;; Macro-expanded program
(define/contract (f x)
  ((cons/c real? str?) → real?)
  (let* ([x1 x]
         [k2 (λ () (error "incomplete"))])
    (if (cons? x1)
        (let* ([x1-car (car x1)]
               [x1-cdr (cdr x1)]
               [k1 (λ () (let ([r x1-car]
                               [s x1-cdr])
                           (/ (str-len s) r)))]
               [r1 x1-car]
               [s1 x1-cdr])
          (if (≤ r1 1) (str-len s1) (k1)))
        (k2))))
```

Figure 3.1: A pattern-matching example, before and after expansion.

pair of a real and string while also not a pair—a contradiction. The SMT solver will report that this path is infeasible and we have verified the match error can not occur. Ultimately, an exhaustive symbolic execution of just this component proves that `f` respects all its contracts (explicit or implicit), and no input can cause it to violate any of those.

Although the previous symbolic execution is straightforward for this example, it relies on knowledge of pattern-matching. Realistic programming languages can have a broad set of built-in features and, in the case of Racket, even user-definable syntax. Even `match` is simply a standard-library macro, with a pattern-matching facility that can be extended by user-defined macros. Building a verification process for a core language, after macro-expansion, allows better scaling of the process to large code bases using high-level language features, so long as the analysis can reason in terms of the fully-expanded intermediate language.

The program at the right column of Figure 3.1 shows the result of macro

expanding function `f` into core Racket. The expanded program is more complicated code in terms of simpler forms and is idiomatic of dynamically typed languages in that correctness relies on path-sensitive properties such as aliasing and numerical bounds. For example, the `match` macro generates an alias $\texttt{x}_1$ for `x`, and then a thunk $\texttt{k}_2$ for exiting with an error if the match fails. Similarly, $\texttt{r}_1$, `r`, and $\texttt{x}_1\texttt{-car}$ are aliases, and the program invokes thunk $\texttt{k}_1$, which performs a division on `r` only when its alias $\texttt{r}_1$ is greater than 1. Some thunks in the expansion of `f` would be unsafe if invoked from an arbitrary context; however, as this program uses them only in a correct way, our analysis proves that they are free from run-time errors. As our analysis constructs path conditions, refined across all dynamic checks, regardless of whether they are contracts specifically, no guard need be associated with these intermediate thunks. While in unexpanded Racket there are many language forms which may branch and refine the path condition, such as `if`, `match`, `case`, `cond`, `for`, `do`, etc, fully expanded Racket only has the `if` form. In addition, the many forms with complicated and unique control-flow behavior are compiled into administrative bindings, calls, returns, conditionals and continuations.

Verification of the expanded program proceeds in a manner similar to the original but needs to precisely track invariants about aliases and each closure's free variables. Variables in the analysis are bound to an *abstract* value that finitely approximates all possible run-time values (when it is entirely unknown, this is simply a • denoting a fully opaque, or unknown, value) paired with a *symbolic* value—a name that may be referenced in the path condition. The full mechanics of abstract and symbolic values is detailed and explained in Section 3.3. When execution reaches

the `let*` form (a sequential let-binding form), it assigns to $\texttt{x}_1$ the value that `x` holds, which is an opaque value named `x`. Instead of being bound to a specific pair of concrete values, as it would be in any real execution, `x` is bound to a fully opaque abstract value ($\bullet$) and a symbolic name, `x` (named for the original parameter), that is referenced in the path condition and constrained by it to be a pair of a real number and a string. At the `let*`-binding $[\texttt{x}_1\ \texttt{x}]$, this symbolic name propagates from `x` to $\texttt{x}_1$. The next binding evaluates the λ-term for a match error and assigns it to $\texttt{k}_2$. For closures, the analysis records both an abstract closure—the syntactic λ-term along with its abstract binding environment and the path condition at its creation (to constrain any free variables)—along with a symbolic name which is simply its syntactic λ-term.

The analysis then reaches a conditional to check if $\texttt{x}_1$ is a pair. In the "then" branch, the symbolic name `x` is assumed in the path condition to be a pair and any reference to variable $\texttt{x}_1\texttt{-car}$ or $\texttt{r}_1$ returns a symbolic value named `(car x)`, and for $\texttt{x}_1\texttt{-cdr}$ or $\texttt{s}_1$, the symbolic name `(cdr x)`. Symbolic names may refer to the value of any program expression—in this case, primitive operations. Most operations can be straightforwardly proven safe, except for the division in thunk $\texttt{k}_1$, where the path condition saved at the time of its creation did not assume that `r` was non-zero; this property was only recorded later before the application of $\texttt{k}_1$ when `r`'s alias $\texttt{r}_1$ was assumed greater than `1`. However, we treat the fact that $\texttt{k}_1$'s symbolic name is a λ-term as a signifier that the closure bound to $\texttt{k}_1$ was created within the caller. Assuming that the program has been α-renamed, any free variable shared between $\texttt{k}_1$ and the current scope must therefore denote the

```
(define/contract (f g)
  (((→ void?) → void?) → even?)
  (define n 2)
  (define double!
    (λ () (set! n (* 2 n))))
  (g double!)
  n)
```

(a) Effectful function escaping to unknown context (safe, verified).

```
(define/contract (f g)
  (((→ void?) → void?)
      → (and/c (≤/c 2) int?))
  (define n 0)
  (define inc! (λ () (set! n (+ 1 n))))
  (g inc!)
  (if (< n 2)
      (begin (g void) n)
      2))
```

(b) Effectful function escaping to unknown context (unsafe, unverified).

Figure 3.2: Symbolic functions and mutable state.

same value. This justifies strengthening $\texttt{k}_1$'s saved path-condition with the caller's invariants, establishing that `(car x)` is greater than `1` and proving the division in $\texttt{k}_1$ safe. This is a precision-enhancing technique discussed further in Section 3.4.1. In all, this process proves that the expanded version of `f` is also free of run-time errors.

3.2.2 Effectful Callbacks and Mutated Location Tracking

The next two examples demonstrate the verification of a higher-order function with mutable state. Function `f` in Figure 3.2a takes as its argument any function `g` that accepts a callback, and promises to return an even integer. Internally, `f` defines a mutable variable `n`, defines a callback `double!` that modifies `n` to its double, and passes the callback to its argument `g`. Although `g`'s behavior is arbitrary, it only causes modifications to `f`'s local state `n` in a controlled way through `double!`. Our analysis is able to prove that `n` is always an even number, regardless of `g`'s implementation, ensuring that `f`'s result respects its contract.

To verify `f` in Figure 3.2a, we apply it to a fully opaque symbolic value. Within `f`'s body, any reference to variable `g` returns a symbolic value named `g`. The semantics of higher-order contract monitoring wraps `g` in a contract promising that `f` only gives it a `void`-returning thunk, and ensuring that `g` only returns `void` [38]. Upon applying `g` to `double!` and transferring control to `g`, the execution simulates arbitrary computation in `g` capable of affecting `f`'s behavior and revealing its reachable errors. Specifically, `g` can both return a fully opaque symbolic value, and invoke (an arbitrary number of times) any function from `f` that has leaked to it (in this case, `double!`). Each update to `n` widens the value at `n` to approximate both old and new values. By choosing an appropriate domain of abstract values (e.g., one that preserves common predicates such as sign and parity tests) and providing a precise abstract interpretation for arithmetic over this domain, our analysis proves that `n` is always an even number regardless of how many times `double!` is invoked [56,57]. Returning `n` can thus be shown to satisfy `f`'s contract on its range and our symbolic execution verifies that `f` cannot be blamed for any violation of its contract. Section 3.4.1 describes the widening of values more generally.

Special care needs to be taken when a effectful callback escapes to an arbitrary context. For example, in Figure 3.2b, an opaque function `g` is invoked on `inc!`, a function that increments local variable `n`. The function, `f`, then tests to see if `n` remains strictly less than 2, in which case it invokes `g` on `void` and returns `n`, otherwise it returns 2. In the "then" branch, it may appear as though the call to `(g void)` may not effect `n` since the `void` function has no effect. However, an implementation of `g` that satisfies the contract can save the earlier reference to

`inc!` and invoke it again many times when `g` is applied to `void`, invalidating any assumption about `n`'s upper-bound.

3.2.3 Abstracting Symbolic Execution

Traditionally, symbolic execution is used to find potential errors, not to verify programs. Symbolic execution explores a number of program paths precisely but does not typically provide a terminating over-approximation of all possible program paths [47, 58–61]. As described thus far, our process would not terminate on many programs. Consider `factorial`, shown in Figure 3.3: execution repeatedly unfolds `factorial` at each recursive call, applying the function to a fresh symbolic integer.

```
(define/contract (factorial z)
  ((and/c int? (≥/c 0)) → (and/c int? (≥/c 1)))
  (if (≤ z 1)
      1
      (* z (factorial (- z 1)))))
```

Figure 3.3: Factorial

To ensure termination, we apply a well-studied method for systematically abstracting an operational semantics through finitization of dynamic program components [24]. The method yields many choices for finitizing the structure of an abstract machine [62] that are sound, and permits a method for tuning poly-variance to trade-off between precision and performance by changing the machine's allocation behavior [63]. Our instantiation of this framework approximates recurring values and path-conditions at different iterations of the same loop, summarizing repeated values and properties of data as loop invari-

$$
\begin{array}{rl}
\text{[Expressions]} & E ::= U \mid X \mid (E\ E)^L \mid (\mathsf{if}\ E\ E\ E) \mid (\mathsf{set!}\ X\ E) \\
& \quad \mid (E \to (\lambda\ (X)\ E)) \mid (\mathtt{mon}_L^L\ E\ E) \\
\text{[Value Literals]} & U ::= (\lambda\ (X)\ E) \mid N \mid O \mid \bullet \\
\text{[Integers]} & N ::= \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
\text{[Primitives]} & O ::= \mathtt{int?} \mid \mathtt{proc?} \mid \mathtt{zero?} \mid \mathtt{flat\text{-}contract?} \mid \mathtt{add1} \mid \ldots \\
\text{[Variables]} & X, Y = \langle \text{identifiers} \rangle \\
\text{[Labels]} & L = \langle \text{identifiers} \rangle
\end{array}
$$

Figure 3.4: Syntax of λ_S.

ants, while providing exact execution for loop-free program fragments. We describe our implementation in detail in Section 3.4.

In the case of `factorial`, execution branches on `(≤ z 1)`, yielding two paths, and learns that `factorial` either returns `1` or multiplies `z` with the result of its recursive call (knowing that `z` is greater than `1`). The recursive call similarly returns `1` in one of its branch, yielding `(* z 1)` as a result of the parent call to `factorial`. Through finitization of dynamically generated program components, the analysis reaches a fixed point, learning an over-approximation of `factorial`'s behavior: it either returns `1`, or the product of its argument `z` $\geq$ `1` with an integer no less than `1`. Across all cases, the solver can verify that `factorial` satisfies its contracts.

3.3 Static verification through symbolic execution

We present our symbolic execution using language λ_S, an untyped lambda calculus extended with mutable state and first-class higher-order contracts. The language's grammar is given in Figure 3.4.

Apart from standard values such as λ-abstractions `(λ (`X`) `E`)`, natural numbers (N), and primitive operations (O), λ_S includes symbolic values ($\bullet$), each standing for an arbitrary value that is syntactically closed. (For example, $\bullet$ cannot stand for `(λ (x) y)` because `y` is free.)

Most forms are standard, such as variable reference (X), conditional `(if`EEE`)`, and variable mutation `(set!` X E`)`. We annotate each function application `(`E E`)`^{L} with a source label L to serve as a potential party to blame in case of a contract failure.

Contracts are first-class values in λ_S and belong to the same syntactic category as expressions. Expression `(`E $\rightarrow$ `(λ (`X`) `E'`))` denotes a higher-order dependent contract, which is a pair of contract domain E and range "maker" `(λ (`X`) `E'`)` that computes a range E' dependent on the argument bound to X for each specific function application. For example,

```
(int? → (λ (x) (λ (a) (and (int? a) (> a x)))))
```

is a contract for functions that map each integer to a greater integer.

Finally, the monitoring form `(mon`$^L_{L'}$ E E'`)` denotes the dynamic enforcement of contract E between expression E' and its surrounding context, where L is the party to blame if E' produces a value that fails the contract, and L' is to blame if the context consuming the result of E' uses the result in a way that violates the contract. For example, `(mon`$^L_{L'}$ `int? add1)` evaluates to a blame on L for producing the function `add1` instead of an integer, and the application

$((\texttt{mon}^{L}_{L'}\ (\texttt{int?} \to \texttt{int?})\ \texttt{add1})\ \ \texttt{proc?})$ evaluates to a blame on L' for supplying the primitive function `proc?` to a guarded function `add1` expecting an integer.

Our language λ_S is minimal, but models core features found in practical programming languages. For example, pattern matching can be expanded into simple conditionals as shown in Section 3.2.1, and mutable boxes can be modeled using closures and mutable variables, as demonstrated in Figure 3.5.

```
(define (box x)
  (λ (msg)
    (match msg
      ['set-box! (λ (w) (set! x w))]
      ['unbox x])))
(define (set-box! b v) ((b 'set-box!) v))
(define (unbox b) (b 'unbox))
```

Figure 3.5: Mutable box as closures.

3.3.1 Semantics

We define the semantics of λ_S using a reduction relation over machine states, ς, each constituted of four components as shown in Figure 3.6.

At a high level, our abstract machine is just a closure-creating, store-passing interpreter, factored into several small-step rules and a single big-step rule that steps a configuration within an evaluation context (see Figure 3.7). We then instrument this machine with several extra components. In addition to a configuration and store, each state tracks a store-cache (mapping variables to locally precise values and their symbolic names) and a path condition (accumulating a conjunction of facts, in terms of these symbolic names, that characterizes the current execution

$$
\begin{array}{rl}
\text{[States]} & \varsigma ::= (C, M, \Phi, \Sigma) \\
\text{[Closures]} & C ::= \mathrm{A} \mid (E, \mathrm{P}) \mid (C\ C)^L \mid (\mathsf{if}\ C\ C\ C) \mid (\mathsf{set!}\ (X, \mathrm{P})\ C) \\
& \quad \mid (C\ \rightarrow\ ((\lambda\ (X)\ E), \mathrm{P})) \\
& \quad \mid (\mathtt{mon}_L^L\ C\ C) \mid (\mathsf{rt}_X^{\overrightarrow{X}}\ S\ M\ \Phi\ C) \\
\text{[Evaluation Contexts]} & \mathcal{E} ::= [\,] \mid (\mathcal{E}\ C)^L \mid (W\ \mathcal{E})^L \mid (\mathsf{if}\ \mathcal{E}\ C\ C) \mid (\mathsf{set!}\ (X, \mathrm{P})\ \mathcal{E}) \\
& \quad \mid (\mathcal{E}\ \rightarrow\ ((\lambda\ (X)\ E), \mathrm{P})) \\
& \quad \mid (\mathtt{mon}_L^L\ \mathcal{E}\ C) \mid (\mathtt{mon}_L^L\ W\ \mathcal{E}) \mid (\mathsf{rt}_X^{\overrightarrow{X}}\ S\ M\ \Phi\ \mathcal{E}) \\
\text{[Answers]} & \mathrm{A} ::= W \mid \mathtt{blame}_L^L \\
\text{[Post-values]} & W ::= (V, S) \\
\text{[Values]} & V ::= N \mid O \mid \bullet \mid \mathsf{Clo}(X, E, \mathrm{P}, \Phi) \mid \mathsf{Grd}(\alpha, \alpha) \mid \mathsf{Arr}_{\mathsf{L}}^{\mathsf{L}}(\alpha, \alpha) \\
\text{[Symbols]} & S ::= E \mid \emptyset \\
\text{[Path-conditions]} & \Phi = \mathcal{P}(Exp) \\
\text{[Address]} & \alpha = \langle \text{any enumerable set, e.g. } \mathbb{N} \rangle \\
\text{[Store-cache]} & M = \mathsf{Var} \rightarrow (\,WVal + \{\emptyset\}) \\
\text{[Environments]} & \mathrm{P} = \mathsf{Var} \rightarrow Addr \\
\text{[Value stores]} & \Sigma = Addr \rightarrow \mathcal{P}(\,Val)
\end{array}
$$

Figure 3.6: State components for symbolic execution.

path). We also include higher-order contracts as distinguished values, allow labeled blame objects to result from evaluation, and track strongly updated variables.

3.3.1.1 State Components

Each machine state consists of four components:

1. A closure (C) is either an answer (A), an expression (E) paired with an environment (P), or an inductively defined closure whose structure mimics that of plain expressions (E).

2. A store-cache (M) is a finite map from each variable, X, in scope to either a value that X must hold, or the special symbol $\emptyset$ indicating that X may have been modified arbitrarily.

3. A path condition (Φ) is a set (interpreted as a conjunction) of expressions assumed to have evaluated to true.

4. A store (Σ) that maps each address (α) to a set of values ($\overrightarrow{V}$). For a standard concrete execution that is deterministic, each value set is a singleton. We generalize the store's range to be a set, however, to allow modeling non-determinism resulting from over-approximation of multiple execution paths.

Several further sub-components constitute these. Binding environments (P) map each variable in scope to an address (α) in the value store. An answer (A) is either a post-value (W) or an error ($\texttt{blame}^{L}_{L'}$) blaming component labeled L for violating the contract with L'. A post-value (W) is a value (V) paired with a

symbolic name (S) which relates this value to relevant constraints in the current path condition. The special symbol $\emptyset$ indicates the lack of a symbolic name to appear in the path condition.

A value (V) is either a number (N), primitive operator (O), function closure ($\mathsf{Clo}(X, E, \mathrm{P}, \Phi)$), higher-order contract ($\mathsf{Grd}(\alpha, \alpha)$), guarded function ($\mathsf{Arr}^{\mathsf{L}}_{\mathsf{L}}(\alpha, \alpha)$), or abstract value ($\bullet$). We use `0` to represent falsehood, and any other value to represent truth. For convenience, we assume `zero?` is a total predicate in λ_{S} that tests for falsehood, and `nonzero?` is its complement. Similarly, `flat-contract?` tests if a value is usable as a flat contract (e.g. either a primitive function or a lambda), and `dep-contract?` tests if a value is a dependent contract.

The grammar for evaluation contexts ($\mathcal{E}$) follows that of closures (C) and enforces a standard left-to-right call-by-value semantics. In particular, we evaluate functions before arguments, and contracts before monitored expressions.

The semantics is mostly standard with the addition of a store-cache and a path-condition that tracks path-sensitive information about locations and symbolic values, respectively. For example, an entry $\texttt{x} \mapsto (\bullet, (+\ \texttt{y}\ \texttt{n}))$ in the store-cache means that location `x` holds the symbolic value `(+ y n)`, and expression `(> x n)` in the path-condition indicates the constraint on symbolic values `x` and `n`. The name `x`, when it appears as a symbol, refers to the value first bound to location `x` when `x` is in scope.

3.3.1.2 Reduction relation

We define the small-step operational semantics using reduction relation ($\longmapsto$), which defines the evaluation of a machine state as a sequence of atomic steps. The relation ($\longmapsto$) is defined as the context-closure of relation ($\longmapsto_v$) over redexes, as shown in figure 3.7. We present each aspect of this reduction relation in turn.

Some rules (e.g. *[Grd]*, *[AppClo]*, *[MonFun]*) involve allocating a value in the store at some address α—we leave the choice of address allocation open, because any allocation results in a sound over-approximation of the standard concrete semantics [24, 63]. As we will see in Section 3.3.5, different allocation choices decide whether the semantics is a traditional bug-finding symbolic execution or static verification with different trade-offs between precision and termination.

DISTRIBUTION OF ENVIRONMENT INTO SUB-EXPRESSIONS Rule *[Distr]* in Figure 3.8 shows the reduction of closures of the form (E, P), where E contains one or more sub-expressions. The (partial) meta-function distr shows the straightforward definition of distributing environments into sub-expressions.

VALUES Rule *[Lit]* in Figure 3.9 shows the reduction for value literals. Each of these expressions evaluates to an answer determined by meta-function lit. The definition of lit is straightforward for base values. For each λ-abstraction, the meta-function saves the current path-condition to remember invariants about free variables, in addition to the environment as standard. Rule *[Grd]* shows the reduction of a higher-order contract once its domain is evaluated: the reduction steps to a

contract object with both its domain and range-maker components allocated in the store.

VARIABLE REFERENCING AND MUTATION Figure 3.9 also shows reduction rules for referencing and mutating variables. Rule *[Var]* references the value at variable X by performing a lookup in the store-cache (M) as well as in the store (Σ). As shown in the definition of relation lookup, we either use the cache result if the cache indicates a definite hit, or look up in the store as standard if the cache indicates that the variable has stopped being tracked precisely. Rule *[Set]* strongly updates the store-cache and weakly updates the store.

CONDITIONALS The last rules in Figure 3.9 shows reduction for conditionals. If the evaluated condition is plausibly non-0, execution steps to the "then" branch of the conditional and refines the path-condition to reflect the new assumption (rule *[CondTrue]*). Here, relation $\mathsf{feasible}(\Phi, O, W, \Phi')$ guards the rule, ensuring that value W could possibly satisfy predicate O, given the current invariants in path-condition Φ, and remembering the branch condition as an assumption of a strengthened path-condition Φ'. If the condition is plausibly 0, execution steps to the "else" branch of the conditional and refines the path condition to reflect the inverse assumption (rule *[CondFalse]*). When both branches are plausible, execution non-deterministically steps to both, each with the appropriately strengthened path condition.

$$\frac{(C_1, M_1, \Phi_1, \Sigma_1) \longmapsto_v (C_2, M_2, \Phi_2, \Sigma_2)}{(\mathcal{E}[C_1], M_1, \Phi_1, \Sigma_1) \longmapsto (\mathcal{E}[C_2], M_2, \Phi_2, \Sigma_2)}$$

Figure 3.7: Reduction on states.

$$((E, \mathrm{P}), M, \Phi, \Sigma) \;\; \longmapsto_v \;\; (C, M, \Phi', \Sigma) \quad \textit{[Distr]}$$
$$\text{if} \qquad \mathsf{distr}(E, \mathrm{P}) = C$$

$$\begin{aligned}
\text{where} \qquad \mathsf{distr}(\texttt{(}E_1\ E_2\texttt{)}^L, \mathrm{P}) &= \texttt{(}(E_1, \mathrm{P})\ (E_2, \mathrm{P})\texttt{)}^L \\
\mathsf{distr}(\texttt{(if }E\ E_1\ E_2\texttt{)}, \mathrm{P}) &= \texttt{(if }(E, \mathrm{P})\ (E_1, \mathrm{P})\ (E_2, \mathrm{P})\texttt{)} \\
\mathsf{distr}(\texttt{(set! }X\ E\texttt{)}, \mathrm{P}) &= \texttt{(set! }(X, E)\ (E, \mathrm{P})\texttt{)} \\
\mathsf{distr}(\texttt{(}E\ \rightarrow\ \texttt{(}\lambda\ \texttt{(}X\texttt{)}\ E'\texttt{))}, \mathrm{P}) &= \texttt{(}(E, \mathrm{P})\ \rightarrow\ \texttt{((}\lambda\ \texttt{(}X\texttt{)}\ E'\texttt{),}\mathrm{P}\texttt{))} \\
\mathsf{distr}(\texttt{(mon}^L_{L'}\ E\ E'\texttt{)}, \mathrm{P}) &= \texttt{(mon}^L_{L'}\ (E, \mathrm{P})\ (E', \mathrm{P})\texttt{)}
\end{aligned}$$

Figure 3.8: Distribution of environment.

$$((U, \mathrm{P}), M, \Phi, \Sigma) \longmapsto_v (\mathsf{lit}(U, \mathrm{P}, \Phi), M, \Phi, \Sigma) \quad \textit{[Lit]}$$

$$((\lambda\ (X)\ E), \mathrm{P}), M, \Phi, \Sigma) \longmapsto_v ((\mathsf{Grd}(\alpha_1, \alpha_2), \emptyset), M, \Phi, \Sigma') \quad \textit{[Grd]}$$
where for some addresses α_1, α_2 $\quad \Sigma' = \Sigma \sqcup [\alpha_1 \mapsto V, \alpha_2 \mapsto \mathsf{Clo}(X, E, \mathrm{P}, \Phi)]$

$$((X, \mathrm{P}), M, \Phi, \Sigma) \longmapsto_v (W, M, \Phi, \Sigma) \quad \textit{[Var]}$$
where $\mathsf{lookup}(\Sigma, \mathrm{P}, M, X) \ni W$

$$((\mathsf{set!}\ (X, \mathrm{P})\ (V, S)), M, \Phi, \Sigma) \longmapsto_v ((1, \emptyset), M', \Phi, \Sigma') \quad \textit{[Set]}$$
where $m' = m[X \mapsto (V, S)]$ $\quad \Sigma' = \Sigma \sqcup [\mathrm{P}(X) \mapsto V]$

$$((\mathsf{if}\ W\ C_1\ C_2), M, \Phi, \Sigma) \longmapsto_v (C_1, M, \Phi', \Sigma) \quad \textit{[CondTrue]}$$
where $\mathsf{feasible}(\Phi, \texttt{nonzero?}, W, \Phi')$

$$((\mathsf{if}\ W\ C_1\ C_2), M, \Phi, \Sigma) \longmapsto_v (C_2, M, \Phi', \Sigma) \quad \textit{[CondFalse]}$$
where $\mathsf{feasible}(\Phi, \texttt{zero?}, W, \Phi')$

---

where $\mathsf{lit}(N, \_, \_) = (N, N)$ $\quad \mathsf{lit}(O, \_, \_) = (O, O)$
$\mathsf{lit}(\bullet, \_, \_) = (\bullet, \emptyset)$ $\quad \mathsf{lit}((\lambda\ (X)\ E), \mathrm{P}, \Phi) = (\mathsf{Clo}(X, E, \mathrm{P}, \Phi), (\lambda\ (X)\ E))$

where $\mathsf{lookup}(\Sigma, \mathrm{P}, M, X) \ni W$, if $M(X) = W$
$\mathsf{lookup}(\Sigma, \mathrm{P}, M, X) \ni (V, \emptyset)$, if $M(X) = \emptyset$ and $V \in \Sigma(\mathrm{P}(X))$

Figure 3.9: Reduction on values, variables, mutation, and conditionals.

$$((\mathtt{mon}^L_{L'}\ W'\ W), M, \Phi, \Sigma) \longmapsto_v ((\mathsf{if}\ (W'\ W)\ W\ \mathtt{blame}^L_{L'}), M, \Phi', \Sigma) \quad \textit{[MonFlat]}$$
where $\mathsf{feasible}(\Phi, \texttt{flat-contract?}, W', \Phi')$

$$((\mathtt{mon}^L_{L'}\ W'\ W), M, \Phi, \Sigma) \longmapsto_v ((\mathsf{if}\ (\texttt{proc?}\ W)\ \mathsf{Arr}^{\mathsf{L}}_{\mathsf{L}'}(\alpha_1, \alpha_2)\ \mathtt{blame}^L_{L'}), M, \Phi', \Sigma') \quad \textit{[MonFun]}$$
where $\mathsf{feasible}(\Phi, \texttt{dep-contract?}, W', \Phi')$ $\quad (V, \_) = W$ $\quad (V', \_) = W'$
and for some addresses α_1, α_2, $\quad \Sigma' = \Sigma \sqcup [\alpha_1 \mapsto V', \alpha_2 \mapsto V]$

Figure 3.10: Reduction on contract monitors.

CONTRACT MONITORING Figure 3.10 shows reduction rules for monitoring contracts.

Rule *[MonFlat]* shows the straightforward monitoring of a flat contract—the contract is simply applied on the value as a predicate. If the value passes this predicate, it is returned as-is; otherwise, a blame on the party providing the value is raised.

Rule *[MonFun]* shows the monitoring of a higher-order contract, which first performs a first-order check ensuring the target is indeed a function, blaming the party providing the value if it is not. If the value is indeed a function, monitoring saves the higher-order contracts, the function being checked, along with the blame parties into a guarded function to perform checks at each subsequent application, following the semantics of monitoring higher-order contracts. The details of applying a guarded function are described later in application rule *[AppArr]*.

APPLICATION Figure 3.11 shows reduction rules for application. Values that can be used as functions in λ_S are primitive operations, closures, and guarded functions, whose applications are shown in rules *[AppPrim]*, *[AppClo]*, and *[AppArr]*, respectively. Applying an opaque value results in two possibilities covered in rule *[AppOpq]*.

Application of primitive operations rely on relation δ as in rule *[AppPrim]*. We generalize δ to a relation instead of meta-function to express non-determinism in the presence of symbolic values. The result's symbolic name is created through meta-function `ap`, which reconstructs the application, except returning $\emptyset$ if either

arguments is $\emptyset$.

Rule *[AppClo]* governs application of a closure and allocates the argument in the store at an address α. Because some variables have new visible bindings, the store-cache and path-condition need to be updated. The target closure's parameter (X) now refers to a distinct location from the caller's X (if it exists), so it receives a fresh entry in the store-cache pointing to the argument's value as well as X as the symbolic name. The closure's free variables, on the other hand, may or may not refer to the same locations as they do at the call site, so their store-cache entries are simply invalidated and treated conservatively. The closure application reduces to the function body with the extended environment and store as standard, but also saves the caller's store-cache, path-condition, and symbolic name to reinstate upon a return.

Rule *[AppArr]* describes the application of a guarded function, which is decomposed into the monitoring of the argument against the contract's domain with reversed blame parties, followed by the application of the function under guard, whose result is in turn monitored against the computed contract range. If the function's guarding contract is not a concrete higher-order contract, it decomposes into opaque domain and range contracts.

Rule *[AppOpq]* describes two non-deterministic cases that result from applying an opaque function. Because a blame on an unknown program component is an irrelevant analysis result, we ignore unknown functions that introduce errors of their own. A well-behaved function, on the other hand, interacts with the rest of the program in limited ways: it either returns a value, or if its argument V is

a function, applies V to some value. Because each application of V may modify program state, and the effect of each application (*e.g.*, whether it triggers an error) may depend on the program state resulting from applying a stateful function (either V itself or another function that has previously escaped from transparent code to unknown code), arbitrary repetition of this application needs to be considered. We therefore maintain a set of values that have escaped to unknown code at a special address $\bullet$, and upon invocation of an opaque function, we emulate an arbitrary number of invocations of escaped code before finally returning an opaque value as a result. The first case of rule *[AppOpq]* approximates all possible returns from the unknown function by returning an opaque value. The second case of rule *[AppOpq]* approximates all possible applications of the unknown function by applying any value from the transparent code that has "leaked" into the unknown code (including the argument from the latest opaque application, V), and then passes the result back into an opaque function.

$$((((O, S')\,(V, S))\texttt{)}^{L}, M, \Phi, \Sigma) \;\longmapsto_v\; ((V', \mathsf{ap}(S', S)), M, \Phi, \Sigma) \qquad \textit{[AppPrim]}$$
where $V' \in \delta(\Sigma, O, V)$

$$((((\mathsf{Clo}(X, E, \mathrm{P}, \Phi), S_f)\,(V, S))\texttt{)}^{L}, M, \Phi, \Sigma) \;\longmapsto_v\; ((\texttt{rt}_X^{\overrightarrow{Y}}\ S'\ M\ \Phi\ (E, \mathrm{P}'))\texttt{)}, M', \Phi', \Sigma') \qquad \textit{[AppClo]}$$
where $\overrightarrow{Y} = dom(\mathrm{P}) \qquad W = (V, S) \qquad M' = M[X \mapsto (V, X)][\overrightarrow{Y \mapsto \emptyset}] \qquad S' = \mathsf{ap}(S_f, S)$
and for some address α, $\mathrm{P}' = \mathrm{P}[X \mapsto \alpha] \qquad \Sigma' = \Sigma \sqcup [\alpha \mapsto V]$

$$((\mathsf{Arr}^{\mathsf{L}}_{\mathsf{L}'}(\alpha_c, \alpha_f)\ W)\texttt{)}^{L}, M, \Phi, \Sigma) \;\longmapsto_v\; ((\texttt{mon}^{L}_{L'}\ W_r\ (W_f\ (\texttt{mon}^{L'}_{L}\ W_d\ W)))\texttt{)}, M, \Phi, \Sigma) \qquad \textit{[AppArr]}$$
where $\mathsf{Grd}(\alpha_d, \alpha_r) \in \Sigma(\alpha_c) \qquad V_d \in \Sigma(\alpha_d) \qquad V_r \in \Sigma(\alpha_r) \qquad V_f \in \Sigma(\alpha_f)$
$W_d = (V_d, \emptyset) \qquad W_r = (V_r, \emptyset) \qquad V_f = (V_f, \emptyset)$

$$((\texttt{(}\bullet, S')\,(V, S))\texttt{)}^{L}, M, \Phi, \Sigma) \;\longmapsto_v\; \begin{cases} (W_\bullet, M, \Phi', \Sigma') \\ ((W_\bullet\ (W_i\ W_\bullet))\texttt{)}, M, \Phi', \Sigma') \end{cases} \qquad \textit{[AppOpq]}$$
where $\mathsf{feasible}(\Phi, \texttt{proc?}, (\bullet, S'), \Phi') \qquad \Sigma' = \Sigma \sqcup [\alpha_\bullet \mapsto \{V\}]$
$V_i \in \Sigma'(\alpha_\bullet) \qquad W_\bullet = (\bullet, \emptyset) \qquad W_i = (V_i, \emptyset)$

$$((W'\ W)\texttt{)}^{L}, M, \Phi, \Sigma) \;\longmapsto_v\; (\texttt{blame}^{L}_{\Lambda}, M, \Phi', \Sigma) \qquad \textit{[AppErr]}$$
where $\mathsf{feasible}(\Phi, \texttt{nonproc?}, W', \Phi')$

Figure 3.11: Reduction on application sites.

$$(\texttt{(rt}_X^{\overrightarrow{Y}}\ S'\ M'\ \Phi'\ (V, S))\texttt{)}, M, \Phi, \Sigma) \;\longmapsto_v\; ((V, S''), M'', \Phi, \Sigma) \qquad \textit{[Ret]}$$
where $M'' = M[X \mapsto M'(X)][\overrightarrow{Y \mapsto \emptyset}] \qquad S'' = \emptyset$ if $S = \emptyset$, S' otherwise

Figure 3.12: Reduction on function returns.

To illustrate how rule *[AppOpq]* uncovers errors, consider the example in Figure 3.13 where execution discovers a potential division-by-0 error in function `f` when passed to an unknown context in an execution branch where `f` is applied thrice. In another example, given in Figure 3.14, the first unknown context cannot discover any error in the function `app` which flows to it, because there is no possible error. However, its result, stored at variable `h`, can potentially reference any value that has escaped to an unknown context. After the the effectful function `inc!` flows to the unknown context, the variable `n` has potentially been modified to `0`. Application of `h` then soundly discovers the potential error in `app` by invoking `app` again.

```
(let ([f (let ([x −3])
           (λ (_)
             (set! x (+ 1 x))
             (/ 1 x)))])
  (• f))
```

Figure 3.13: Havoc stateful callback.

```
(let* ([n −3]
       [inc! (λ (_) (set! n (add1 n)))]
       [app (λ (_) (/ 1 n))]
       [h (• app)])
  (• inc!)
  (h 0))
```

Figure 3.14: Escaped stateful callback.

In Section 3.3.4, we provide a precise definition of behavioral over-approximation and show that these rules are sufficient to soundly approximate the application of an unknown function with arbitrary code. Although a naïve implementation of rule *[AppOpq]* is impractical, we employ several optimizations to enable verification of realistic programs presented in Section 3.4.

Lastly, in *[AppErr]*, applying a potentially non-functional value blames the party performing the application.

Restoring context Figure 3.12 shows rule *[Ret]* for returning a value to the caller. The store-cache entry for the distinct bound-variable X is restored, while entries for free variables are simply invalidated. In addition, the value receives the symbolic name from the caller's scope.

3.3.2 Primitive operations

Figure 3.15 shows a definition of select primitive operations extended to symbolic values.

Although many primitives such as `add1` are partial, we define O in λ_S to be the *unsafe* versions of the primitives, which are total functions that always successfully return a value. We therefore assume that references to primitives are appropriately guarded with contracts (e.g. `add1` would be guarded with `(int? → (λ (_) int?))`) and that programmers have no direct access to unsafe primitives.

The definition preserves precision for concrete arguments and returns an opaque value otherwise.

3.3.3 Path-condition satisfiability

Effective verification relies on precise proving of infeasible path-conditions to eliminate implausible blames and avoid exploration of spurious paths. While simple properties such as implication and exclusion between type-like predicates are easy to

check, more sophisticated properties such as arithmetic take more work to implement efficiently. Making good use of an existing SMT solver can reduce implementation effort without giving up the ability to prove rich invariants. Unfortunately, most SMT solvers only support first-order formulae, which are a significant gap from higher-order effectful expressions.

We overcome this issue by making the following observation: run-time monitoring, even of higher-order values, only requires checking a first-order property at any given point in the program execution. Therefore, contract verification ultimately reduces to proving implications between first-order properties. We rely on the operational semantics to account for the execution of a program, while accumulating first-order invariants in the path-condition to be able to prove necessary properties. Call outs to the solver can be seen a precision optimization that prunes infeasible paths of execution. We present a method to translate the path-condition into first-order formulae such that unsatisfiable formulae implies an infeasible execution path. The formulae's satisfiability can be solved by an existing SMT solver such as Z3 [40] or CVC4 [64].

$$
\begin{array}{lll}
\delta(\Sigma, \texttt{int?}, V) & \ni \texttt{0} & \text{if } V \neq N \\
\delta(\Sigma, \texttt{int?}, V) & \ni \texttt{1} & \text{if, } V = N \text{ or } V = \bullet \\
\delta(\Sigma, \texttt{add1}, N) & \ni N + 1 & \\
\delta(\Sigma, \texttt{add1}, V) & \ni \bullet & \text{where, } V \neq N
\end{array}
$$

Figure 3.15: Primitive operations.

The target formulae uses a unitype embedding `V` of the source language's dynamic type system as shown in Figure 3.16. Source language values are encoded in the solver by pairing together a run-time type tag and an integer denoting the

identity of the value: for source integers, it is just the integer itself; for all other kinds of values such operators, functions, etc., it just distinguishes values.

Figure 3.17 shows the translation.

```
type V = Int (unbox_int: ℤ)
       | Op  (op_id: ℤ)
       | Lam (lam_id: ℤ)
       | ...
function istrue v = (v ≠ Int 0)
```

Figure 3.16: Datatype encoding for SMT solver.

The main translation takes a path-condition Φ and produces a formula stating properties about run-time values. It straightforwardly asserts that the translation of each term in the path-condition is not `Int 0`. Unsatisfiability of this formula would imply an infeasible execution path.

The translation of each expression E produces a term of sort V in the logic. Base values are straightforwardly mapped to those in the logic, while the translation of functions such as primitives and lambdas merely retain the type tag. The translation uses a fresh id for each lambda literal, essentially existentializing the translated value. Primitive operations that have a correspondence in the logic are translated as is. For operations and expressions that do not have obvious translations, we simply existentialize the result, as seen in the default cases of $\{\!\{\cdot\}\!\}$ and $\{\!\{\cdot,\cdot\}\!\}$.

3.3.4 Soundness

This section proves the symbolic execution semantics is sound—that it discovers any possible blame. Specifically, given a program with holes, if *any* instantiation of the holes causes a blame on a label from the incomplete program, then running

$$\{\!|\cdot|\!\} : \Phi \to \mathit{Formula} \qquad \{\!|\cdot|\!\} : E \to \mathit{Term}$$

$$\{\!|E \ldots|\!\} = \wedge\,(\texttt{istrue}\ \{\!|E|\!\})\ldots \qquad \{\!|N|\!\} = \texttt{Int}\ N$$

$$\{\!|O|\!\} = \texttt{Op}\ \mathsf{unique}(O)$$

$$\{\!|\cdot,\cdot|\!\} : O \times E \to \mathit{Term} \qquad \{\!|(\lambda\ (X)\ E)|\!\} = \texttt{Lam}\ X, \text{ where } X \text{ fresh}$$

$$\{\!|\texttt{add1}, E|\!\} = \texttt{Int}\ (1 + \texttt{unbox\_int}\ \{\!|E|\!\}) \qquad \{\!|(O\ E)|\!\} = \{\!|O, E|\!\}$$

$$\ldots \qquad \{\!|E|\!\} = X, \text{ where } X \text{ fresh}$$

$$\{\!|O, E|\!\} = X, \text{ where } X \text{ fresh}$$

Figure 3.17: Translation of path-conditions and expressions into first-order formulae.

the program (with holes) under the symbolic semantics discovers the same blame (Theorem 3).

To prove our soundness theorem, we first define what it means for an incomplete program to *approximate* a complete one, then through a preservation lemma, we show reduction preserves this approximation.

Figure 3.18 shows important rules for the approximation relation between expressions and state components. We describe only the important rules and defer to the appendix for full definitions. The base case of the derivation involves the instantiation of holes ($\bullet$), where each hole either stands for a literal base value, or a syntactically closed λ-abstraction whose body ($E^\bullet$) does not contain further holes and only contains a label ($\mathsf{L}_\bullet$) distinct from any label from the transparent part of the code. Approximation rules for expressions arise from the straightforward structural induction.

The approximation between state components is indexed by an *abstraction*

[Instantiated Exprs] $E^\bullet \in \mathsf{OExp} ::=$

$$(\lambda\ (X)\ E^\bullet) \mid N \mid O \mid X \mid (E^\bullet\ E^\bullet)^{\mathsf{L}_\bullet} \mid (\mathsf{if}\ E^\bullet\ E^\bullet\ E^\bullet) \mid (\mathsf{set!}\ X\ E^\bullet)$$

$$\frac{\mathit{free}(E^\bullet) \subseteq \{X\}}{(\lambda\ (X)\ E^\bullet) \sqsubseteq \bullet} \qquad \frac{E \sqsubseteq E'}{(\lambda\ (X)\ E) \sqsubseteq (\lambda\ (X)\ E')} \qquad \frac{E_1 \sqsubseteq E_1' \quad E_2 \sqsubseteq E_2'}{(E_1\ E_2)^L \sqsubseteq (E_1'\ E_2')^L}$$

$$\frac{E \sqsubseteq E' \quad \mathrm{P} \sqsubseteq_F \mathrm{P}' \quad \Phi \sqsubseteq_F \Phi'}{\mathsf{Clo}(X, E, \mathrm{P}, \Phi) \sqsubseteq_F \mathsf{Clo}(X, E', \mathrm{P}', \Phi')} \qquad \frac{\mathit{free}(E^\bullet) \subseteq \{X\} \quad \mathsf{restricted_F}(\mathrm{P})}{\mathsf{Clo}(X, E^\bullet, \mathrm{P}, \Phi) \sqsubseteq_F \bullet}$$

$$\frac{C \sqsubseteq_F C' \quad C_1 \sqsubseteq_F C_1' \quad C_2 \sqsubseteq_F C_2'}{(\mathsf{if}\ C\ C_1\ C_2) \sqsubseteq_F (\mathsf{if}\ C'\ C_1'\ C_2')} \qquad \frac{C_1 \sqsubseteq_F C_1' \quad C_2 \sqsubseteq_F C_2' \quad L \neq \mathsf{L}_\bullet}{(C_1\ C_2)^L \sqsubseteq_F (C_1'\ C_2')^L}$$

$$\frac{C_1 \sqsubseteq_F C_1' \quad C_2 \sqsubseteq_F C_2' \quad \mathsf{L}_\bullet \notin \{L, L'\}}{(\mathtt{mon}^L_{L''}\ C_1\ C_2) \sqsubseteq_F (\mathtt{mon}^L_{L''}\ C_1'\ C_2')} \qquad \frac{C \sqsubseteq_F C' \quad \mathcal{E} \sqsubseteq_F \mathcal{E}' \quad L \neq \mathsf{L}_\bullet}{\mathcal{E}[(C\ [\,])^L] \sqsubseteq_F \mathcal{E}'[(C'\ [\,])^L]}$$

$$\frac{\mathcal{E} \sqsubseteq_F \mathcal{E}'}{\mathcal{E} \sqsubseteq_F \mathcal{E}'[((\bullet, S)\ [\,])]} \qquad \frac{\mathsf{restricted_F}(C) \quad \mathcal{E} \sqsubseteq_F \mathcal{E}'[((\bullet, S)\ [\,])]}{\mathcal{E}[(C\ [\,])^{\mathsf{L}_\bullet}] \sqsubseteq_F \mathcal{E}'[((\bullet, S)\ [\,])]}$$

$$\frac{C \sqsubseteq_F C' \quad M \sqsubseteq_F M' \quad \Phi \sqsubseteq_F \Phi' \quad \Sigma \sqsubseteq_F \Sigma'}{(C, M, \Phi, \Sigma) \sqsubseteq_F (C', M', \Phi', \Sigma')}$$

$$\frac{\mathsf{restricted_F}(C) \quad \mathcal{E} \sqsubseteq_F \mathcal{E}'[((\bullet, S)\ [\,])] \quad M \sqsubseteq_F M' \quad \Phi \sqsubseteq_F \Phi' \quad \Sigma \sqsubseteq_F \Sigma'}{(\mathcal{E}[C], M, \Phi, \Sigma) \sqsubseteq_F (\mathcal{E}'[((\bullet, S)\ W)], M', \Phi', \Sigma')}$$

Figure 3.18: Approximation between program expressions.

map F from each address in the instantiated component to one in the approximating component. The approximation between closures is only established between closures whose corresponding sub-components are approximating, or between an opaque value and a closure whose control component is purely instantiated, and an environment component that only maps to "unknown" addresses simulated by the special address $_\bullet$ (enforced by map F). Predicate $\mathsf{restricted_F}(\,\cdot\,)$ restricts state components to only contain addresses that are simulated by $_\bullet$. Approximation between

evaluation contexts also include structural and non-structural cases: inserting transparent "frames" into the holes of both evaluation contexts preserves approximation, and an inner opaque application context approximates any number of insertions of purely instantiated "frames". Finally, approximation between states (ς) is either established structurally through component-wise approximation, or non-structurally where an opaque application approximates a state whose evaluation context and top frames are purely instantiated.

We also define the multi-step standard reduction ($\longmapsto\!\!\!\!\!\rightarrow$) as the reflexive-transitive closure of the standard reduction ($\longmapsto$).

Lemma 1 (Reduction preserves approximation)**.** *If* $\varsigma_1 \sqsubseteq_F \varsigma_1'$ *and* $\varsigma_1 \longmapsto \varsigma_2$*, then there exists* ς_2' *and* F' *such that* $\varsigma_2 \sqsubseteq_{F'} \varsigma_2'$ *and* $\varsigma_1' \longmapsto\!\!\!\!\!\rightarrow \varsigma_2'$*.*

Proof. *By case analysis on the reduction* $\varsigma_1 \longmapsto \varsigma_2$ *and approximation* $\varsigma_1 \sqsubseteq_{F'} \varsigma_1'$*. We defer to the appendix for the full proof, and link to our mechanization, written in Lean. Most cases are straightforward and* ς_2' *approximates* ς_1' *in lock step. The main complication comes from applying symbolic function, where the instantiated state* ς_1 *transfers control to purely instantiated code (*$E^\bullet$*), and the symbolic state steps with* [*AppOpq*]*. When this occurs, the same state that succeeds* [*AppOpq*] *continues to approximate an arbitrary number of states that* ς_1 *steps to, as long as* ς_1*'s control comes from instantiated code. By approximation, instantiated code can only transfer to transparent code through returning, or applying one of the "leaked" values approximated by those at* $\bullet$*, which* [*AppOpq*] *soundly simulates.*

With the established small-step soundness of λ_S, we prove that running an in-

complete program E' approximates the result of running any of its full instantiation E. We define a helper meta-function $\mathsf{load}(E) = ((E, \{\}), \{\}, \emptyset, \{\})$ that loads the initial state of a program.

Theorem 3 (Blame soundness)**.** *If* $E_1 \sqsubseteq E_1'$ *and* $\mathsf{load}(E_1) \longmapsto\!\!\!\!\!\rightarrow (\mathcal{E}[\mathtt{blame}^L_{L'}], M, \Phi, \Sigma)$*, where* $L \neq \mathsf{L}_\bullet$*, then there exists* $\mathcal{E}'$*,* M'*,* Φ'*, and* Σ'*, such that* $\mathsf{load}(E_1') \longmapsto\!\!\!\!\!\rightarrow (\mathcal{E}'[\mathtt{blame}^L_{L'}], M', \Phi', \Sigma')$*.*

Proof. *The proof proceeds by rule-induction on the derivation of* $(\longmapsto\!\!\!\!\!\rightarrow)$ *in the concrete error trace. The base case (reflexive) vacuously holds. The inductive case (transitive) holds by lemma 13, where for each single reduction step* $(\longmapsto)$ *on the concrete state, the abstract state continues to approximate the concrete state in zero or more steps.*

Corollary 1 follows from theorem 3, stating a practical implication for verification: if an incomplete program is safe under the symbolic execution, no instantiation of the program can causes a blame on any part of it.

Corollary 1 (Verified components cannot be blamed)**.** *If* $\mathsf{load}(E) \not\longmapsto\!\!\!\!\!\rightarrow (\mathcal{E}[\mathtt{blame}^L_{L'}], M, \Phi, \Sigma)$ *for any label* L *appearing in* E*, then there is no instantiation* $E' \sqsubseteq E$ *such that* $\mathsf{load}(E') \longmapsto\!\!\!\!\!\rightarrow (\mathcal{E}'[\mathtt{blame}^L_{L'}], M', \Phi', \Sigma')$*.*

Proof. *The corollary holds as the contrapositive of Theorem 3.*

3.3.5 From symbolic execution to verification

Traditionally, symbolic execution was used for finding bugs in programs, and not for static verification, because, as originally formulated, it does not provide a terminating over-approximation that guarantees the absence of run-time errors. To turn bug-finding symbolic execution into a verification process, we employ an existing method for turning an operational abstract-machine semantics into an over-approximation through a systematic finitizing of machine components [24]. Thus far, we have allowed the closure to grow without bound, and have left value addresses (α) allocation unspecified. A fresh allocation at each transition will yield a concrete execution (assuming no opaque values), but a different allocation strategy that repeatedly reuses addresses, and joins (conflates) values at those addresses within the store, results in an approximation of multiple execution traces. Indeed, any allocation policy is sound [65]. This is because all possible abstract allocators are consistent simulations of the concrete allocator because the latter always allocates a fresh address. This leaves the choice of allocation as a central "tuning knob" for adjusting the analysis's precision using any desired degree of poly-variance or context sensitivity [63]. We also transform the evaluation contexts into explicit *continuations*, and store-allocate continuations at function boundaries and permit two continuations to become conflated at a single *continuation address*; this redefines each continuation to be a sequence of intra-procedural frames paired with a continuation address for the current invocation. Although the continuation allocator may also be adjusted arbitrarily, recent work has shown that in order to achieve precise

call-and-return matching at no asymptotic cost to analysis complexity, the choice of continuation address should be fixed as (E, P) where E and P are the call target's control and environment respectively [66].

The property we desire for path-sensitive contract verification is that the analysis should only approximate values at different iterations of the same loop, and provide exact execution otherwise. We therefore instrument execution with another component recording the set of control transfers from each source's location to a target's function body. Each such set is an abstraction for a family of traces that differ from one another only by the number of iterations through the same loops. By pairing each syntactic component (e.g. variables) with this set in the allocated address, we obtain an abstraction that meaningfully summarizes program components such as values, continuations, and path conditions with probable cycles resulting from loops. Although this allocation strategy does not guarantee a worst-case polynomial time analysis, (*i.e.*, a loop-free exponential time program would result in exponential time analysis), it tends to give good precision and analysis time for real programs. In addition, modularity helps mitigate the potential worst cases as the user can always break a large program into smaller modules to verify separately.

Finally, we perform a standard global-store widening that weakens the correlation between the store for values, continuations, and path conditions, and other machine components by lifting these to the top level collecting semantics and maintaining the store as the least-upper-bound of all stores visited across all paths. If some of the precision lost during this transformation is needed, it may be regained through the use of a more precise allocation strategy. Strategies, such as Shivers'

time-stamp algorithm [67], may also be used to avoid revisiting a machine configuration until the global store is updated.

3.4 Implementation and evaluation

This section discusses practical improvements in implementing contract verification, and examines our verifier's analysis time and precision on a variety of benchmarks.

3.4.1 Practical improvements

Richer abstract values and widening of base values. The formalism in Section 3.3.1 uses only a single abstract value ($\bullet$) that represents "any value". In our implementation, we enrich each such abstract value to carry a *refinement set* containing predicates it is known to have satisfied. For example, $\bullet^{\{\texttt{int?},\texttt{positive?}\}}$ denotes an abstract positive integer. These refinements provide our analysis semantics a way to short-circuit a call to the SMT solver. For our experiments, we restrict the refinement set to predicates on base types, along with those that syntactically appear in the program (e.g., user-defined contracts), as this provides a balance between precision and convergence. When two different values share the same address, they are widened to an abstract value whose refinement set they both satisfy. In addition, primitives such as addition and multiplication are extended to operate precisely on such abstract values.

Avoid re-running escaped values. Rule *[AppOpq]* in Section 3.3.1 over-approximates all behavior triggered by the unknown part of the program, but is expensive when implemented naïvely due to an ever-growing set of leaked values to be applied at each opaque application. To reduce this cost, we memoize the result of applying each leaked value by the portion of the value store that can potentially affect its behavior, and only re-run a leaked value if the memoized portion of the store has widened since that value was last run. This is especially effective at speeding-up mostly-functional programs since pure functions do not depend on or modify mutable state, and are thus only explored once.

Inter-procedural path-sensitivity Path-sensitivity across function boundaries is crucial for verifying programs with predicates that are abstracted arbitrarily. We achieve this improvement by augmenting the semantics with a *memo-table* that explicitly records information about each application's results and the corresponding path-conditions at each result. At any point in the execution, the memo-table maintains an over-approximation of properties that must hold for each application's arguments and results. Each entry in the memo-table is then translated into an uninterpreted first-order function along with formulae about arguments and results for observed cases. These additional formulae yield more constraints that allow eliminating more spurious paths.

Sharing invariants for provably same locations Application rule [App-Clo] conservatively uses the callee's saved path-condition, and returning rule [Ret]

conservatively invalidates store-cache entries for the callee's free-variables. In the restricted case when the target function is known to share the same free-variables as its caller, properties pertaining to variables shared between a call site and the invoked closure can be combined, strengthening instead of invalidating the saved path condition. Our semantics maintains the invariant that a value's symbolic name is a λ-term only when it is instantiated in the same scope as its caller (by inspection of the reduction rules, only rule *[Lit]* produces a λ-term as a symbolic name). Identical variable names in these cases imply identical dynamic locations (assuming that the program has been α-renamed). We therefore achieve additional precision in these particular cases by sharing the path-condition's constraints and the store-cache entries for those locations between callers and callees that are provably the same.

LET-ALIASING Finally, realistic programming languages allow storing intermediate results in variables, and some programs may rely on reasoning through aliases such as those introduced by macro-expansion as shown in Figure 3.1. In a language with `let`-aliasing, we simply allow the store-cache to initialize each `let`-bound variable to the first value that flowed to them, effectively canonicalizing the symbolic names for values at `let`-bound variables. For example, in the

```
(define (f x)
  (let* ([y (car x)]
         [z y])
    (when (integer? y)
      (set! x #f)
      (set! y #f)
      (add1 z))))
```

Figure 3.19: Stateful program with let-aliasing.

following safe function `f` (Figure 3.19), looking up both `y` and `z` within the function body would give a value with symbolic name `(car x)`. Any test on `y` would give information about `z` and vice versa. As previously noted in Section 3.3.1, the name `x` appearing in symbolic names means the value first bound to location `x` and not the location `x` itself. Any subsequent modification to location `x` only modifies the store-cache and does not invalidate the path-condition.

3.4.2 Implementation

We extend the core semantics described in Section 3.3.1 to a practical implementation that verifies contracts in full Racket programs. By handling core forms directly, and invoking the macro expander to desugar all others, the tool is able to work on significant Racket programs. Compared to the formalism, the implementation provides significant extensions.

First, base values are much richer, including the full numeric tower and values such as strings and symbols. Second, we support data-types such as pairs, mutable boxes, mutable vectors, and user-defined structs with mutable fields. Third, we support additional contract combinators including disjunction, conjunction, recursion, etc, and monitor contracts using *indy* instead of *lax* semantics as presented in rule *[AppArr]* in Section 3.3.1 for complete contract monitoring with correct blame parties [44]. Finally, we support multiple return values and arbitrary function arities, resulting in several additional possible errors.

The implementation is available at `https://github.com/philnguyen/soft-contract`.

3.4.3 Evaluation

To evaluate the tool's effectiveness, we collect benchmarks from several lines of previous work including soft typing for Scheme [26], occurrence type-checking [1], higher-order model checking [16], and symbolic execution [9,18]. In addition, we verify other realistic libraries collected from different sources. We include summarized results for small benchmarks from previous work on verification of pure programs to show the lack of regress. Different benchmark suites each emphasize different aspects of verification. The `occurrence-typing` suite includes small programs whose correctness heavily relies on reasoning about path-sensitivity and aliasing, which is common in untyped programs. The `hors` suite includes many higher-order recursive programs, where safety relies on inter-procedural reasoning. The benchmarks `snake`, `tetris`, and `zombie` are moderately-sized programs with expressive contracts that were collected from an introductory programming course. Finally, remaining benchmarks are existing Racket libraries and programs collected from multiple sources, written in the full Racket language with imperative features. In total, our benchmarks are comprised of 86.7% stateful benchmarks (by lines of code) and 13.2% pure functional benchmarks.

Table 3.1 show benchmark results. Line counts do not include empty and commented lines. The number of checks is a static count of the number of safety checks, including those in primitives and user-specified contracts, which could be eliminated if proven correct. Although the number of checks seems unintuitively high, it reflects the reality of safe dynamic languages. For example, each call site

checks if it is applying a function, and each arithmetic operation checks that it is passed numbers. We also include the number of checks resulting from user-written contracts in parentheses on the right of columns Checks, False Pos, and True Pos. True and false positives are determined through manual inspection. Finally, verification time is measured in seconds.

Our results show that our tool not only can verify almost all contracts and works for many interesting programming patterns, with reasonable analysis time even for large programs. For example, `slatex` initializes mutable boxes with sentinel values (e.g. `#f`), then updates them in a type-consistent way afterwards (e.g. proper non-empty list). Our analysis proves all these uses safe. Another program, `nucleic2-modular`, uses vectors to emulate records with fields having different types, and passes data to many higher-order and partially applied functions, and our analysis verifies that all the indices are in-bounds and updates and references are type-consistent. In addition, the analysis's modularity makes it practical, where the programmer can break a large program into multiple modules to verify separately. For example `nucleic2` is originally a closed program taken from a standard Scheme benchmark suite, and has literal vectors contributing to more than half of the code. Although a good stress test, closed programs are not the focus of our modular verification. Therefore, we abstracted out the input data and verified the computation, demonstrating the intended use case. Finally, among the potential errors reported, some are genuine bugs, as in `slatex`, such as applying an operation expecting a pair where an empty list is possible. We also fix these errors and report the result as `slatex*`.

Program	Lines	Checks	Time (s)	False Pos	True Pos
soft-typing	108	656 (37)	2.064	0 (0)	0 (0)
hors	266	2,194 (119)	4.828	2 (2)	0 (0)
occurence-typing	87	647 (51)	2.423	0 (0)	0 (0)
snake	142	1,232 (96)	2.485	0 (0)	0 (0)
tetris	259	2,390 (200)	6.578	0 (0)	0 (0)
zombie	235	1,049 (39)	3.000	0 (0)	0 (0)
fector	110	388 (20)	4.578	4 (0)	0 (0)
hash-srfi-69	290	1,920 (97)	4.125	1 (1)	0 (0)
leftist-tree	102	916 (25)	0.656	8 (0)	0 (0)
leftist-tree*	110	918 (25)	0.562	0 (0)	0 (0)
morsecode	185	1,013 (12)	4.968	0 (0)	0 (0)
nucleic2-modular	884	6,621 (11)	88.453	1 (0)	0 (0)
nucleic2-modular*	889	6,644 (11)	84.062	0 (0)	0 (0)
ring-buffer	51	353 (19)	0.563	8 (0)	0 (0)
ring-buffer*	58	354 (19)	0.438	0 (0)	0 (0)
slatex	2,300	11,633 (2)	1,213.650	2 (0)	6 (0)
slatex*	2,305	11,693 (2)	1,217.850	2 (0)	0 (0)
TOTAL	8.381	49,861 (785)	2,641.283	28 (3)	6 (0)

Table 3.1: Benchmark Results

Imperative benchmarks include some realistic programs we cannot fully verify. Further inspection reveals that false positives come from a few specific programming patterns.

First, the tool cannot yet reason about invariants established by a module that controls the instantiation of certain structures and maintains strong invariants about all instances (for example, that a "node" is always part of a non-empty proper tree). This is seen in the `ring-buffer` and `leftist-tree` programs. Because our semantics is conservative in assuming that opaque values can come from anywhere, we cannot precisely reason about this pattern. A simple and efficient solution permitting reasoning about this idiom is an important goal for future work. Modified versions (`ring-buffer*` and `leftist-tree*`) with deep structural contracts enable

the analysis to succeed in verifying the programs.

Second, our analysis does not yet precisely verify invariants established by effectful functions, as seen in the `fector` benchmark. In this module, several functions rely on an operation `reroot!`, presented in Figure 3.20 to guarantee that the content of the mutable box is a vector.

```
(define (reroot! fv)
  (match (unbox fv)
    [(list i x fv*)
     (reroot! fv*)
     (let ((v (unbox fv*)))
       (let ((x* (vector-ref v i)))
         (vector-set! v i x)
         (set-box! fv v)
         (set-box! fv* (list i x* fv))))]
    [_ (void)]))
(reroot! fv)
; use of `vector-length` not verified
(vector-length (unbox fv))
```

Figure 3.20: `reroot!` example from fector

Because most data-structures in verification arise from unknown sources, and these data structures, after passing through recursive contracts, are cyclic (to approximate all possible values inhabiting the contracts), addresses typically stand for multiple concrete addresses, preventing symbolic execution from performing strong updates to such addresses. More precise abstraction and reasoning for mutable recursive data-structures is a second goal for future work. Techniques such as abstract garbage collection and counting [68] can help tracking precisely the cardinality of abstract addresses, thus justifying strong updates.

3.5 Related work

Our work builds on existing approaches to static contract verification via symbolic execution. We relate our current contributions to these efforts and then more broadly to work on verification in higher-order settings.

3.5.1 Symbolic execution

Symbolic execution simulates a program's evaluation on symbolic values which are unknown and may stand in for a number of possible concrete values. Path conditions—formulas unique to a particular sequence of branches—constrain these symbolic variables and denote infeasible runs where contradictory. In first-order settings, symbolic execution has a mature and well-investigated methodology [10,47]; in higher-order settings however, it remains an active area of ongoing research. In a higher-order setting, where a concrete value may be a first-class function, a variety of sound choices exist for modeling the application of an opaque (symbolic) function which do not exist in first-order languages [9].

There is also a general difference in motivation; while most applications of symbolic execution involve bug finding and code auditing, our focus is on its use for modular program verification and static contract checking.

3.5.2 Static contract verification

We have built on prior work [9,18] that develops static contract verification as a (higher-order) symbolic execution of untyped functional programs (in this case,

Racket). Previous work following this approach only handles pure functions, and while robust for untyped functional programs, it falls down in the presence of even well-encapsulated mutable state and other non-functional idioms. Further, the implementation presented in that work handled only a small subset of Racket.

Another approach [20, 21] embeds dynamic monitors into the target program and simplifies them away using compiler techniques and a specialized symbolic engine. This approach of symbolic simplification may be applicable to untyped programs; however, a crucial pass used in this approach, dubbed logicization, requires type annotations in order to translate program expressions into a first-order logic (FOL). A similar method for Haskell [22] leverages a denotational semantics that can be mapped onto first-order logic; this is both dependent on type information and on the pure call-by-name semantics of Haskell.

Contract verification in the setting of first-order contracts is also more restricted, and its investigation more mature. A prominent example is the work on verifying C# contracts done in the Code Contracts project [69] and the Spec# system [70, 71], with which, contract counter examples can be generated and explored using a debugger.

Our approach allows higher-order dependent contracts and mutable state, does not assume types to guide the verification process, supports blame, and verifies run-time type safety in addition to richer contracts as part of the same process. In addition, the aforementioned type-based approaches assume explicit monitoring of recursive calls which allow the use of contracts as inductive hypotheses in such calls. Our approach permits this as well, but remains flexible enough to accommodate

Racket's semantics which does not monitor recursive call sites.

3.5.3 Refinement type checking

Refinement type systems permit the inclusion of logical propositions within type annotations and represent another approach to statically stating and proving richer properties of programs—as such, there is meaningful overlap with contract verification. Refinement type systems either restrict the expressivity of type refinements so that checking is decidable [4], or they permit arbitrary refinements, as do contracts in Racket, and use a general-purpose solver in the attempt to discharge refinements [5, 23, 72]. When a refinement cannot be discharged, a system may reject the program as a whole [5, 72], or, as in the case of hybrid type checking [23], it may residualize a run-time check to dynamically enforce each unverified refinement. Manifest contracts [43, 73] equip static types with contracts as refinements, verifying contracts either statically via sub-typing, or using a dynamic cast. Manifest contracts have also been extended to algebraic data and mutable state [74, 75], including stateful contracts. Residualized run-time checks correspond to our approach of *soft* contract verification which degrades gracefully, removing only those contracts which are verified. Unlike our approach, manifest contracts and hybrid type checking require type annotations and only permit predicates on base types; while our approach extends to dependent contracts, no mechanism currently exists for mixing flat and higher-order specifications in refinement types. Furthermore, contract evaluation may become stuck, diverge, or have side effects, while refinements are

more restricted. Special care must be taken where refinements themselves contain a potentially failing cast [23, 43]. Dependent JavaScript [76, 77] supports expressive refinements for stateful JavaScript programs, including sophisticated dependent specifications. Unfortunately, this approach relies on extensive type annotations and whole-program analysis.

3.5.4 Higher-order model checking

Higher-order model checking is also applicable to verification problems in this setting. This approach proceeds by compiling a target program into a higher-order recursion scheme (HORS)—these are essentially programs in the simply-typed λ-calculus, with finitely inhabited types, that generate unbounded trees representing all possible program evaluation paths. While HORS generalizes finite-state and pushdown systems, its model checking problem remains decidable while in this ideal setting of simply-type λ-calculus and finite base types [78–80]; however, there remains a significant gulf between this and real-world language features. Other work has broadened the applicability of this approach to cases [81], to untyped languages [82, 83], and to infinite data domains such as integers and algebraic data-types [16, 84]. The complexity of higher-order model checking is n-EXPTIME-hard [85] but practical progress has lead to engines which can handle checking some "small but tricky ... functional programs in under a second" [86].

While our approach tackles untyped, higher-order, stateful programs with sophisticated real-world language features, higher-order model checking is restricted

to small, pure code snippets using a more restricted set of features. In addition, our approach allows programmers to add dynamically enforced program invariants via contracts and dispatch them gradually while the HORS approach only supports assertions on first-order data which must all be verified. Our approach also permits verification in the presence of unknown library functions (not only base values), a crucial allowance for modular program verification. Our evaluation demonstrates that our tool can verify many of the "small but tricky" examples checked in the HORS literature.

3.5.5 Broadly related static analysis

Separation logic [87–89] provides a framework for reasoning compositionally about the abstract effect of computations—we do not. More broadly, summarization-based and bottom-up approaches aim to produce modular and compositional analyses of program components. Our work takes a more operational view and does not summarize the effects of known functions (instead it simulates them under an approximation). It does aim, however, to summarize the effects of arbitrary unknown functions. This is done by operationally tracking which heap locations an unknown function has access to. This process does bear some resemblance to a frame rule or localization technique: if an unknown function does not have access to a location, it cannot have an effect on it. As our evaluation shows, this is sufficient to verify the preponderance of contracts in our benchmark suite.

As the target language is functional, benchmarks use mutable data sparingly

and usually in a well-encapsulated manner. In addition, contracts in Racket usually enforce properties of a component's input and output data, not about its side effects. For these reasons as well, we were able to obtain good results without separation logic. We are not automatically verifying full functional correctness properties of programs that make heavy use of pointer manipulation. While it is possible to express, for example, a linked-list reversal algorithm that does pointer swapping, our approach is unlikely to do a good job proving it correct. Rather, we target type- and dependent-type-like properties expressed as contracts on higher-order, imperative programs.

The trace semantics described in [90] is also related to our system. Both use non-determinism and opaque values to model the behavior of components in a context that includes mutation. This work uses denotational semantics and addresses fundamentally distinct concerns. This system yields a sound and complete representation of a program's operational behavior, whereas our system yields a computable approximation aimed at static analysis applications.

Other notions of abstraction for addresses can also be used with our approach; for example, a recency abstraction [91] tracks abstraction cardinality (singleness) [92] in addition to a distinguished non-approximate heap location. If a heap location is known to be single, our approach may be able to yield improved results where a strong-update is enabled ahead of a contract.

3.6 Conclusion

Contracts allow programmers to enforce sophisticated invariants within their code using the power and expressiveness of the host language. However, this flexibility comes at a cost to run-time efficiency and without any compile-time assurance of correctness. Soft contract verification offers a remedy—by attempting to verify contracts statically; where a contract can be verified, its code may be removed, permitting optimization of the underlying program, and the program property it had enforced at run-time will have been proven for all possible executions. We demonstrate that symbolic execution may be extended to support higher-order languages with mutable state in modular fashion, permitting arbitrary interaction with unknown external components. This extension to opaque (unknown and potentially stateful) functions requires us to make subtle but crucial choices; for example, accounting for the possibility of a known function escaping permanently to an opaque context. Our approach scales to the full Racket programming language and our evaluation shows that our tool can verify more than 99.9% of dynamic checks across a suite of realistic (14% pure and 86% stateful) benchmarks.

Chapter 4: Termination as a Run-time Contract

Termination is an important but undecidable program property, which has led to a large body of work on static methods for conservatively predicting or enforcing termination. One such method is the *size-change termination* approach of Lee, Jones, and Ben-Amram, which operates in two phases: (1) abstract programs into "size-change graphs," and (2) check these graphs for the *size-change property*: the existence of paths that lead to infinite decreasing sequences.

We transpose these two phases with an operational semantics that accounts for the *run-time enforcement of the size-change property*, postponing (or entirely avoiding) program abstraction. This choice has two key consequences: (1) size-change termination can be checked at run-time and (2) termination can be rephrased as a safety property analyzed using existing methods for systematic abstraction.

We formulate run-time size-change checks as *contracts* in the style of Findler and Felleisen [38]. The result compliments existing contracts that enforce partial correctness specifications to obtain *contracts for total correctness*. Our approach combines the robustness of the size-change principle for termination with the precise information available at run-time. It has tunable overhead and can check for non-termination without the conservativeness necessary in static checking. To ob-

tain a sound and computable termination analysis, we apply existing abstract interpretation techniques directly to the operational semantics, avoiding the need for custom abstractions for termination. The resulting analyzer is competitive with with existing, purpose-built analyzers.

4.1 Size-change Contracts

A Fool's Errand Imagine for a moment there existed a run-time mechanism for checking whether a program, in its current state, will run forever or eventually terminate. Such a check would be eminently useful. Any run-time mechanism for enforcing partial correctness could easily be made to enforce total correctness by use of this check. Moreover, static verification of termination would boil down to proving these run-time checks always succeed, much like how type systems prove run-time tag checks always succeed.

Of course, whether a program eventually terminates is one of the most useful, yet fundamentally and famously unknowable, properties of programs [93, 94]. Moreover, due to its nature as a liveness property—it cannot be violated in a finite execution—it cannot be directly checked at run-time.

An Indirect Tack Despite this situation, an indirect partial solution is possible by instead considering a safety property that implies the liveness property. This indirect approach underlies successful static termination analysis tools such as Terminator [95]. Given such a safety property, enforcing it at run-time would ensure a non-terminating program would eventually "go wrong" by violating the

safety property, at which point it could be stopped. The one, unavoidable, wrinkle is that there will be some programs that run astray of the safety property, despite eventually terminating. In this approach, static verification of termination could, as suggested before, be phrased and designed just as any other safety verification problem by proving the impossibility of a run-time check failure. This approach has the added advantage that any program can be dynamically monitored regardless of whether it can be statically verified.

A Universal Safety Property for Termination To design a run-time termination checker, the critical question is: what is a good safety property to enforce that implies termination? Tools such as Terminator, AProVE, and many others discover a program-specific termination argument, either through static analysis or CEGAR-style refinement. While such approaches have proved quite successful in learning complex termination arguments, these approaches undermine the ability to dynamically monitor termination.

To remedy the situation, we propose using a universal property. A promising candidate is the so-called *size-change principle* of [27]. The principle has proved useful in static termination checking and has a well understood theory. Unfortunately, the original work on size-change termination, which was developed for static verification, defines the size-change principle as a property of a program *abstraction*: a set of so-called size-change graphs (roughly a program call graph annotated with information about non-ascending data flows between function parameters).

This Paper We propose a run-time check inspired by the size-change principle for program termination that *dynamically* builds and checks precise size-change graphs. This dynamic mechanism is useful in its own right, but also can be used as a basis for designing static termination checkers. Such static checkers can benefit from advances in static analysis, particularly in abstract interpretation, since termination checks are integrated into the language specification and do not require custom abstractions or algorithms.

We formalize a semantics for a higher-order functional language that enforces the size-change principle, thereby ensuring all programs terminate (§4.2.5). Moreover, we introduce a behavioral software contract, in the style of Findler and Felleisen [38], that enables the selective enforcement of size-change termination. Such contracts, when combined with traditional pre- and post-condition contracts, form a notion of contracts for total-correctness.

We also develop a static termination checker (§4.3) by applying the static contract verification technique of chapter 3 to the size-change semantics. The resulting tool has no termination analysis specific abstractions, it simply treats the size-change principle check as it would any run-time check, and yet an empirical evaluation (§4.4) shows that it is competitive with several state-of-the-art purpose-built termination analyzers: Liquid Haskell, Isabelle, and ACL2.

Contributions This section contributes:

1. a semantic account of the size-change principle,

2. a proved-correct contract for size-change-based termination of functions,

3. an implementation technique that preserves proper tail-calls and enables tunable run-time overhead, and

4. a static termination checker obtained by generic abstract interpretation techniques.

4.2 Examples and Intuitions

This section develops intuitions for how dynamic checking of *size-change termination* (SCT) works via worked examples. We begin by sketching how SCT works in the original static setting of [27].

4.2.1 The Factorial of Termination Papers

Consider the Ackermann function, the standard-bearer of examples for papers on termination due its simplicity as a total—but not primitive recursive—function, presented in figure 4.1 in Scheme notation:

```
1 (define (ack m n)
2   (cond [(= 0 m) (+ 1 n)]
3         [(= 0 n) (ack (- m 1) 1)]
4         [else    (ack (- m 1)
5                       (ack m (- n 1)))]))
```

Figure 4.1: Ackermann implementation

For the moment, assume the function is only applied to natural numbers. Under that assumption, `ack` always terminates and the SCT method suffices to prove it.

SAFE SIZE-CHANGE GRAPHS: The approach starts by using program analysis or abstract interpretation to enumerate the ways in which a call to `ack` could result in a subsequent call to `ack` before returning. We can see there are three potential recursive calls within the function definition on lines 3, 4, and 5. For each of these calls, describe the pairwise relations between the arguments of the call and recursive call in terms of their size. (The original SCT approach assumes the language has only well-founded data types with a known partial order.)

So for example, consider the possible call:

$$\texttt{(ack m n)} \rightsquigarrow \texttt{(ack (- m 1) 1)}.$$

There are two parameters, so we consider four possible size-change relations between the inputs and recursive call. It is clear that the `m` parameter is strictly smaller in the recursive call compared to the input of the original call. This change is described with a "size-change graph," $\{(\texttt{m} \downarrow \texttt{m})\}$, which is a binary relation saying that whatever value is given for `m` in the original call will become a strictly smaller argument `m` in the recursive call. But there is no size-change relation between the original input `n` and recursive parameter `m` or `n`, nor between the original `m` and recursive `n`, which we know is `1`: each could become larger, smaller, or stay the same.

Moving on to the call in line 5:

$$\texttt{(ack m n)} \rightsquigarrow \texttt{(ack m (- n 1))},$$

we can see that `m` is unchanged and `n` is strictly smaller between calls (but there's no relation between `m` and `n`), so we describe this call with the graph: $\{(\texttt{m} \mathrel{\overline{\downarrow}} \texttt{m}), (\texttt{n} \downarrow \texttt{n})\}$, which says `m` is non-ascending and `n` is descending.

Finally, consider the call in line 4:

$$\texttt{(ack m n)} \rightsquigarrow \texttt{(ack (- m 1) ...)},$$

where the elided code is the nested call to `ack` of line 5. Here it is clear that `m` strictly descends, but unclear what happens with `n`. So we can describe this call with the size-change graph as used for the call in line 3.

At this point, we now have a sound collection of size-change graphs for all possible successive calls to `ack`. They are sound since they properly account for all possible strict descent or non-ascending transitions that occur in recursive calls at run-time. As a side note: it is always safe to omit graph arcs (potentially losing sufficient evidence to prove termination), but all arcs included in a graph must soundly over-approximate all possible run-time behaviors.

SIZE-CHANGE PRINCIPLE: The next task is to check this set of graphs for the *size-change termination principle* (SCP) to see if every infinite computation would give rise to an infinitely decreasing value sequence, according to the size-change graphs. To do this, we consider closing the set of graphs under sequential composition of size-change graphs. The sequential composition of two graphs models two successive calls to construct the size-change from the first to last call, and is defined, informally,

as follows: there is a strict descending arc between two parameters, if there exists a path between the parameters containing a strict descent; there is a is a non-ascending arc if there exists a path containing only non-ascent arcs. Otherwise, there is no path.

Coincidentally, the set of graphs for `ack` is already closed under sequential composition, but to see an example, here's the sequential composition of calling `ack` on line 3 (or 4) followed by `ack` on line 5:

$$\{(\mathtt{m} \downarrow \mathtt{m})\}; \{(\mathtt{m} \overline{\downarrow} \mathtt{m}), (\mathtt{n} \downarrow \mathtt{n})\} = \{(\mathtt{m} \downarrow \mathtt{m})\},$$

which is equivalent, in terms of size-change, as calling `ack` on line 3 (or 4).

Once closed, we check each size-change graph to see if it

1. is idempotent, i.e. $g; g = g$, and

2. lacks a self descending arc, i.e. $(x \downarrow x)$ for some parameter x.

If such a graph exists, it represents a potential sequence of calls that can be iterated infinitely often with no descent and thus it violates the size-change principle. If it lacks such a graph, the program terminates. In the case of `ack` both graphs have self-descending arcs and therefore terminates.

Dynamic size-change graphs: Having established the basic notions of the static SCT approach, we now turn to a dynamic approach to monitoring size-change termination.

The main idea is that rather than rely upon a program analysis to enumerate

the various ways a function may call itself, we simply run the program and observe such calls. Each time a function invokes itself, a size-change graph is dynamically generated. Throughout a computation, the call sequence of size-change graphs is accumulated. Before entering a function call, the current call sequence is checked for the size-change principle. If it is violated, the program is stopped and an error signaled; otherwise the call proceeds.

A program violating the size-change principle eventually accumulates a call sequence witnessing the violation; a program maintaining the principle eventually terminates.

In a similar vein, we need not rely on static analysis to infer the size-change relation between arguments. At run-time, there are concrete values available at both the call and recursive call site. Inferring the size-change graph boils down to checking a partial order pairwise on the arguments. This is both easy to do and potentially much more precise than the static approach. For example, there may be size-change relations that hold on the particular path of execution under scrutiny, which do not hold in general.

To make things concrete, reconsider `ack`. When switching perspectives to the dynamic setting, we are no longer concerned with proving termination for all possible executions of the function, but rather with a particular application. Consider `(ack 2 0)`. The complete tree of call sequences and generated size-change graphs is shown in Figure 4.2, but let us step through its construction. In calling `(ack 2 0)`, control

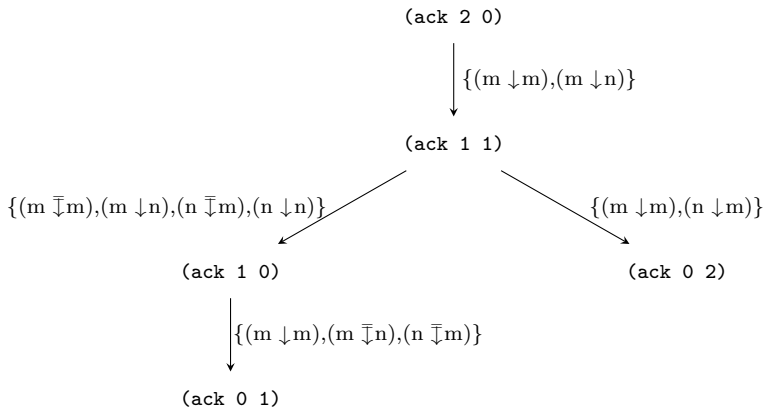

Figure 4.2: Calls and size changes for `(ack 2 0)`

reaches the recursive call on line 3, so we have the call sequence:

$$\texttt{(ack 2 0)} \rightsquigarrow \texttt{(ack 1 1)},$$

from which we can read off the size-change graph. Just as in the static case, we have ($\texttt{m} \downarrow \texttt{m}$), but additionally, we know that ($\texttt{m} \downarrow \texttt{n}$). This fact does not hold in all runs of `ack`, but it holds in this one.

Aside: it is worth noting that this additional program fact is not necessary in this particular example. After all, we have statically proven `ack` terminates in all cases using less information. But for the purposes of illustration, we can see that more information is available at run-time; and in principle, it is possible to safely execute size-change terminating programs that are not statically verifiable, just as by analogy it is possible to dynamically monitor type safety of programs that do not trigger run-time type errors, yet are statically ill-typed.

Returning to the example: having generated the graph for this call, we then

check the SCT principle for the active sequence of calls; in this case there is just the one graph: $\{(\texttt{m} \downarrow \texttt{m}), (\texttt{m} \downarrow \texttt{n})\}$, which satisfies the size-change property, so execution proceeds.

Now `(ack 1 1)` reaches the `else` branch and invokes a recursive call to `(ack 1 0)` on line 5. This call generates the graph $\{(\texttt{m} \overline{\downarrow} \texttt{m}), (\texttt{m} \downarrow \texttt{n}), (\texttt{n} \overline{\downarrow} \texttt{m}), (\texttt{n} \overline{\downarrow} \texttt{n})\}$. We now check the size-change graphs of the sequence leading to this point, i.e., the size-change graphs of:

$$\texttt{(ack 2 0)} \rightsquigarrow \texttt{(ack 1 1)} \rightsquigarrow \texttt{(ack 1 0)},$$

and determine if the size-change property holds, which it does. Now `(ack 1 0)` reaches `(ack 0 1)` with graph $\{(\texttt{m} \downarrow \texttt{n}), (\texttt{n} \downarrow \texttt{m}), (\texttt{m} \overline{\downarrow} \texttt{m}), (\texttt{n} \overline{\downarrow} \texttt{n})\}$, and the call sequence still satisfies SCP. At this point `(ack 0 1)` terminates with `2`. This brings control back to the evaluation of `(ack 1 1)`, which is now ready to proceed to second call to `ack` on line 4 with the arguments `(ack 0 1)`. At this point, we have the call sequence:

$$\texttt{(ack 2 0)} \rightsquigarrow \texttt{(ack 1 1)} \rightsquigarrow \texttt{(ack 0 2)}.$$

Note the calls to `(ack 1 0)` and `(ack 0 1)` are no longer active since they have returned. Again we check the SCP of the size-change graph sequence for active calls, which holds and the program terminates.

A SOMETIMES-BUGGY ACKERMANN: We have seen how run-time SCT monitoring works for programs that maintain the size-change principle, but what about buggy programs that do not? Consider the `ack` example, but change the call on line 4 from `(ack (- m 1) ...)` to `(ack m ...)`. Computing `(ack 2 0)` would proceed as before until reaching the call on line 5, corresponding to the right branch of the tree in Figure 4.2, i.e. representing the call sequence:

$$\texttt{(ack 2 0)} \rightsquigarrow \texttt{(ack 1 1)} \rightsquigarrow \texttt{(ack 1 2)},$$

whose last size-change graph is now $\{(\texttt{m} \mathrel{\overline{\downarrow}} \texttt{m}), (\texttt{n} \mathrel{\overline{\downarrow}} \texttt{m})\}$. But this graph is idempotent and contains no self-descents, so at the point of this call a size-change violation is signaled.

4.2.2 Keeping Closures in Order

The original formulation of SCT was for a first-order functional language with a well-founded partial order on values. This was done largely to simplify the first phase of static SCT verification where call-graphs and size-change relations are generated. In higher-order languages, however, computing call-graphs is itself a significant, extensively studied problem [96]. In the dynamic formulation, higher-order functions do not pose a serious challenge since calls are observed as they occur.

The one remaining issue concerns the choice of partial order for functions. We make a simple choice and consider all closures to be incomparable. Consequently, no

termination proof goes through by an argument about closure size. This is not to say that all programs that use higher-order functions will be rejected by the size-change monitor, just that they must have some descent on base values between calls to the same function. Our empirical evaluation (§4.4) confirms this is a reasonable choice. To illustrate, let us consider a program that recursively accumulates a closure and eventually applies it in the base case of the function.

Consider a `len` function for lists, written in CPS as in figure 4.3. Static analysis of size-change termination relies on an underlying control-flow graph, which must eventually conflate all closures generated on line 5, regardless of call-sensitivity. This results in a spurious loop where each closure bound to `k` may appear to call one with a *larger* argument, failing the size-change principle.

```
1  (define (len l) (loop l (λ (x) x)))
2  (define (loop l k)
3    (cond [(empty? l) (k 0)]
4          [(cons? l)
5           (loop (rest l) (λ (n) (k (+ 1 n))))]))
```

Figure 4.3: List length in CPS

Dynamic checking of size-change termination does not have this problem, because all the closures are exact and distinct. Even though the number of closures is arbitrary, they are finite up to the previous loop descending on `l`, which has been proven to terminate. The call sequence for `(len '(2 1))`, which is a

sequence of tail-calls:

$$\texttt{(len '(2 1))} \rightsquigarrow \texttt{(loop '(2 1) (}\lambda\texttt{ (x) x))} \rightsquigarrow$$

$$\texttt{(loop '(1) } k_1\texttt{)} \rightsquigarrow \texttt{(loop '() } k_2\texttt{)} \rightsquigarrow$$

$$\texttt{(}k_2\texttt{ 0)} \rightsquigarrow \texttt{(}k_1\texttt{ 1)} \rightsquigarrow \texttt{((}\lambda\texttt{ (x) x) 2)} \rightsquigarrow \texttt{2}$$

The recursive calls of `loop` to itself are easily proven safe through descent on the list. The successive calls to continuations are arbitrarily many but finite. Here k_1 and k_2 stand for different closures of the `(`λ `(n) (k (+ 1 n)))` term. The computation proceeds to an answer since SCP is only checked between calls to the *same* closure, directly or indirectly.

It is possible to define a partial order on closures, and this may be a worthwhile addition to our approach. For example, [97] extend SCT to the untyped λ-calculus and use a partial order based on closure depth to order functions. In theory, this could be used to dynamically order closures in our approach, too, however pragmatically, it requires run-time facilities for "opening" closures [98], which are not typically available.

4.2.3 Termination and Blame

It is useful to assert size-change termination of particular functions, without necessarily asserting termination of the whole program. For this reason, we introduce a contract, `terminating/c`, in the style of [38]. One key component of contract semantics is *blame* to explain the party at fault in contract errors. While our formal model does not represent blame, our implementation does. The addition of

blame is at once simple and powerful in the setting of termination contracts. Each `terminating/c` use marks a blame party, and if the function so wrapped fails to terminate on some call, that location in the program is blamed. No sophisticated run-time machinery is required.

The addition of blame enables a virtuous cycle in program development. If a terminating function `f` calls `g`, then any failure to terminate on the part of `g` will be blamed on `f`. To protect themselves from being blamed, the author of `f` can in turn impose the same contract on `g`, leading to richer specifications and precise errors pinpointing the faulty component. Finally, the provision of size-change termination contracts enables a gradual-typing-style integration of total and partial program components.

4.2.4 The Power of Dynamic Enforcement

Checking termination of a interpreter for a language that is Turing-complete is challenging—after all, the interpreter does not terminate on all programs. Nevertheless, dynamic size-change monitoring allows the interpretation of many interesting programs to finish. In Figure 4.4, we present a λ-calculus implementation that first compiles the term to a procedure and then applies this procedure to an environment. The compilation itself terminates by structural recursion, which is simple to check, but the compilation result is a procedure whose termination is not obvious. In fact, in this example, the first test program $\mathtt{c}_1$ terminates when run, but $\mathtt{c}_2$ loops infinitely. Dynamic size-change monitoring flexibly allows the first one to finish, and quickly catches the divergence in the second one. The ability to check for ter-

mination of specialized programs highlights the advantages of dynamic termination checking.

Execution of `(c1 (hash))` terminates because no function ever calls itself with a non-decreasing argument. In contrast, during the execution of `(c2 (hash))`, the compilation result of `(λ (y) (y y))` calls itself (indirectly) with a non-decreasing argument (in this case, identical), hence caught by the monitoring. As shown in the evaluation section (§4.4), our implementation is able to confirm the termination of a Scheme interpreter executing merge-sort.

```
1  (define comp
2    (terminating/c
3      (λ (e)
4       (match e
5         [`(λ (,x) ,e)
6          (let ([c (comp e)])
7            (λ (ρ) (λ (z) (c (hash-set ρ x z)))))]
8         [`(,e1 ,e2)
9          (let ([c1 (comp e1)] [c2 (comp e2)])
10           (λ (ρ) ((c1 ρ) (c2 ρ))))]
11        [(? symbol? x) (λ (ρ) (hash-ref ρ x))]))))
12 (define c1
13   (terminating/c ; Okay
14     (comp '((λ (x) (x x)) (λ (y) y)))))
15 (define c2
16   (terminating/c ; Okay
17     (comp '((λ (x) (x x)) (λ (y) (y y))))))
18 (c1 (hash)) ; Okay
19 (c2 (hash)) ; Error
```

Figure 4.4: A checked λ-calculus implementation

4.2.5 Dynamic SCT Monitoring

This section introduces language λ_{SCT}, which is λ-calculus, extended with base values and primitive operations, and with a modified semantics ensuring that all programs terminate. Figure 4.5 shows λ_{SCT}'s syntax and semantics.

4.2.6 A Terminating Semantics

The domain of values (V) in λ_{SCT} includes primitives (O), integers (N), pairs (V_1, V_2), and closures ($\mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})$). No primitive in λ_{SCT} is allowed to cause divergence.

We present the semantics of λ_{SCT} in Figure 4.5. The semantics is defined by relation $\mathrm{P}, M \cup \{\bot\} \vdash E \Downarrow \mathrm{A}$, which extends the standard semantics by accumulating a size-change table M. The size-change table maps each function (V) to the most recent arguments it was applied to, in the current dynamic extent, as well as a sequence of size-change graphs ($\overrightarrow{G}$) recording ways in which arguments of (V) descend. A size-change graph (G) is a set of arcs of the form ($i \downarrow j$) or ($i \,\overline{\downarrow}\, j$), indicating that the i-th argument always strictly descends ($\downarrow$) or never ascends ($\overline{\downarrow}$) to the j-th argument.

An evaluation answer (A) can be a value, run-time error ($\mathsf{error}^{\mathsf{RT}}$), or size-change error ($\mathsf{error}^{\mathsf{SC}}$). A run-time error is one resulting from misuse of language constructs as standard in a programming language (e.g. applying a primitive to arguments not in its intended domain, applying a non-function, or a function of the wrong arity, etc.). A size-change error is one raised by size-change monitoring upon detecting a size-change violation. We omit rules that introduce run-time errors and error propagation, as they are entirely standard and not the focus of this chapter.

Rule *[SC-App-Clo]* shows application of a closure. In λ_{SCT}, all applications are enforced to have the size-change property. Before executing the function's body as in the standard semantics, we update the size-change table and guard against a

$$
\begin{array}{rl}
\text{[Expressions]} & E ::= O \mid N \mid (\lambda\ (\overrightarrow{X})\ E) \mid X \mid (E\ \overrightarrow{E}) \mid (\mathsf{if0}\ E\ E\ E) \\
\text{[Value Literals]} & N ::= 0 \mid -1 \mid 1 \mid \ldots \\
\text{[Primitives]} & O ::= + \mid \mathsf{cons} \mid \mathsf{car} \mid \mathsf{cdr} \mid \ldots \\
\text{[Values]} & V ::= O \mid N \mid (V, V) \mid \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P}) \\
\text{[Standard Answers]} & A ::= V \mid \mathsf{error}^{\mathsf{RT}} \\
\text{[Answers]} & \mathrm{A} ::= A \mid \mathsf{error}^{\mathsf{SC}} \\
\text{[Environments]} & \mathrm{P} = X \rightarrow V \\
\text{[Size-change Table]} & M \in V \rightharpoonup \overrightarrow{V} \times \overrightarrow{G} \\
\text{[Size-change Graph]} & G \in \mathcal{P}(\mathbb{N} \times R \times \mathbb{N}) \\
\text{[Change]} & R ::= \downarrow \mid \overline{\downarrow}
\end{array}
$$

SC-Err
$$\frac{}{\mathrm{P}, \bot \vdash E \Downarrow \mathsf{error}^{\mathsf{SC}}}$$

SC-Prim
$$\frac{}{\mathrm{P}, M \vdash O \Downarrow O}$$

SC-Base
$$\frac{}{\mathrm{P}, M \vdash N \Downarrow N}$$

SC-Lam
$$\frac{}{\mathrm{P}, M \vdash (\lambda\ (\overrightarrow{X})\ E) \Downarrow \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})}$$

SC-Var
$$\frac{}{\mathrm{P}, M \vdash X \Downarrow \mathrm{P}(X)}$$

SC-If-T
$$\frac{\mathrm{P}, M \vdash E \Downarrow 0 \qquad \mathrm{P}, M \vdash E_1 \Downarrow \mathrm{A}}{\mathrm{P}, M \vdash (\mathsf{if0}\ E\ E_1\ E_2) \Downarrow \mathrm{A}}$$

SC-If-F
$$\frac{\mathrm{P}, M \vdash E \Downarrow V \text{ where } V\ 0 \qquad \mathrm{P}, M \vdash E_2 \Downarrow \mathrm{A}}{\mathrm{P}, M \vdash (\mathsf{if0}\ E\ E_1\ E_2) \Downarrow \mathrm{A}}$$

SC-App-Clo
$$\frac{\begin{array}{c}\mathrm{P}, M \vdash E \Downarrow \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}') \qquad \mathrm{P}, M \vdash \overrightarrow{E_x} \Downarrow \overrightarrow{V_x} \\ \mathrm{P}'[\overrightarrow{X \mapsto V_x}], \mathit{upd}(M, \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \overrightarrow{V_x}) \vdash E' \Downarrow \mathrm{A}\end{array}}{\mathrm{P}, M \vdash (E\ \overrightarrow{E_x}) \Downarrow \mathrm{A}}$$

Figure 4.5: Syntax and semantics of λ_{SCT}.

violation to the size-change property. Helper function *upd* updates the size-change table with the function's latest arguments and size-change graph, potentially returning $\bot$ if there is a size-change violation. If *upd* does not return a table, the evaluation aborts with an error as in rule *[SC-Err]*.

4.2.7 Updating and Monitoring Size-change Graphs

Figure 4.6 lists helper functions that update and monitor SCT.

Function *upd* takes the size-change table (M), function (V), and its latest arguments ($\overrightarrow{V_n}$). It computes a new size-change graph (G_n) for the transitions from the previous arguments ($\overrightarrow{V_{n-1}}$) to these new arguments, ensures that the new graph sequence ($G_n :: \overrightarrow{G_{n-1}}$) does not violate the size-change property, and then updates the graph in m. If function V has not been applied before and there is no entry in m, a trivial entry with the current argument list as well as the empty graph sequence is stored.

Function *graph* computes a size-change graph from two value lists. For each value V_j at index j in the latter list that is observed to be strictly smaller than some value V_i at index i in the former list, an arc $(i \downarrow j)$ is included in the graph. When the values are equal, we include $(i \overline{\downarrow} j)$ instead.

The composition (;) of two size-change graphs (G_0 and G_1) includes an arc $(i \downarrow k)$ if there is an arc $(i\ R\ j)$ in G_0 and $(j\ R\ k)$ in G_1, with at least one arc being a strict descent. If i propagates to k only through non-ascendancy, the weaker arc $(i \overline{\downarrow} k)$ is included.

Finally, predicate *prog?* checks for the lack of violation to the size-change

$$
\begin{aligned}
\mathit{upd} &: M \times V \times \overrightarrow{V} \to M \cup \{\bot\} \\
\mathit{upd}(M,V,\overrightarrow{V_n}) &= M[V \mapsto (\overrightarrow{V_n}, [])], \text{ if } V \notin M \\
\mathit{upd}(M,V,\overrightarrow{V_n}) &= \begin{cases} M[V \mapsto (\overrightarrow{V_n}, G_n :: \overrightarrow{G_{n-1}})] \\ \qquad \text{if } \mathit{prog?}(G_n :: \overrightarrow{G_{n-1}}) \\ \bot \quad \text{otherwise} \end{cases} \\
&\text{where } (\overrightarrow{V_{n-1}}, \overrightarrow{G_{n-1}}) \equiv M(V) \\
&\text{and } G_n = \mathit{graph}(\overrightarrow{V_{n-1}}, \overrightarrow{V_n})
\end{aligned}
$$

$$
\begin{aligned}
\mathit{graph} &: \overrightarrow{V} \times \overrightarrow{V} \to G \\
\mathit{graph}(\overrightarrow{V}, \overrightarrow{V'}) &= \{(i \downarrow j) \mid V_i \in \overrightarrow{V}, V_j \in \overrightarrow{V'}, V_j \prec V_i\} \\
&\cup \{(i \overline{\downarrow} j) \mid V_i \in \overrightarrow{V}, V_j \in \overrightarrow{V'}, V_j = V_i\}
\end{aligned}
$$

$$
\begin{aligned}
(;) &: G \times G \to G \\
G_0 \; ; \; G_1 &= \{(i \downarrow k) \mid (i \downarrow j) \in G_0, (j \; R \; k) \in G_1\} \\
&\cup \{(i \downarrow k) \mid (i \; R \; j) \in G_0, (j \downarrow k) \in G_1)\} \\
&\cup \{(i \overline{\downarrow} k) \mid (i \overline{\downarrow} j) \in G_0, (j \overline{\downarrow} k) \in G_1, \\
&\qquad \nexists j.(i \; \downarrow \; j) \in G_0 \wedge (j \; R \; k) \in G_1, \\
&\qquad \nexists j.(i \; R \; j) \in G_0 \wedge (j \; \downarrow \; k) \in G_1\}
\end{aligned}
$$

$$
\begin{aligned}
\mathit{prog?} &: \overrightarrow{G} \to \mathbb{B} \\
\mathit{prog?}(G_n ... G_1) &= \bigwedge\nolimits_{1 \le i \le j \le n} \mathit{desc?}(g_i; ...; g_j)
\end{aligned}
$$

$$
\begin{aligned}
\mathit{desc?} &: G \to \mathbb{B} \\
\mathit{desc?}(G) &= (G = G;G) \implies \exists i.(i \downarrow i) \in G
\end{aligned}
$$

Figure 4.6: Updating and monitoring size-change

termination principle: a graph sequence $G_n ... G_1$ violates the size-change principle if there exists a sub-sequence $G_i; ...; G_j$ (where $1 \le i \le j \le n$) that is both idempotent and lacking of an strict descending arc of a parameter to itself.

4.2.8 Well-founded Partial Order

Figure 4.7 shows an example of a well-founded partial order ($\preceq$) on values in λ_{SCT}. It is defined on integers by comparing absolute values, and a field of a data-structure is considered smaller than any data-structures that contain it (i.e.,

$$
\begin{array}{rcll}
\prec, \preceq & \subseteq & V \times V & \\
N_1 & \prec & N_2 & \text{if } |N_1| < |N_2| \\
V & \prec & (V', \_) & \text{if } V \preceq V' \\
V & \prec & (\_, V') & \text{if } V \preceq V' \\
V & \preceq & V' & \text{if } V \prec V' \text{ or } V = V'
\end{array}
$$

Figure 4.7: Example well-founded partial order $\preceq$

the tail of any list is considered *less than* than the original list). Although simple, this relation is sufficient to check for termination in most programs that descend on integers and data-structures. If a program descends following a different order, the user of λ_{SCT} can replace the default order with an appropriate one.

4.2.9 Totality of Evaluation

We may note that all programs in λ_{SCT} terminate, either by adhering to the size-change principle, or by violating it and aborting with an error.

Theorem 4 (Termination of λ_{SCT})**.** *For all* E, P, M, *where* $fv(E) \subseteq dom(\mathrm{P})$, $\mathrm{P}, M \vdash E \Downarrow \mathrm{A}$ *for some* A.

4.2.10 Soundness and Completeness

The size-change property is a safe over-approximation to ensure termination. The correctness of monitoring this property can therefore be understood as any strategy that satisfies the following properties:

soundness: if a program evaluates to a value under the modified semantics, running the program without termination checking gives the same result.

SCT-completeness: if a program terminates and maintains the under the standard semantics, running that program under the modified size-change property un-

der the standard semantics, running that semantics with termination checking gives the same result. program under the modified semantics with termination checking gives the same result.

In addition, because all programs terminate under the modified semantics when termination checking is enabled, all diverging programs are caught as error-raising programs.

We now formally establish the correctness of λ_{SCT}'s size-change monitoring semantics with respect to its standard semantics.[1]

Theorem 5 (Soundness of size-change monitoring in λ_{SCT})**.** *If* $\text{P}, M \vdash E \blacklozenge A$*, then* $\text{P} \vdash E \Downarrow A$*.*

Proof. *By induction on the derivation of* $\text{P}, M \vdash E \blacklozenge A$*.*

Corollary 2 (Size-change monitoring catches divergence)**.** *If program* E *diverges under the standard semantics, then* $\{\}, \{\} \vdash E \blacklozenge \mathsf{error}^{\mathsf{SC}}$*.*

Proof. *From Theorem 4, E either evaluates to a standard answer or* $\mathsf{error}^{\mathsf{SC}}$ *under size-change monitoring. By contrapositive of Theorem 5, E evaluates to* $\mathsf{error}^{\mathsf{SC}}$ *if E diverges.*

A SEMANTICS THAT PRODUCES CALL SEQUENCES Before stating and proving completeness of size-change monitoring, we define a mostly-standard semantics that also evaluates to set of size-change tables along with the answer, but performs no guarding against any size-change violation. It is in lock-step with the standard semantics, and resembles the terminating semantics in accumulating the size-change table. Figure 4.8 shows this semantics.

Lemma 2 (Completeness of call-sequence semantics)**.**
If $\text{P} \vdash E \Downarrow V$ *then* $\text{P}, \{\} \vdash E \Downarrow V, \{M \ldots\}$ *for some* $\{M \ldots\}$*.*

[1] λ_{SCT}'s standard dynamic semantics is unsurprising and can be found in the supplemental material.

$$\textsc{CC-Base}\quad \frac{}{\mathrm{P}, M \vdash N \Downarrow N, \{M\}}$$

$$\textsc{CC-App}\quad \frac{\mathrm{P}, M \vdash E \Downarrow \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \{M' \ldots\} \qquad \mathrm{P}, M \vdash \overrightarrow{E_x} \Downarrow \overrightarrow{V_x}, \overrightarrow{\{M_x \ldots\}} \quad \mathrm{P}'[\overrightarrow{X \mapsto V_x}], \mathit{ext}(M, \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \overrightarrow{V_x}) \vdash E' \Downarrow A, \{M'' \ldots\}}{\mathrm{P}, M \vdash (E\ \overrightarrow{E_x}) \Downarrow A, \{M' \ldots\} \cup \overrightarrow{\{M_x \ldots\}} \cup \{M'' \ldots\}}$$

$$\begin{array}{rcl} \mathit{ext} & : & M \times V \times \overrightarrow{V} \to M \\ \mathit{ext}(M, V, \overrightarrow{V}_n) & = & M[V \mapsto (\overrightarrow{V}_n, [])], \text{ if } V \notin M \\ \mathit{ext}(M, V, \overrightarrow{V}_n) & = & M[V \mapsto (\overrightarrow{V}_n, G_n :: \overrightarrow{G}_{n-1})] \\ \text{where} & & (\overrightarrow{V}_{n-1}, \overrightarrow{G}_{n-1}) \equiv M(V) \\ \text{and} & & G_n = \mathit{graph}(\overrightarrow{V}_{n-1}, \overrightarrow{V}_n) \end{array}$$

Figure 4.8: Call-sequence Semantics of λ_{SCT}.

Proof. *By induction on the derivation of* $\mathrm{P} \vdash E \Downarrow V$.

Lemma 3 (Completeness of s.c. monitoring w.r.t. call-sequence semantics)**.** *If* $\mathrm{P}, M \vdash E \blacktriangleright \mathsf{error}^{\mathsf{SC}}$ *and* $\mathrm{P}, M \vdash E \Downarrow V, \{M' \ldots\}$ *then there exists* M_i *in* $\{M' \ldots\}$ *and* V *such that* $\neg \mathit{prog?}(G)$ *where* $(\overrightarrow{V_x}, G) = M_i(V)$.

Proof. *By induction on the derivation of* $\mathrm{P}, M \vdash E \blacktriangleright \mathsf{error}^{\mathsf{SC}}$.

Theorem 6 (Completeness of size-change monitoring in λ_{SCT})**.** *If* $\mathrm{P}, \{\} \vdash E \blacktriangleright \mathsf{error}^{\mathsf{SC}}$ *and* $\mathrm{P} \vdash E \Downarrow V$ *then* $\mathrm{P}, \{\} \vdash E \Downarrow V, \{M \ldots\}$ *such that there exists* M_i *in* $\{M \ldots\}$ *and* V *such that* $\neg \mathit{prog?}(G)$ *where* $(\overrightarrow{V_x}, G) = M_i(V)$.

Proof. *Follows from Lemma 2 and Lemma 3.*

4.2.11 Termination Checking as a Contract

It can be useful to enforce termination checking selectively on parts of the code rather than on the entire program. We present a simple extension to λ_{SCT} called λ_{CSCT}, which adds a construct $(\mathsf{term/c}\ E)$ that guards (E) with a contract ensuring it behaves as a size-change-terminating function. Other than executing the bodies of contract-guarded functions, the λ_{CSCT} semantics is similar to the standard

$$\begin{array}{rl} \text{[Expressions]} & E ::= \ldots \mid (\mathsf{term/c}\ E) \\ \text{[Values]} & V ::= \ldots \mid \mathsf{term/c}(\mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})) \end{array}$$

$$\textsc{Wrap-Lam}\quad \frac{\mathrm{P} \vdash E \Downarrow \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})}{\mathrm{P} \vdash (\mathsf{term/c}\ E) \Downarrow \mathsf{term/c}(\mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P}))}$$

$$\textsc{App-Term}\quad \frac{\mathrm{P} \vdash E \Downarrow \mathsf{term/c}(\mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}')) \qquad \mathrm{P} \vdash \overrightarrow{E_x} \Downarrow \overrightarrow{V_x} \qquad \mathrm{P}'[\overrightarrow{X \mapsto V_x}], \mathit{upd}(\{\}, \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \overrightarrow{V_x}) \vdash E' \blacktriangleright \mathrm{A}}{\mathrm{P} \vdash (E\ \overrightarrow{E_x}) \Downarrow \mathrm{A}}$$

Figure 4.9: Syntax and semantics of λ_{CSCT}.

semantics. Figure 4.9 shows the key extension to λ_{CSCT}'s syntax and semantics. The *[Wrap-Lam]* rule shows the introduction of a termination-checked function. Only closures are capable of violating SCT in λ_{SCT}, so we only wrap closures and return other values as-is.

4.3 Static SCT Verification

Given termination formulated as a dynamically checkable property, we can systematically turn these dynamic checks into static verification by building on prior work in higher-order symbolic execution [9, 18, 25, 99].

Symbolic execution extends the standard semantics with symbolic values that can stand for any values (including higher-order values), and maintains a path-condition, which is a formula about facts that must hold for symbolic values on each path. Because termination checks ultimately decompose into "less-than" checks, which check for a definite descent of values along a well-founded partial order, there

is no special challenge in using symbolic execution for size-change termination checking. Symbolic execution can readily leverage SMT solvers for precise reasoning about path-conditions, proving termination that depends on sophisticated path-sensitivity.

Although symbolic execution has traditionally been used to find bugs [47, 58–61] as opposed to verifying programs as correct, we can apply a well studied technique for abstracting the operational semantics through finitizing the program's dynamic components [24, 25] and obtain a verification that particular errors cannot occur at run-time.

4.3.1 Extended Semantics

Figure 4.10 shows extension to λ_{SCT}, called λ_{SSCT} that allows symbolic execution, as well as the key extension to the semantics that enables symbolic execution.

We extend the set of values (V) with *symbolic values* (S), which can stand for any value. The semantics of λ_{SSCT} must then account for symbolic values, which means some expressions can non-deterministically evaluate to multiple answers to soundly over-approximate all the cases resulting from possible instantiations of symbolic values. Symbolic execution maintains a *path-condition* (Φ) that characterizes each path, which is a set of symbolic values assumed to have evaluated to true (interpreted as a conjunction).

With symbolic values, orders between values are necessarily more conservative. The size-change graphs computed between symbolic value lists in Figure 4.6 have, in general, no more arcs than in the concrete case. Each arc now represents a must-descend or must-non-ascend relationship over all possible concrete paths.

$$
\begin{array}{rl}
\text{[Values]} & V ::= \ldots \mid S \\
\text{[Symbolic Values]} & S ::= X \mid N \mid (O\ \vec{S}) \\
\text{[Path Conditions]} & \Phi = \vec{S}
\end{array}
$$

$$\textsc{Sym-If-T}\quad \frac{\mathrm{P}, \Phi \vdash E \Downarrow_s S, \Phi' \qquad \mathrm{P}, (= S\ 0) :: \Phi' \vdash E_1 \Downarrow_s \mathrm{A}, \Phi''}{\mathrm{P}, \Phi \vdash (\mathsf{if0}\ E\ E_1\ E_2) \Downarrow_s \mathrm{A}, \Phi''}$$

$$\textsc{Sym-If-F}\quad \frac{\mathrm{P}, \Phi \vdash E \Downarrow_s S, \Phi' \qquad \mathrm{P}, (\neq S\ 0) :: \Phi' \vdash E_2 \Downarrow_s \mathrm{A}, \Phi''}{\mathrm{P}, \Phi \vdash (\mathsf{if0}\ E\ E_1\ E_2) \Downarrow_s \mathrm{A}, \Phi''}$$

Figure 4.10: Semantics of symbolic λ_{SSCT}.

A sufficiently precise symbolic execution, coupled with effective SMT solving, can maintain a graph with enough arcs to prove that functions will always maintain their size-change properties.

Proposition 1 (Soundness of static verification)**.** *If* $\{\} \vdash E \Downarrow V$ *and* $\{\} \vdash E_1 \Downarrow V_1$ *and* $(E\ E_1)$ *diverges, then* $\{\}, \{\} \vdash ((\mathsf{term/c}\ E)\ S) \Downarrow_s \mathsf{error}^{\mathsf{SC}}, \Phi'$ *(*S *is a fresh symbolic value).*

Proof. *Follows from soundness of dynamic checking of size-change termination (Theorem 5) and soundness of higher-order symbolic execution (Theorem 3).*

4.3.2 Ackermann Revisited

Now consider again the example `ack`, a termination-checked Ackermann function shown in Section 4.2.

Suppose `ack`'s precondition is that its arguments are natural numbers. To verify `ack`, we apply the function on symbolic natural numbers `m` and `n` that have passed `ack`'s precondition with the path-condition `{(≥ m 0), (≥ n 0)}`. With

these symbolic inputs, execution considers all three branches, and accumulates in the path-condition assumptions about the values: in the first branch, `m` is `0`; in the second branch, `m` is positive and `n` is `0`; in the last branch, both `m` and `n` are positive.

The first branch simply returns and does not trigger any size-change monitoring. The second branch reaches a recursive call with the path-condition {(≥ m 0),(≠ m 0),(= n 0)}. The recursive call proceeds, checking for all relationships that can be established between the old and new arguments as in Figure 4.6. In this case, with the path-condition that `m` is positive, symbolic execution easily proves that `(- m 1)` is less than `m` according to the partial order defined in Figure 4.7. No other definite order can be established between the new arguments `(- m 1)`, `1` and the old ones `m`, `n`. This gives the new size-change graph of $\{(m \downarrow m)\}$.[2] We extend `ack`'s set of size-change graphs with this new graph. In addition, symbolic execution can prove the new call to `ack` receives the same path-condition as the previous call: both new arguments `(- m 1)` and `1` are natural numbers.

The third branch reaches the inner recursive call to `ack` before reaching the outer one. The path-condition, again, is sufficient for establish the descent from `n` to `(- n 1)` and maintenance of `m`, yielding the new graph $\{(m \overline{\downarrow} m), (n \downarrow n)\}$. When execution reaches the outer recursive call to `ack`, the descent from `m` to `(- m 1)` can be straightforwardly established. In each case, symbolic execution can also prove that the new arguments are natural numbers.

Figure 4.11 summarizes all the ways `ack` can call itself recursively. Because no composition of size-change graphs drawn from this set can yield a graph that

[2] We use variable names instead of indices for graph nodes in this section.

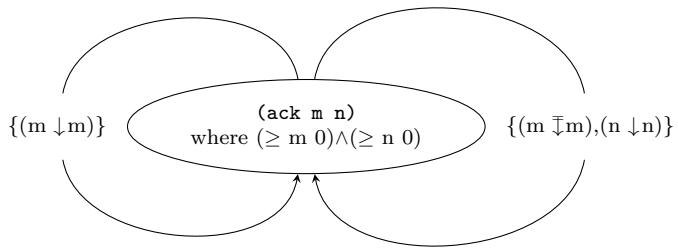

Figure 4.11: Abstract call and size-change graphs for `ack`.

violates the size-change principle (i.e. one that is both idempotent and lacking of a self-descent arc), `ack` never violates size-change termination.

4.4 Implementation and Evaluation

We implement the semantics presented in Section 4.2.5 as a library in the Racket programming language through instrumentation of the application form, and also apply the contract verifier developed from chapter 3 to statically verify this semantics.

We then evaluate the effectiveness and efficiency of size-change monitoring. Effective monitoring should allow all or most terminating programs to finish execution, and quickly catch diverging programs. Efficient monitoring should introduce little overhead compared to execution without monitoring. Finally, we evaluate the contract verifier, now capable of verifying total correctness, on its effectiveness when used as a theorem prover.

4.4.1 Implementation

An application `(f x ...)` in Racket is syntactic sugar for `(#%app f x ...)`, and libraries can modify what an application means by redefining the `#%app` form. For our purposes, we redefine the application form to implement the rules [SC-App-Clo] in Figure 4.5 [App-Term] in Figure 4.9. If size-change termination is being enforced, the `#%app` form looks up the size-change table to guard against violations.

We evaluate two techniques to maintain size-change tables. The first technique wraps each application with code that imperatively updates and restores the table. The second uses continuation-marks [100]. The former can be implemented in most languages, and gives relatively good performance, but breaks proper tail calls. The latter is simple to implement in languages with support for continuation marks, and preserves tail calls, but shows high overheads in tight loops.

Our semantics implicitly assumes that closures can be compared structurally for equality, which is not possible in practice. We instead hash the closure and consider all closures with the same hash code to be equivalent. This preserves soundness as the table m cannot grow infinitely, but could produce false positive error reports. Note that this incompleteness does not affect the static analysis, which is derived from the semantics itself. Future work includes run-time support for more precise comparison between closures.

In addition, we expose a parameter specifying the custom partial order for use in termination checks, with a default implementation as described in Figure 4.7.

Although a naive implementation would be prohibitively expensive, with a

few optimizations, the overhead can be brought down to acceptable for the goal of debugging

Reducing monitoring frequency The construction and checking of size-change graphs is expensive, but need not be performed each time a function calls itself recursively. Because strict progress down any well-founded partial order can only be maintained a finite number of times, any non-SCT program will violate the size-change principle regardless of the monitoring frequency. We therefore use exponential back-off to reduce the frequency of extending and monitoring each function's size-change. This significantly reduces the monitoring overhead, although risks keeping data from earlier iterations live for longer necessary.

Avoiding instrumentation for known functions Functions that are known to terminate need no instrumentation. We maintain a white-list of primitives known to terminate.

Monitoring size-change graphs only for loop entries We identify "loop entries" to monitor instead of constructing and monitoring a size-change graph for each function. For example, suppose `even?` and `odd?` are mutual recursive functions, where the top-level context calls `even?`, then only `even?` is a loop-entry and requires size-change monitoring.

4.4.2 Effectiveness and Efficiency on Terminating Programs

Table 4.1 shows terminating programs we use to evaluate the dynamic checks and static analysis of terminating contracts. The programs were collected from previous work on termination checking: size-change termination for first-order programs (sct) [27]; size-change termination for higher-order programs (ho-sct) [28]; Liquid Haskell (lh) [6]; Isabelle [29]; ACL2 [30]; and a collection of larger Scheme benchmarks that terminate by the size-change principle.

The table shows the precision of dynamic checking and static analysis, as well as comparison with other systems where possible. Most programs are small and under 15 lines. The largest program is `scheme` with 1,100 lines, which implements an interpreter for R5RS Scheme that interprets the mergesort algorithm on a list of strings. We did our best efforts to translate programs from one system to another. For example, `sct-2` is originally an untyped program composing a heterogeneous list which cannot be typed in Liquid Haskell and Isabelle. We translated `sct-2` to work with an equivalent custom tree data-type.

Several cases where the programs need modifications to be successfully verified by the systems are annotated in the table. For example, `sct-1` and `sct-2` originally use conditionals, and can only be verified when converted to use pattern-matching. Some other programs are only verified successfully with annotational help on termination, such as explicit lexical ordering (e.g. `lh-merge`), or a custom partial-order (e.g. `acl2-fig-2`). Some programs are not expressible in all systems. For example, ACL2 cannot check higher-order programs, and the type systems in Liquid

Program	Dyn.	Static	LH	Isabelle	ACL2
sct-1 (rev)	✓	✓	✓^{R}	✓	✓
sct-2	✓	✓	✗	✓^{R}	✓
sct-3 (ack)	✓	✓	✓^{A}	✓	✓
sct-4	✓	✓	✗	✓	✓
sct-5	✓	✓	✗	✓	✓
sct-6	✓	✓	✗	✓	✓
ho-sc-ack	✓	✗	-^{T}	-^{T}	-^{H}
ho-sct-fg	✓	✓	✓	✓	-^{H}
ho-sct-fold	✓	✓	✓^{A}	✓	-^{H}
isabelle-perm	✓	✓	✗	✓	✓
isabelle-f	✓	✗	✗	✓	✓
isabelle-foo	✓	✗	✗	✓	✓
isabelle-bar	✓	✗	✗	✓	✓
isabelle-poly	✓	✗	✗	✗	✗
acl2-fig-2	✓^{O}	✗	✗	✗	✗
acl2-fig-6	✓	✓	✗	✗	✗
acl2-fig-7	✓	✗	✗	✗	✓
lh-gcd	✓	✗	✓	✓	✓
lh-map	✓	✓	✓	✓	-^{H}
lh-merge	✓	✓	✓^{A}	✓	✓
lh-range	✓^{O}	✗	✓^{A}	✗	✓
lh-tfact	✓	✓	✓	✓	✓
dderiv	✓	✓	A: With annotations		
deriv	✓	✗	O: Custom partial order		
destruct	✓	✗	H: No H.O. functions		
div	✓	✓	T: Not typable		
nfa	✓	✓	R: Rewritten to use		
scheme	✓	✗	pattern matching		

Table 4.1: Evaluation on terminating programs

Haskell and Isabelle do not support the Y-combinator that has self-applications (e.g. `ho-sct-ack`). To our surprise, current versions of the tools cannot check some of their own benchmarks despite our best efforts to reproduce (e.g. `isabelle-poly` for Isabelle; `acl2-fig-2` and `acl2-fig-6` for ACL2). Overall, our system works well for a wide range of programs and idioms, including higher-order untyped programs with moderate side effects (such as in the Scheme benchmarks).

Figure 4.12 shows the slowdown of dynamic checks for select programs: `factorial`, `sum`, and `merge-sort`, as well as their interpreted version inside a Scheme interpreter. These programs demonstrate that different patterns of computation incur different amounts of overhead from size-change monitoring. For programs that do significant work between recursive calls, such as `factorial` or the Scheme interpreter, overhead is negligible. For programs that don't do significant work between recursive calls, such as `sum`, the overhead is significant. For programs that operate over large data-structures such as `merge-sort`, overhead is much more significant. That the overhead stays fixed when the input grows (for continuation-mark implementation on tight loops, approximately two orders of magnitude) suggests that further optimization effort to trim down the constant factor can make monitoring suitable for realistic uses.

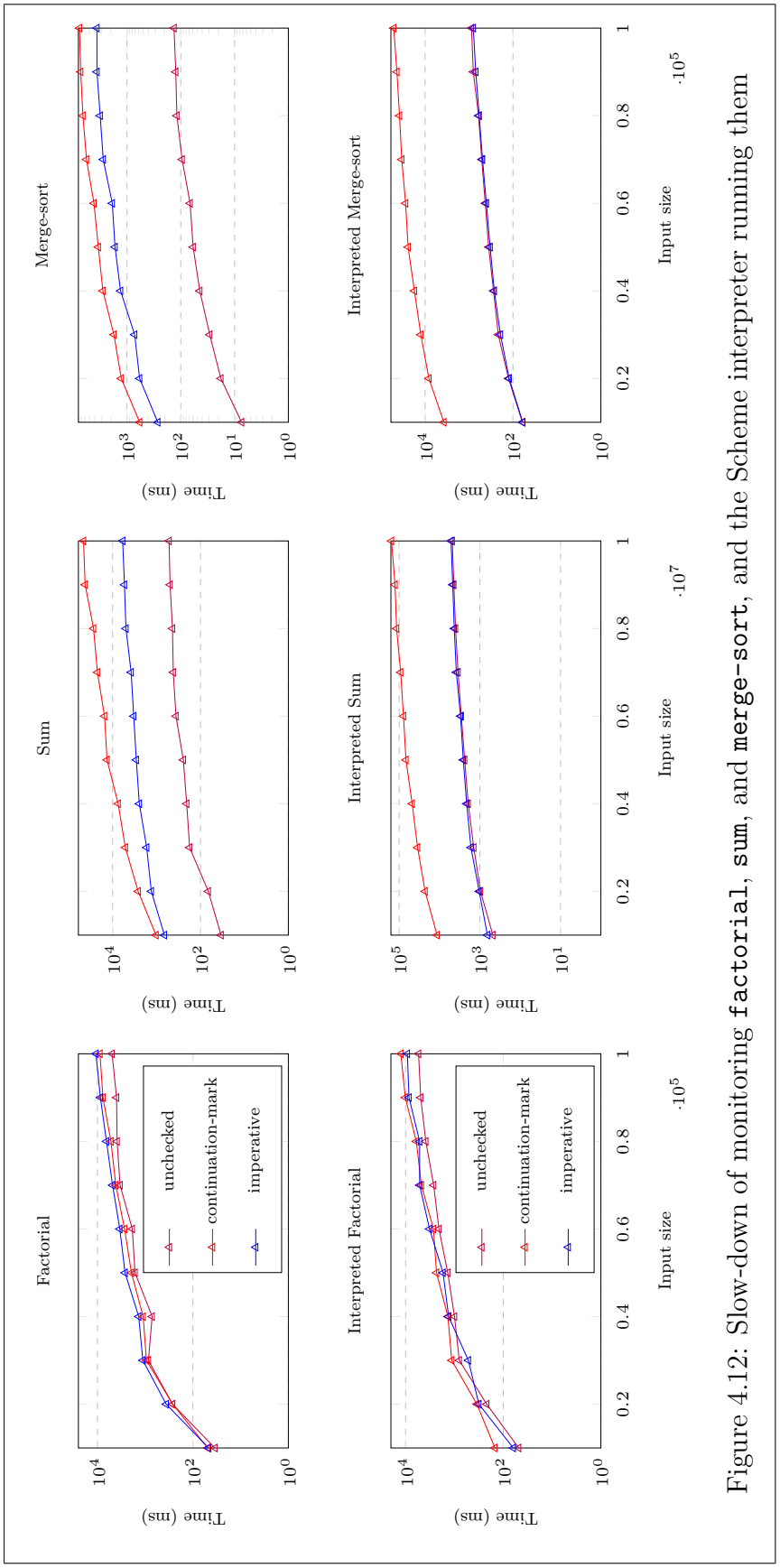

Figure 4.12: Slow-down of monitoring `factorial`, `sum`, and `merge-sort`, and the Scheme interpreter running them

4.4.3 Effectiveness on Diverging Programs

We also evaluate dynamic monitoring on non-terminating programs to determine how quickly the monitoring system catches divergence. These programs include modified versions of correct programs, as well as one originally incorrect program (`nfa`) that our static analysis discovered. Because violation of the size-change principle tend to show up in early iterations, our dynamic contracts catch the error very early, resulting in immeasurable delay from the start of the program to the point where divergence is detected.

The `nfa` program is particularly interesting, because it is a Scheme benchmark that has been around for decades. It is a program that implements a non-deterministic finite automaton of the regular expression `((a|c)*bcd)|(a*bc)`, then run the automaton on the string $\texttt{a}^{133}\texttt{bc}$. The function shown in figure 4.13 implements one state recognizing the sub-expression `(a|c)*` with the bug underlined.

The bug was never discovered, because the particular benchmark input did not trigger the divergence, and most static analysis only check for partial correctness. Our static analysis was the first to discover this error after many years.

```
1 (define (state1 input)
2   (and (not (null? input))
3        (or (and (char=? (car input) #\a)
4                 (state1 (cdr input)))
5            (and (char=? (car input) #\c)
6                 (state1 input))
7            (state2 input))))
```

Figure 4.13: Buggy function in `nfa` benchmark

4.4.4 Theorem Proving as Total-contract Verification

We evaluate the contract verifier's effectiveness as a theorem prover, taking advantage of total contracts. In particular, total contracts can be viewed as propositions referring to values in the programs, and values that satisfy these contracts are proofs. The ability to use total contracts as theorems empowers programmers with a new means of stating invariants. In particular, properties about multiple runs of functions cannot be expressed as contracts on those functions. Figure 4.14 shows an instance of such properties, where no contract can enforce that a function is monotonic. Instead, we state function `inc`'s monotonicity externally as `(monotonic/c inc)`, and provide a proof `inc-is-monotonic` that confirms this fact holds for any pair of integers `m` and `n`. In this case, the proof returns an arbitrary value `'trivial` that is computationally uninteresting, and the contract verifier can easily verify the "side condition" `(implies (≤ m n) (≤ (f m) (f n)))`.

The contract verifier can verify more interesting properties such as associativity of `append` or distributivity of `map` over `append`. In these cases, the proof only needs to call itself recursively on a smaller list in order to bring in the inductive hypothesis. Figure 4.15 shows what the proofs look like. In effect, proofs only need to describe the high-level structure, and the contract verifier, possibly with the help of an underlying SMT solver, can take care of simple details such as congruence closure and arithmetic.

Using the contract verifier for theorem proving, we are able to prove common properties of functions on lists, the most interesting so far being insertion sort pro-

```
;; inc's property not enforcable as inc's own contract:
;; ∀m,n, m ≤ n ⟹ inc(m) ≤ inc(n)
(define inc (λ (n) (+ n 1)))

;; Proposition as syntactic sugar for total dependent function contract
(define-syntax-rule (∀ ([x c] ...) prop)
  (and/c terminating/c
         (->i ([x c] ...)
              (_ {x ...} (λ (_) prop)))))

;; Proposition ``maker''
(define (monotonic/c f)
  (∀ ([m int?] [n int?]) (implies (≤ m n) (≤ (f m) (f n)))))

;; Proposition on `inc`
(define/contract inc-is-monotonic (monotonic/c inc)
  (λ (m n) 'trivial))
```

Figure 4.14: Monotonicity stated externally

ducing an ordered list. Even though the tool allows concise and high-level proofs, one main shortcoming is that it is unintuitive when it fails unless the programmer is familiar with the inner workings of the tool. For example, the proof may fail because the omitted details involve a theory not supported by the underlying solver, or because it does not perform a case analysis on some data, which symbolic execution may not automatically do due to the risk of path explosions. More predictable theorem proving as well as more helpful error messages are important future work in order to make the contract verifier practical as a theorem prover.

4.5 Related Work

Our work builds on the size-change termination (SCT) approach [27] and on static contract verification via symbolic execution (chapter 3). We relate our con-

```
(define map ...)
(define append ...)

(define/contract append-assoc
  ;; Claim
  (∀ ([xs list?] [ys list?] [zs list?])
    (equal? (append xs (append ys zs))
            (append (append xs ys) zs)))
  ;; Proof
  (λ (xs ys zs)
    (match xs
      ['() 'trivial]
      [(cons _ xs*) (append-assoc xs* ys zs)])))

(define/contract map-distr-append
  ;; Claim
  (∀ ([f (and/c terminating/c (any/c -> any/c))]
      [xs list?]
      [ys list?])
    (equal? (map f (append xs ys))
            (append (map f xs) (map f ys))))
  ;; Proof
  (λ (f xs ys)
    (match xs
      ['() 'trivial]
      [(cons _ xs*) (map-distr-append f xs* ys)])))
```

Figure 4.15: Proofs of functions on lists

tributions to dynamic and static termination checking, and then to static contract verification.

4.5.1 Dynamic Termination Checking

To the best of our knowledge, no existing work enforces termination dynamically using behavioral contracts. Related work has investigated dynamic loop detection, non-termination auditing, and more restricted declarative languages.

The auditing tool LOOPER [101] dynamically monitors a Java program in order to detect non-termination using concolic (concrete and symbolic) execution. Along the path of a potentially non-terminating loop, it derives a path condition paired with a memory map (an encoding of heap values at the end of a loop iteration as a function of their initial values), and uses an SMT solver to check if the initial path condition (after zero iterations) implies itself under the loop iteration's memory map. If this fails, LOOPER will observe another iteration and record a new path condition and memory map. When each path condition implies the next (under that iteration's memory map), in a cyclic chain that terminates with the original path condition, the program will not terminate.

Unlike our contracts, LOOPER does not monitor code for non-termination during normal execution; instead, it is deployed by an auditor to determine whether an apparent loop is an actual one. While LOOPER can provide an affirmative proof that code will not terminate, our approach will signal that a function does not obey SCT, a more conservative notion of termination. This means our approach is

susceptible to false positives and may blame functions which do always terminate, but will never permit non-termination. LOOPER, on the other hand, is susceptible to false negatives and may fail to prove an execution to be definitively non-terminating. LOOPER's soundness is also contingent on all changes to memory being visible and accounted for in the memory map, which is not always the case in C due to external state and shared-memory parallelism.

JOLT [102] (and successor BOLT [103]) is an infinite-loop detection and recovery tool for C programs. It instruments C code to dynamically monitor for loops that are in the exact same state at two consecutive iterations. Compared with LOOPER, this is an especially conservative detection for non-termination, however JOLT also has a facility for skipping the program counter past the end of the loop to recover from non-termination and show that this simple technique is effective in many cases (sometimes depending on inputs).

There are also dynamic termination schemes for more restricted languages. For example, dynamic checking for active database rules [104], or queries in general logic programs [105, 106]. [105] exploits features unique to SLDNF-trees to identify loop goals with a provably finite term-size. [106] provides a declarative fixed-point semantics that captures termination properties (for an interpretation of Prolog) with the explicit goal of facilitating the extraction of a static analysis using abstract interpretation.

4.5.2 Static Termination Checking

A variety of approaches have been used for static verification of termination and non-termination. None of these systems combine dynamic and static verification in a single system, or allow terminating and non-terminating components to be composed. We begin with the systems we compare with in §4.4.2.

[97] extend the SCT approach to higher-order languages—specifically, the untyped λ-calculus. As all values in this language are functions, they select the "height" of a closure as its size. [28] then extended this approach to handle user-defined data-types and general recursion. This work was not empirically evaluated in the context of a real programming system [107], but establishes techniques we build on. SCT has been extended to monotonicity constraints, which have been shown to be more general than traditional SCT [108]; these could be formulated as a dynamic contract in future work.

[30] develops a static analysis for automatic termination proofs in the context of the ACL2 system—a functional language and first-order logic for theorem proving. All programs admitted by ACL2 must be terminating, as non-termination could render it inconsistent, however manual termination proofs are complex and require deep expertise. The paper's approach uses precise calling-context graphs in order to refine static control flow with path feasibility based on accumulating governors (sets of branch points governing control flow for a sub-expression). Strongly connected components are then further refined using a calling-context measure in order to discover a well-founded order over which parameters descend. A major innovation on

traditional SCT approaches is the refinement of feasible paths using governors. Our approach analogously tracks path conditions for static verification. Their method was effective at proving more than 98% of the roughly 10k functions of the ACL2 regression suite terminating. [29] then extends the approach to Isabelle/HOL and certifies the termination proofs with LCF-style theorem proving.

LiquidHaskell uses termination proving to ensure precision and soundness for its refinement type system in the presence of lazy evaluation [109]. Subtle unsoundness can result from using refinement types in conjunction with call-by-name evaluation and the direct approach to fixing this unsoundness, by expressing potential non-termination as a type refinement, leads to substantial imprecision. LiquidHaskell bridges this gap by encoding size-change invariants, over user-specified well-founded metrics, directly into the existing type system (as further type refinements). This permits proofs over programs to circularly depend on termination proofs during SMT solving. Broadly this same approach is taken to directly encode termination proofs, via size-change refinements, with dependent types in DependentML [110]. LiquidHaskell has a scalable implementation, used to verify correctness and termination properties over a corpus of real-world Haskell libraries ($\geq 10k$ LOC). TEA is also a termination analysis for Haskell, based on techniques of path analysis and abstract reduction [111].

TNT is a concolic executor for statically enumerating non-terminating *lassos* in C programs—paths that fold back on themselves, forming a non-terminating loop [112]. statically precise enough to handle cases that rely on symbolic shape information such as cyclic lists.

[113] use a modal logic allowing predicates to be written that are qualified by a program expression they pertain to. Qualified formulae are trivially rendered true by a diverging program, so a manifest contradiction (i.e., false) being interpreted as true constitutes a proof of non-termination for the qualifying expression. The approach then uses a refinement process to identify the specific conditions on data that will lead to proving this contradiction. This system was only evaluated on small expressions ($\leq$25 lines) in a language of pure built-in expressions, assignments, conditionals, and `while` loops.

AProVE is a system for automating termination (and non-termination) proofs of term-rewriting systems (TRSs) [114–116] built using the dependency pair framework [117, 118]. Unlike previous methods for proving TRSs terminating, which required the right-hand side of each rewrite rule to be simplified compared with its left-hand side, the dependency pair framework only requires corresponding subterms at recursive calls be simplified. This innovation is analogous to the SCT approach's requirement that arguments be descending over some well-founded order as opposed to static control-flow being strictly stratified. [119] extends the dependency pair framework to higher-order functions. [120] contrast and synthesize the dependency pair framework with SCT.

Numerous techniques have been proposed and evaluated for verifying termination in languages such as C and Java, where higher-order programming is uncommon. Terminator [95, 121, 122] and transition invariants [123–125] as well as others [126–129] have seen extensive development, and share some key ideas with our approach, but differ substantially in goals and language from our system, and

thus make significantly different choice in approach.

TERMINATOR is a program analysis and verification tool for proving termination of C programs statically, which has been used to prove the termination of low-level programs such as Windows device drivers. Like our system, it relies on the "indirect approach" described in the introduction—find a safety property which implies termination, add a check for that property to the program, and verify using an existing tool that the check cannot fail. The key difference with our approach is in the choice of property. TERMINATOR aims to prove termination of tricky first-order loops, and thus must find potentially-complex custom ranking functions (found via [122]) for each program to be verified. To find these properties, it relies on a counter-example guided abstraction refinement (CEGAR) [130] loop which attempts to verify termination using an off-the-shelf verifier and refines the property upon failure. Terminator starts with a very simple property and repeatedly improves it, generating complex predicates with non-trivial relationships between multiple variables. In contrast, our approach (similar to other approaches for higher-order languages) picks a *single* general safety property and uses it for all programs. This limits the ability of our tool to verify the termination of loops such as those TERMINATOR aims at, but allows our tool to run as a contract without first requiring several runs of a static verifier. Additionally, constructing static verification tools for heap-manipulating imperative programs is much trickier in the higher-order setting we consider.

4.6 Conclusion

Termination is a fundamental program correctness property, but uncheckable even at run-time. To avoid this limitation, we adapt the size-change principle from static termination analysis to perform dynamic checking of termination, exploiting the insight that every infinite execution must have a call that fails to follow the size change principle. This leads to the first run-time mechanism for enforcing termination in a general-purpose programming system. As it is formulated as a behavioral contract, this also makes it the first contract for total correctness. By checking termination as a contract, we can enforce termination in settings where static checking is fundamentally impossible, as in an interpreter.

Further, we compose our dynamic checking strategy with prior work showing how to statically verify compliance with contracts in higher-order languages to produce a novel static checker for program termination—without any termination-specific work. We compare our static checker against three state-of-the-art custom tools on their own benchmarks, and find that ours is able to statically verify programs that exceed the capacities of each of the existing tools.

Sound dynamic enforcement of liveness properties opens up new possibilities for program correctness, analysis, and specification—in this chapter we have taken only the first step.

Chapter 5: Future Work and Conclusion

This section explores future work enabled by higher-order symbolic execution and concludes.

5.1 Future work

Future directions of this line of work includes applying it to program verification and optimization, as well as improving symbolic execution itself.

Overhead elimination for sound gradual typing A gradual type system [131–134] allows mixing typed and untyped modules in a single program. To ensure soundness, the type system generates contracts to protect the typed world whenever it interacts with the untyped one, guarding it against values flowing from the untyped that violate invariants ensured by the type system. Experiments show that such mechanism can be prohibitively expensive [135], and some systems weaken soundness guarantees to achieve practical performance [136]. By applying this work to verify the untyped modules against contracts generated by typed ones, we should be able to eliminate all or most of the overhead from soundness enforcement without requiring any work from the programmers to modify the programs.

More sophisticated notions of termination Size-change termination is particularly simple and robust: It is easy to formulate as a run-time check and can capture many terminating functional programs and proofs. But there has been further developments since the original work [108]. It would be useful to explore how to efficiently implement more sophisticated notions of termination beyond SCT.

Run-time contracts for other liveness properties As demonstrated in chapter 4, there are benefits in formulating a liveness property as a run-time checkable property: The run-time check may be more precise, and integrates well with the contract system, which provides a tool for gradual enforcement. Discovery of other instances of this idea, and possibly generalizing it, is therefore useful.

Incremental symbolic execution Analysis tools benefit greatly from incrementality, where a small change to the source code only triggers re-analyzing of a small part of it. A recent development in systematic derivation of static analysis from big-step operational semantics [25] is particularly amenable to incrementality, where an entry in the memo-table only needs re-analyzing if the expressions that it depends on change. Making symbolic execution incremental would enable interactive usage of the tool, shortening the feedback loop.

5.2 Conclusion

This dissertation explores symbolic execution for higher-order languages and its application in leveraging run-time checks as specifications, enabling a gradual

verification that is sound, precise, and modular. First, it shows that symbolic symbolic execution is feasible in a higher-order setting, and can find bugs and generate concrete counterexamples in a relatively complete way with respect to the underlying first-order SMT solver. Next, it shows that not only useful as a bug-finding tool, symbolic execution can also be finitized and used as a verification that is particularly effective against first-class, higher-order contracts, whose expressiveness and dynamicity prevent more conventional techniques such as type systems or direct translation to logics. Finally, it shows that even liveness properties such as termination and total correctness can also be expressed as dynamic safety checks, which integrates well with the existing contract system and can be verified by symbolic execution in the same way as every other safety property, inheriting all existing benefits. The verification framework obtained from dynamic checks and symbolic execution is effective in gradually and non-intrusively enriching an existing language with more static checks.

Appendix A: Proofs

A.1 Proof of Soundness and Relative Completeness of Counterexamples

A.1.1 Definitions

This section shows full definitions for metafunctions and relations omitted from the paper.

Figure A.1 shows metafunction $app?_X[\![E]\!]$ that determines whether expression E applies variable X or not. This definition is used in determining whether a function directly applies its argument. Next, figure A.2 shows the approximation relation between heaps. Next, figure A.3 shows the approximation between states of the form (E, Σ). Finally, figure A.4 shows the translation of a heap into a formula.

$$
\begin{array}{rcl}
app?_X[\![(X\ E)]\!] & = & \texttt{true} \\
app?_X[\![(\texttt{if}\ E\ E_1\ E_2)]\!] & = & app?_X[\![E]\!] \vee app?_X[\![E_1]\!] \vee app?_X[\![E_2]\!] \\
app?_X[\![(E_1\ E_2)]\!] & = & app?_X[\![E_1]\!] \vee app?_X[\![E_2]\!] \\
app?_X[\![(O\ E\ ...)]\!] & = & \vee\ (app?_X[\![E]\!])\ ... \\
app?_X[\![E]\!] & = & \texttt{false}
\end{array}
$$

Figure A.1: Checking for direct application

HEAP-EMPTY
$$\emptyset \sqsubseteq_{\vec{L}}^{\{\}} \emptyset$$

HEAP-EXT
$$\frac{\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma}{\Sigma'[L' \mapsto S'] \sqsubseteq_{\vec{L}}^{F} \Sigma}$$

HEAP-INT
$$\frac{\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma}{\Sigma'[L' \mapsto N] \sqsubseteq_{\vec{L}}^{F[L \mapsto L']} \Sigma[L \mapsto N]}$$

HEAP-LAM
$$\frac{\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma \qquad (E', \Sigma') \sqsubseteq_{\vec{L}}^{F} (E, \Sigma)}{\Sigma'[L' \mapsto (\lambda\ (X)\ E')] \sqsubseteq_{\vec{L}}^{F[L \mapsto L']} \Sigma[L \mapsto (\lambda\ (X)\ E)]}$$

HEAP-OPQ-1
$$\frac{\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma \qquad lab_{\Sigma'}[\![V']\!] \cap \vec{L} = \emptyset}{\Sigma'[L' \mapsto V'] \sqsubseteq_{\vec{L}}^{F[L \mapsto L']} \Sigma[L \mapsto \bullet^{T}]}$$

HEAP-OPQ-2
$$\frac{\Sigma'[L' \mapsto V'] \sqsubseteq_{\vec{L}}^{F} \Sigma[L \mapsto \bullet^{TP\ldots}] \qquad \Sigma' \vdash V' : F(P_1)\,\checkmark}{\Sigma'[L' \mapsto V'] \sqsubseteq_{\vec{L}}^{F} \Sigma[L \mapsto \bullet^{TP\ldots P_1}]}$$

HEAP-CASE-1
$$\frac{\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma \qquad lab_{\Sigma'}[\![E']\!] \cap \vec{L} = \emptyset}{\Sigma'[L' \mapsto (\lambda\ (X)\ E')] \sqsubseteq_{\vec{L}}^{F[L \mapsto L']} \Sigma[L \mapsto \mathtt{case}^{T}\ [\,]]}$$

HEAP-CASE-2
$$\frac{\begin{array}{c}\Sigma''[L' \mapsto (\lambda\ (X)\ E')] \sqsubseteq_{\vec{L}}^{F} \Sigma[L \mapsto \mathtt{case}^{T}\ [\ldots]] \\ F(L_x) = L'_x \qquad ([X/L'_x]E', \Sigma'') \longmapsto\!\!\!\!\rightarrow (V', \Sigma') \qquad (V', \Sigma') \sqsubseteq_{\vec{L}}^{F} (V, \Sigma)\end{array}}{\Sigma'[L' \mapsto (\lambda\ (X)\ E')] \sqsubseteq_{\vec{L}}^{F[L \mapsto L']} \Sigma[L \mapsto \mathtt{case}^{T}\ [\ldots L_x \mapsto V]]}$$

HEAP-CONST-FUNC
$$\frac{\begin{array}{c}\Sigma'' \sqsubseteq_{\vec{L}}^{F} \Sigma \\ F(L) = L' \qquad lab_{\Sigma''}[\![E']\!] \cap \vec{L} = \emptyset \qquad \Sigma''(L') = (\lambda\ (X)\ E') \qquad FV[\![E']\!] = \emptyset \\ lab_{\Sigma'}[\![E'']\!] \cap \vec{L} = \emptyset \qquad (E', \Sigma'') \longmapsto\!\!\!\!\rightarrow (V', \Sigma) \qquad (V', \Sigma') \sqsubseteq_{\vec{L}}^{F} (V, \Sigma)\end{array}}{\Sigma'[L'_a \mapsto V'] \sqsubseteq_{\vec{L}}^{F[L_a \mapsto L'_a]} \Sigma[L_a \mapsto V, L \mapsto (\lambda\ (X)\ L)_a]}$$

HEAP-CLO
$$\frac{\begin{array}{c}\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma \qquad F(L) = L' \\ \Sigma'(L') = (\lambda\ (X)\ E') \qquad FV[\![E']\!] = \{X\} \qquad \neg(app?_X[\![E']\!]) \qquad lab_{\Sigma'}[\![E']\!] = \emptyset\end{array}}{\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma[L \mapsto (\lambda\ (X)\ (\lambda\ (Y)\ ((\bullet\ X)\ Y)))]}$$

HEAP-HAVOC
$$\frac{\begin{array}{c}\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma \qquad F(L) = L' \\ \Sigma'(L') = (\lambda\ (X)\ E') \qquad FV[\![E']\!] = \{X\} \qquad app?_X[\![E']\!] \qquad lab_{\Sigma'}[\![E']\!] = \emptyset\end{array}}{\Sigma' \sqsubseteq_{\vec{L}}^{F} \Sigma[L \mapsto (\lambda\ (X)\ (\bullet\ (X\ \bullet)))]}$$

Figure A.2: Approximation between Heaps

$$\textsc{Loc}\ \frac{F(L) = L'}{(L', \Sigma') \sqsubseteq^F_{\vec{L}} (L, \Sigma)} \qquad \textsc{Err-Opq}\ \frac{L' \notin \vec{L}}{(\mathtt{err}^{L'}_O, \Sigma') \sqsubseteq^F_{\vec{L}} (E, \Sigma)} \qquad \textsc{Err}\ \frac{L' \in \vec{L}}{(\mathtt{err}^{L}_O, \Sigma') \sqsubseteq^F_{\vec{L}} (\mathtt{err}^{L}_O, \Sigma)}$$

$$\textsc{Opq-App}\ \frac{\begin{array}{c} E' \neq \mathrm{A} \qquad lab_{\Sigma'}[\![E']\!] \cap \vec{L} \subseteq lab_{\Sigma}[\![L_x]\!] \qquad ((L'_f\, L'_x), \Sigma'') \sqsubseteq^F_{\vec{L}} ((L_f\, L_x), \Sigma) \\ \Sigma''(L'_f) = (\lambda\ (X)\ E'') \qquad \Sigma(L_f) = \bullet^{T \to T'} \qquad ([X/L'_x]E'', \Sigma'') \longmapsto\!\!\!\!\to (E', \Sigma') \end{array}}{(E', \Sigma') \sqsubseteq^F_{\vec{L}} ((L_f\, L_x), \Sigma)}$$

$$\textsc{If}\ \frac{(E', \Sigma') \sqsubseteq^F_{\vec{L}} (E, \Sigma) \qquad (E'_1, \Sigma') \sqsubseteq^F_{\vec{L}} (E_1, \Sigma) \qquad (E'_2, \Sigma') \sqsubseteq^F_{\vec{L}} (E_2, \Sigma)}{((\mathsf{if}\ E'\ E'_1\ E'_2), \Sigma') \sqsubseteq^F_{\vec{L}} ((\mathsf{if}\ E\ E_1\ E_2), \Sigma)}$$

$$\textsc{App}\ \frac{(E'_1, \Sigma') \sqsubseteq^F_{\vec{L}} (E_1, \Sigma) \qquad (E'_2, \Sigma') \sqsubseteq^F_{\vec{L}} (E_2, \Sigma)}{((E'_1\, E'_2), \Sigma') \sqsubseteq^F_{\vec{L}} ((E_1\, E_2), \Sigma)}$$

$$\textsc{Prim}\ \frac{(E', \Sigma') \sqsubseteq^F_{\vec{L}} (E, \Sigma)\ \ ...}{((O\, E'\, ...), \Sigma') \sqsubseteq^F_{\vec{L}} ((O\, E\, ...), \Sigma)}$$

$$\textsc{Lam}\ \frac{(E', \Sigma') \sqsubseteq^F_{\vec{L}} (E, \Sigma)}{((\lambda\ (X : T)\ E'), \Sigma') \sqsubseteq^F_{\vec{L}} ((\lambda\ (X : T)\ E), \Sigma)} \qquad \textsc{Nat}\ (N, \Sigma') \sqsubseteq^F_{\vec{L}} (N, \Sigma)$$

Figure A.3: Approximation between Expressions

$$\{\!\{L \vec{\mapsto} S\}\!\} = \bigwedge \{\!\{L \vec{\mapsto} S\}\!\}$$
$$\{\!\{L \mapsto n\}\!\} = (L = n)$$
$$\{\!\{L \mapsto \bullet^{\texttt{nat}\ \vec{P}}\}\!\} = \bigwedge\ \{\!\{L \vec{:} P\}\!\}$$
$$\{\!\{L \mapsto \texttt{case}^T\ ...\,[L_1 \mapsto L_2]\ ...\,[L_3 \mapsto L_4]\ ...\}\!\} = \bigwedge\ (\{\!\{L_1 = L_3\}\!\}_{\texttt{nat}} \Rightarrow \{\!\{L_2 = L_4\}\!\}_T)\ ...$$
$$\{\!\{L : ((\lambda\ (X)\ (\texttt{zero?}\ X)))\}\!\} = (L = 0)$$
$$\{\!\{L : ((\lambda\ (X)\ (= X (+\ L_1\ L_2))))\}\!\} = (L = (L_1 + L_2))$$
$$\{\!\{L_1 = L_2\}\!\}_{\texttt{nat}} = (L_1 = L_2)$$
$$\{\!\{L_1 = L_2\}\!\}_{T \to T'} = \{\!\{\Sigma(L_1) = \Sigma(L_2)\}\!\}$$
$$\{\!\{(\texttt{case}^T\ [L_1 \mapsto L_2]\,...\,{=}\texttt{case}^T\ [L_3 \mapsto L_4]\,...)\}\!\} = (\{\!\{(L_1 = L_3) \Rightarrow (L_2 = L_4)\}\!\}\ ...)$$
$$\{\!\{(\lambda\ (X)\ L_1) = (\lambda\ (X)\ L_2)\}\!\} = \{\!\{L_1 = L_2\}\!\}$$
$$\{\!\{(\lambda\ (X)\ (L_1\,(X\,L_2))) = (\lambda\ (X)\ (L_3\,(X\,L_4)))\}\!\} = (\wedge\ \{\!\{L_1 = L_3\}\!\}\ \{\!\{L_2 = L_4\}\!\})$$
$$\{\!\{(\lambda\ (X)\ (\lambda\ (Y)\ ((L_1\,X)\,Y))) = (\lambda\ (X)\ (\lambda\ (Y)\ ((L_2\,X)\,Y)))\}\!\} = \{\!\{L_1 = L_2\}\!\}$$

Figure A.4: Translation of Heap

A.1.2 Theorems and Lemmas

This section presents the proofs for soundness and relative completeness of counterexamples as well as supporting lemmas.

Soundness of counterexamples results from the fact that the reduction leaves enough refinements to unambiguously steer all concretizations of a branch's heap to only follow that branch.

Relative completeness of counterexamples results from soundness of the reduction relation and completeness of the underlying solver for first-order data.

A.1.2.1 Soundness and Completeness of Counterexamples

Proof of theorem 1

Proof. *Follows from lemmas 4, 6, and 7.*

Proof of theorem 2

Proof. *Soundness of the reduction relation (lemma 5) ensures the error is discovered, and the completeness of the first-order solver ensures the heap can be instantiated.*

A.1.2.2 Lemmas on reduction relation

Lemma 4 (Completeness of Refinement in Reduction)**.** *Heap refinements are sufficient to steer all concrete executions through the same path.*

If $(E_1, \Sigma_1) \longmapsto (E_2, \Sigma_2)$ *and* $\Sigma' \sqsubseteq^F \Sigma_2$ *where* $\Sigma_2(L)$ *is structurally similar to* $\Sigma'(F(L))$ *for all* $L \in dom(F)$*, and* $(E_1', \Sigma') \sqsubseteq (E_1, \Sigma_1)$*, then* $(E_1', \Sigma') \longmapsto\!\!\!\!\!\rightarrow (E_2', \Sigma'')$ *such that* $(E_2', \Sigma'') \sqsubseteq (E_2, \Sigma_2)$*.*

Proof. *By case analysis of the reduction and approximation relations.*

- *Case* $(V, \Sigma_1) \longmapsto (L, \Sigma[L \mapsto V])$ *where* V *is fully concrete: then* $(V, \Sigma') \longmapsto (L', \Sigma'[L' \mapsto V])$*, which refines* $(L, \Sigma[L \mapsto V])$
- *Case* $(\bullet_L^T, \Sigma_1) \longmapsto (L, \Sigma[L \mapsto \bullet^T])$ *and* $(V', \Sigma') \sqsubseteq (\bullet_L^T, \Sigma_1)$*: then* $(V', \Sigma') \longmapsto (L', \Sigma'[L' \mapsto V'])$*, which refines* $(L, \Sigma[L \mapsto \bullet^T])$
- *Case* $(\,(\mathsf{if}\ L\ E_0\ E_1)\,, \Sigma_1) \longmapsto (E_i, \Sigma_2)$ *because* $\delta(\Sigma_1, \texttt{zero?}, L) \ni (i, \Sigma_2)$*: then by lemma 9,* $\delta(\Sigma', \texttt{zero?}, L') \ni (i, \Sigma'')$*. Thus* $(\,(\mathsf{if}\ L'\ E_0'\ E_1')\,, \Sigma') \longmapsto (E_i', \Sigma'')$*, which refines* (E_i, Σ_2)
- *Case* $(\,(O\ \vec{L})\,, \Sigma_1) \longmapsto (\mathrm{A}, \Sigma_2)$*: by lemma 9,* $(\,(O\ \vec{L'})\,, \Sigma') \longmapsto (\mathrm{A}', \Sigma'')$*, which refines* (A, Σ_2)

- *Case* $(\ (L_1\ L_2\ ), \Sigma_1) \longmapsto ([X/L_2]E, \Sigma_1)$, $\Sigma_1(L_1) = (\lambda\ (X\ )\ E\ )$ *and* $\Sigma'(L'_1) = (\lambda\ (X\ )\ E'\ )$*: then* $(\ (L'_1 L'_2\ ), \Sigma') \longmapsto ([X/L'_2]E', \Sigma')$*, which refines* $([X/L_2]E, \Sigma_1)$.

- *Case* $(\ (L_1 L_2\ ), \Sigma_1) \longmapsto (L_a, \Sigma_1[L_1 \mapsto \mathtt{case}^T\ [...\ L_2 \mapsto L_a\ ...]])$ *and* $(\ (L'_1 L'_2\ ), \Sigma') \sqsubseteq (\ (L_1\ L_2\ ), \Sigma_1)$ *and* $\Sigma'(L'_1) = \mathtt{case}^T\ [...\ L'_2 \mapsto L'_a\ ...]$*: then* $(\ (L'_1\ L'_2\ ), \Sigma') \longmapsto (L'_a, \Sigma')$*, which refines* $(L_a, \Sigma_1[L_1 \mapsto \mathtt{case}^T\ [...\ L_2 \mapsto L_a\ ...]])$.

- *Case* $(\ (L_1\ L_2\ ), \Sigma_1) \longmapsto (L_a, \Sigma_1[L_1 \mapsto (\lambda\ (X\ )\ L_a\ )])$ *and* $(\ (L'_1\ L'_2\ ), \Sigma') \sqsubseteq (\ (L_1 L_2\ ), \Sigma_1[L_1 \mapsto (\lambda\ (X\ )\ L_a\ )])$ *and* $\Sigma'(L_1) = (\lambda\ (X\ )\ L'_a\ )$*: then* $(\ (L'_1 L'_2\ ), \Sigma') \longmapsto (L'_a, \Sigma')$*, which refines* $(L_a, \Sigma_1[L_1 \mapsto (\lambda\ (X\ )\ L_a\ )])$

- *Case* $(\ (L_1\ L_2\ ), \Sigma_1) \longmapsto (\ (L_3\ (L_2\ L_4\ )), \Sigma_1[L_1 \mapsto (\lambda\ (X\ )\ (L_3\ (X\ L_4\ )))])$ *and* $(\ (L'_1\ L'_2\ ), \Sigma') \sqsubseteq (\ (L_1\ L_2\ ), \Sigma_1)$ *and* $\Sigma'(L'_1) = (\lambda\ (X\ )\ (L'_3\ (X\ L'_4\ )))$*: then* $(\ (L'_1\ L'_2\ ), \Sigma') \longmapsto (\ (L'_3\ (L'_2\ L'_4\ )), \Sigma')$*, which refines* $(\ (L_3\ (L_2\ L_4\ )), \Sigma_1[L_1 \mapsto (\lambda\ (X\ )\ (L_3\ (X\ L_4\ )))])$

- *Case* $(\ (L_1 L_2\ ), \Sigma_1) \longmapsto (\ (\lambda\ (Y\ )\ (\ (L_3\ L_2\ )\ Y\ )), \Sigma_1[L_1 \mapsto (\lambda\ (X\ )\ (\lambda\ (Y\ )\ (\ (L_3\ X\ )\ Y\ )))])$ *and* $(\ (L'_1\ L'_2\ ), \Sigma') \sqsubseteq (\ (L_1\ L_2\ ), \Sigma_1[L_1 \mapsto (\lambda\ (X\ )\ (\lambda\ (Y\ )\ (\ (L_3\ X\ )\ Y\ )))])$ *and* $\Sigma'(L'_1) = (\lambda\ (X\ )\ (\lambda\ (Y\ )\ (\ (L'_3\ X\ )\ Y\ )))$*: then* $(\ (L'_1\ L'\ )2\ ), \Sigma') \longmapsto (\lambda\ (Y\ )\ (\ (L'_3\ L'_2\ )\ Y\ )), \Sigma')$*, which refines* $(\ (L_1 L_2\ ), \Sigma_1[L_1 \mapsto (\lambda\ (X\ )\ (\lambda\ (Y\ )\ (\ (L_3\ X\ )\ Y\ )))$

Lemma 5 (Soundness of Reduction)**.**

If $(E'_1, \Sigma'_1) \longmapsto (E'_2, \Sigma'_2)$ *and* $(E'_1, \Sigma'_1) \sqsubseteq (E_1, \Sigma_1)$ *then* $(E_1, \Sigma_1) \longmapsto\!\!\!\!\!\rightarrow\ (E_2, \Sigma_2)$ *such that* $(E'_2, \Sigma'_2) \sqsubseteq (E_2, \Sigma_2)$.

Proof. *By case analysis of the reduction and approximation relations.*

- *Case* $(N, \Sigma_1') \longmapsto (L', \Sigma_1'[L' \mapsto N])$*:* *We have* $(N, \Sigma_1') \sqsubseteq^F (\bullet_L^{\texttt{nat}}, \Sigma_1)$*, and* $(\bullet_L^{\texttt{nat}}, \Sigma_1) \longmapsto (L, \Sigma[L \mapsto \bullet^{\texttt{nat}}])$ *where* $(L', \Sigma_1'[L' \mapsto N]) \sqsubseteq^{F[L \mapsto L']} (L, \Sigma[L \mapsto \bullet^{\texttt{nat}}])$

- *Case* $(\texttt{(if } L' \, E_0' \, E_1'\texttt{)}, \Sigma_1') \longmapsto (E_i', \Sigma_2')$ *because* $\delta(\Sigma_1', \texttt{zero?}, L') \ni (i, \Sigma_2')$*, and* $(\texttt{(if} L' E_0' E_1'\texttt{)}, \Sigma_1') \sqsubseteq (\texttt{(if} L E_0 E_1\texttt{)}, \Sigma_1)$*:* *By lemma 10, we have* $\delta(\Sigma_1, \texttt{zero?}, L') \ni (i, \Sigma_2)$*, thus* $(\texttt{(if } L \, E_0 \, E_1\texttt{)}, \Sigma_1) \longmapsto (E_i, \Sigma_2)$ *and* $(E_i', \Sigma_2') \sqsubseteq (E_i, \Sigma_2)$*.*

- *Case* $(\texttt{(}O \, \vec{L'}\texttt{)}, \Sigma_1') \longmapsto (\mathrm{A}', \Sigma_2')$ *and* $(\texttt{(}O \, \vec{L'}\texttt{)}, \Sigma_1') \sqsubseteq (\texttt{(}O \, \vec{L}\texttt{)}, \Sigma_1)$*:* *By lemma 10.*

- *Case* $(\texttt{(}L_1' \, L_2'\texttt{)}, \Sigma_1') \longmapsto ([X/L_2']E', \Sigma_1')$ *where* $\Sigma_1'(L_1') = \texttt{(}\lambda \ \texttt{(}X\texttt{)} \ E'\texttt{)}$ *and* $(\texttt{(}L_1' L_2'\texttt{)}, \Sigma_1') \sqsubseteq (\texttt{(}L_1 L_2\texttt{)}, \Sigma_1)$ *where* $\Sigma_1(L_1) = \texttt{(}\lambda \ \texttt{(}X\texttt{)} \ E\texttt{)}$*: then* $(\texttt{(}L_1 L_2\texttt{)}, \Sigma_1) \longmapsto ([X/L_2]E, \Sigma_1)$ *where* $([X/L_2']E', \Sigma_1') \sqsubseteq ([X/L_2]E, \Sigma_1)$

- *Case* $(E_1', \Sigma_1') \longmapsto (E_2', \Sigma_2')$ *where* $E_2 \neq \mathrm{A}$ *and* $(E_1', \Sigma_1') \sqsubseteq (\texttt{(}L_1 \, L_2\texttt{)}, \Sigma_1)$ *because* $\Sigma_2(L_1)$ *is an opaque function: then* $(\texttt{(}L_1 \, L_2\texttt{)}, \Sigma_1)$ *continues to approximate* (E_2, Σ_2') *by the same rule.*

- *Case* $(E_1', \Sigma_1') \longmapsto (\mathrm{A}', \Sigma_2')$ *where* $(E_1', \Sigma_1') \sqsubseteq (\texttt{(}L_1 \, L_2\texttt{)}, \Sigma_1)$ *because* $\Sigma_2(L_1) = \texttt{case}^T \ [...]$*:*

  - *SubCase* $\mathrm{A}' = \texttt{err}_O^L$*:* *By the approximation relation's premise,* $L \notin \vec{L}$*, so the error coming from the opaque program portion is approximated by any expression.*

  - *SubCase* $\mathrm{A}' = L'$*:* *The opaque application either returns a fresh opaque value or an existing value already known to approximate* L'*.*

- *Case* $(E_1', \Sigma_1') \longmapsto (\mathrm{A}', \Sigma_2')$ *where* $(E_1', \Sigma_1') \sqsubseteq (\,(L_1\, L_2\,), \Sigma_1)$ *because* $\Sigma_2(L_1) = (\lambda\ (X)\ L_a)$*:*

  - *SubCase* $\mathrm{A}' = \mathtt{err}_O^L$*: By the approximation relation's premise,* $L \notin \vec{L}$*, so the error coming from the opaque program portion is approximated by any expression.*

  - *SubCase* $\mathrm{A}' = L_a'$*: By the approximation relation's premise,* $(\,((\lambda\ (X)\ E')L_x'), \Sigma_0') \longmapsto (L_a', \Sigma_1')$ *where* $(\lambda\ (X)\ E')$ *is a constant function approximated by* L_a*.*

- *Case* $(E_1', \Sigma_1') \longmapsto (\mathrm{A}', \Sigma_2')$ *where* $(E_1', \Sigma_1') \sqsubseteq (\,(L_1\, L_2\,), \Sigma_1)$ *because* $\Sigma_2(L_1) = (\lambda\ (X)\ (L_3\ (X\, L_4)))$*:*

  - *SubCase* $\mathrm{A}' = \mathtt{err}_O^L$ *where* $L \notin \vec{L}$*: the error coming from the opaque program portion is approximated by any expression.*

  - *SubCase* $\mathrm{A}' = \mathtt{err}_O^L$ *where* $L \in \vec{L}$*: the context* $(L_3\ (\left[\ \right] L_4))$ *discovers the error (lemma 8).*

  - *SubCase* $\mathrm{A}' = L_a'$*: the opaque application returns a fresh opaque value.*

- *Case* $(E_1', \Sigma_1') \longmapsto (\mathrm{A}', \Sigma_2')$ *where* $(E_1', \Sigma_1') \sqsubseteq (\,(L_1\, L_2\,), \Sigma_1)$ *because* $\Sigma_2(L_1) = (\lambda\ (X)\ (\lambda\ (Y)\ ((L_3\, X)\, Y)))$*:*

  - *SubCase* $\mathrm{A}' = \mathtt{err}_O^L$*: the error source* L *must be from the opaque program portion because* $(\,((\lambda\ (X)\ E')\ L_x'), \Sigma_0') \longmapsto\!\!\!\!\rightarrow (E_1', \Sigma_1')$ *and* $(\lambda\ (X)\ E')$ *does not directly apply its argument in* E'*.*

  - *SubCase* $\mathrm{A}' = L_a'$*: the value is approximated by a freshly allocated function* $(\lambda\ (Y)\ ((L_3\, L_x')\, Y))$*.*

Lemma 6. *If a concrete heap instantiates an abstract heap, it instantiates any other heap that reduces to the abstract heap.*

If $\Sigma' \sqsubseteq \Sigma_2$ *and* $(E_1, \Sigma_1) \longmapsto (E_2, \Sigma_2)$ *then* $\Sigma' \sqsubseteq \Sigma_1$.

Proof. *By case analysis of the reduction* $(E_1, \Sigma_1) \longmapsto (E_2, \Sigma_2)$. *In each case, either* Σ_1 *is the same as* Σ_2, *is a restriction of* Σ_2, *or maps the same location in* Σ_2 *to a more approximate value.*

Lemma 7. *If a heap approximates a concrete heap, it approximates any heap that concrete heap reduces to.*

If $(E_1', \Sigma_1') \longmapsto (E_2', \Sigma_2')$ *and* $\Sigma_1' \sqsubseteq \Sigma$, *then* $\Sigma_2' \sqsubseteq \Sigma$.

Proof. *By case analysis of the reduction* $(E_1', \Sigma_1') \longmapsto (E_2', \Sigma_2')$. *In each case, either* Σ_2' *is the same as* Σ_1' *or extends* Σ_1'.

Lemma 8.

If $(\mathcal{E}[L_x'], \Sigma_1') \longmapsto\!\!\!\!\rightarrow (\mathtt{err}_O^L, \Sigma_2')$ *and* $(L_x', \Sigma_1') \sqsubseteq_{\vec{L}} (L_x, \Sigma_1)$ *then* $(\,(L_1\,(L_x L_2)), \Sigma_1) \longmapsto$ $\rightarrow (\mathtt{err}_O^L, \Sigma_2)$.

Proof. *Without loss of generality, any context triggering an error from the concrete program portion can be reduced to the minimal form:* $\mathcal{E} ::= [\ ] \mid (\mathcal{E} L)$, *which the context* $(\,(L_1\,([\ ]\, L_2))$ *produces an approximation of.*

A.1.2.3 Lemmas on Primitives

Lemma 9 (Completeness of Refinement in Primitives)**.**

If $\delta(\Sigma, O, L) \ni (V, \Sigma_1)$ *and* $(L', \Sigma') \sqsubseteq (L, \Sigma)$ *then* $\delta(\Sigma', O, L') \ni (V', \Sigma_1')$ *such that* $(V', \Sigma_1') \sqsubseteq (V, \Sigma_1)$.

Proof. *By inspection of δ and cases of O and the approximation relelation.*

Lemma 10 (Soundness of Primitives)**.**

If $(L', \Sigma_1') \sqsubseteq (L, \Sigma_1)$ *and* $\delta(\Sigma_1', O, L') \ni (\mathrm{A}', \Sigma_1')$*, then* $\delta(\Sigma_1, O, L) \ni (\mathrm{A}, \Sigma_2)$ *such that* $(\mathrm{A}', \Sigma_2') \sqsubseteq (\mathrm{A}, \Sigma_2)$*.*

Proof. *By case analysis of O.*

Lemma 11. *Primitives only refine the heap.*

If $\delta(\Sigma_1, O, L) \ni (\mathrm{A}, \Sigma_2)$ *and* $\Sigma' \sqsubseteq \Sigma_2$*, then* $\Sigma' \sqsubseteq \Sigma_1$*.*

Proof. *By case analysis of O.*

Lemma 12. *Primitives only refine the heap.*

If $\delta(\Sigma_1', O, L) \ni (\mathrm{A}, \Sigma_2')$ *and* $\Sigma_1' \sqsubseteq \Sigma$*, then* $\Sigma_2' \sqsubseteq \Sigma$*.*

Proof. *By case analysis of O.*

A.2 Verification Soundess Proof

A.2.1 Soundness

This appendix establishes the formal soundness of the symbolic execution semantics, justifying its use for program verification. The mechanized proof of soundness is available online at `https://github.com/philnguyen/soft-contract/tree/popl18-ae/mechanized`.

$$\frac{E \sqsubseteq_F E'}{(\lambda\ (X)\ E) \sqsubseteq_F (\lambda\ (X)\ E')} \qquad \frac{}{N \sqsubseteq_F N} \qquad \frac{}{O \sqsubseteq_F O} \qquad \frac{}{X \sqsubseteq_F X}$$

$$\frac{E_1 \sqsubseteq_F E_1' \qquad E_2 \sqsubseteq_F E_2' \qquad L \neq \mathsf{L}_\bullet}{(E_1\ E_2)^L \sqsubseteq_F (E_1'\ E_2')^L}$$

$$\frac{E_1 \sqsubseteq_F E_1' \qquad E_2 \sqsubseteq_F E_2' \qquad E_3 \sqsubseteq_F E_3'}{(\mathsf{if}\ E_1\ E_2\ E_3) \sqsubseteq_F (\mathsf{if}\ E_1'\ E_2'\ E_3')} \qquad \frac{E \sqsubseteq_F E'}{(\mathsf{set!}\ X\ E') \sqsubseteq_F (\mathsf{set!}\ X\ E')}$$

$$\frac{E_1 \sqsubseteq_F E_1' \qquad E_2 \sqsubseteq_F E_2'}{(E_1 \to (\lambda\ (X)\ E_2)) \sqsubseteq_F (E_1' \to (\lambda\ (X)\ E_2'))}$$

$$\frac{E_1 \sqsubseteq_F E_1' \qquad E_2 \sqsubseteq_F E_2' \qquad \mathsf{L}_\bullet \notin \{L', L''\}}{(\mathtt{mon}^L_{L'}\ E_1\ E_2) \sqsubseteq (\mathtt{mon}^L_{L'}\ E_1'\ E_2')} \qquad \frac{\mathit{free}(E^\bullet) \subseteq \{X\}}{(\lambda\ (X)\ E^\bullet) \sqsubseteq_F \bullet} \qquad \frac{}{N \sqsubseteq_F \bullet}$$

Figure A.5: Approximation between expressions

A.2.1.1 Approximation

The approximation relation ($\sqsubseteq$) between components is indexed by an *abstraction map* (F) from each address in the instantiated component to one in the approximating component.

Structural cases are straightforward. In non-structural cases where the right-hand side is $\bullet$, some components in the left-hand side are enforced by predicate $\mathsf{restricted}_\mathsf{F}(\cdot)$ that all controls ($E^\bullet$) are purely instantiated, and all addresses only reference either values instantiated by unknown code or those from the set of "leaked" values from the known code. The latter property is established by making F map all "unknown" addresses to the special address $_\bullet$, which holds $\bullet$ and the set of all "leaked" values from the transparent code.

$$\frac{}{N \sqsubseteq_F N} \qquad \frac{}{O \sqsubseteq_F O} \qquad \frac{E \sqsubseteq_F E' \qquad \mathrm{P} \sqsubseteq_F \mathrm{P}' \qquad \Phi \sqsubseteq_F \Phi'}{\mathsf{Clo}(X, E, \mathrm{P})\Phi \sqsubseteq_F \mathsf{Clo}(X, E', \mathrm{P}')\Phi'}$$

$$\frac{\mathit{free}(E^\bullet) \subseteq \{X\} \qquad \mathsf{restricted_F}(\mathrm{P})}{\mathsf{Clo}(X, E^\bullet, \mathrm{P})\Phi \sqsubseteq_F \bullet} \qquad \frac{V \sqsubseteq_F V' \qquad S \sqsubseteq S'}{(V, S) \sqsubseteq_F (V', S')} \qquad \frac{}{S \sqsubseteq \emptyset}$$

$$\frac{}{S \sqsubseteq S}$$

Figure A.6: Approximation between runtime values

A.2.1.2 Proof

Lemma 13 (Reduction preserves approximation)**.**

If $\varsigma_1 \sqsubseteq_F \varsigma_1'$ *and* $\varsigma_1 \longmapsto \varsigma_2$ *then there is some* ς_2' *and* F' *such that* $\varsigma_2 \sqsubseteq_{F'} \varsigma_2'$ *and* $\varsigma_1' \longmapsto\!\!\!\!\!\rightarrow \varsigma_2'$

Proof. *By case analysis on the derivation of* $\varsigma_1 \sqsubseteq_F \varsigma_1'$ *and* $\varsigma_1 \longmapsto \varsigma_2$*.*

- *Case* $\mathcal{E}[(E, \mathrm{P})] \sqsubseteq_F \mathcal{E}'[(E', \mathrm{P}')]$ *where the* `distr` *relation holds: Next states continue to approximate by rule [Distr].*

- *Case* $\mathcal{E}[(U, \mathrm{P})] \sqsubseteq_F \mathcal{E}'[(U', \mathrm{P}')]$*: Next states continue to approximate by rule [Lit]*

- *Case* $\mathcal{E}[(X, \mathrm{P})] \sqsubseteq_F \mathcal{E}'[(X, \mathrm{P}')]$*: Next states continue to approximate by rule [Var], and existing approximation between environments, store-caches, and stores.*

- *Case* $\mathcal{E}[(N, \mathrm{P})] \sqsubseteq_F \mathcal{E}'[(\bullet, \mathrm{P}')]$*: Next states step by rule [Lit]. The new states*

approximate because $N \sqsubseteq_F \bullet$.

- *Case* $\mathcal{E}[\,(\mathsf{set!}\ (X, \mathrm{P})\ W)\,] \sqsubseteq_F \mathcal{E}'[\,(\mathsf{set!}\ (X, \mathrm{P}')\ W')\,]$*: Next states continue to approximate by rule [Set].*

- *Case* $\mathcal{E}[\,(\mathsf{if}\, W\, C_1\, C_2)\,] \sqsubseteq_F \mathcal{E}'[\,(\mathsf{if}\, W'\, C_1'\, C_2')\,]$*: By soundness of relation* $\mathsf{feasible?}$ *and that* $W \sqsubseteq_F W'$*, RHS must at least reduce through the rule that applies to LHS (either [CondTrue] or [CondFalse]), which preserves approximation.*

- *Case* $\mathcal{E}[\,(W_1\, W_2)^L\,] \sqsubseteq_F \mathcal{E}'[\,(W_1'\, W_2')^L\,]$*:*

  - *If* W_1' *is concrete, both states must reduce through the same reduction rule and the next states preserve the approximation relation.*

  - *If* W_1' *is* $(\bullet, S)$*,* W_1 *must contain purely instantiated code by the definition of* $W_1 \sqsubseteq W_1'$*. By assumption,* W_1*'s environment* P *only has access to "leaked" values approximated by those at address* $\bullet$*. The execution of* $(W_1\, W_2)$ *now has access to* W_2 *in addition, which is soundly approximated by rule [AppOpq] extending* $\bullet$ *to containt the approximating value* W_2'*. (If* α *is the new address pointing to the value at* W_2*, the new abstraction map is* $F[\alpha \mapsto \bullet]$*). The opaque application with store extended at* $\bullet$ *continues to approximate the arbitrary state that* $(W_1\, W_2)$ *steps to.*

- *Case* $(\mathcal{E}[C], M, \Phi, \Sigma) \sqsubseteq_F (\mathcal{E}'[\,(\bullet\ [W])\,], M', \Phi', \Sigma')$ *because* $\mathsf{restricted_F}(C)$ *and* $\mathcal{E} \sqsubseteq_F \mathcal{E}'[\,(\bullet\ [\,])\,]$*:*

  Either the same non-structural approximation continues to hold between RHS and LHS's next state, or if LHS transfers control to transparent code by ap-

plying a function, the function must be approximated by one value in $\Sigma'(\bullet)$*, which [AppOpq] soundly approximates by non-deterministically applying one.*

Proof of theorem 3

Proof. *The proof proceeds by rule-induction on the derivation of* $(\longmapsto\!\!\!\!\!\rightarrow)$ *in the concrete error trace. The base case (reflexive) vacuously holds. The inductive case (transitive) holds by lemma 13, where for each single reduction step* $(\longmapsto)$ *on the concrete state, the abstract state continues to approximate the concrete state in zero or more steps.*

A.3 Proofs of Soundness and SCT-Completeness for Termination Contract

Proof of Theorem 4

Proof. *Consider an infinite sequence of function calls. By Lemma 14 below, there's a closure that keeps being called. The sequence of arguments to this closure cannot satisfy the size-change property an infinite number of times. The diverging program that results in this call sequence will be killed.*

Lemma 14 (Recurring closure)**.** *Along any infinite sequence of function calls, there is at least one closure that is called infinitely often.*

Proof. *Consider the sequence of closures*

$$\mathsf{Clo}(X_1, E_1, \mathrm{P}_1), \ldots, \mathsf{Clo}(X_i, E_i, \mathrm{P}_i), \ldots$$

along the infinite call sequence.

- *Case 1: These come from a finite set. At least one must repeat infinitely often.*
- *Case 2: There are fresh closures that keep being generated dynamically. Because new infinite closures must be generated through finite λ forms, there must be some infinite subset of closures* $\mathsf{Clo}(X_i, E_i, \_)$ *generated by one same form* $(\lambda\ (X_i)\ E_i)$*. Let* $\mathsf{Clo}(X_m, E_m, \_)$ *be the set of closures whose body* E_m *contains this form* $(\lambda\ (X_i)\ E_i)$*.*
  - *Claim: There must be some closure* $\mathsf{Clo}(X_j, E_j, \mathrm{P}_j)$ *that keeps being called infinitely often.*
  - *Proof: By induction on the lexical depth of the term* $(\lambda\ (X_m)\ E_m)$*.*
    - *Subcase 1:* $(\lambda\ (X_m)\ E_m)$ *has lexical depth 0 (i.e. it is a top-level λ). Because it is not enclosed by any λ, the closure* $\mathsf{Clo}(X_m, E_m, \{\})$ *is created only once. By assumption,* $\mathsf{Clo}(X_m, E_m, \{\})$ *is called infinitely often to dynamically create the infinite closure set* $\mathsf{Clo}(X_i, E_i, \_)$*.*
    - *Subcase 2:* $(\lambda\ (X_m)\ E_m)$ *is directly enclosed by* $(\lambda\ (X_n)\ E_n)$*.*
      - *Subsubcase 2a: The set* $\mathsf{Clo}(X_m, E_m, \_)$ *is finite: at least one of them is called infinitely often to generate the infinite closure set* $\mathsf{Clo}(X_i, E_i, \_)$*.*
      - *Subsubcase 2b: The set* $\mathsf{Clo}(X_m, E_m, \_)$ *is infinite: apply the induction hypothesis on* $(\lambda\ (X_n)\ E_n)$ *(where new i is m and new m is n).*

$$\frac{}{\{\} \sqsubseteq_F \{\}} \qquad \frac{\mathrm{P} \sqsubseteq_F \mathrm{P}'}{\mathrm{P}[X \mapsto \alpha] \sqsubseteq_F \mathrm{P}'[X \mapsto F(\alpha)]} \qquad \frac{M \sqsubseteq_F M' \qquad W \sqsubseteq_F W'}{M[X \mapsto W] \sqsubseteq_F M'[X \mapsto W']}$$

$$\frac{C_1 \sqsubseteq_F C_1' \qquad C_2 \sqsubseteq_F C_2'}{\mathcal{E}[(\mathsf{if}\ [\,]\ C_1\ C_2)] \sqsubseteq_F \mathcal{E}'[(\mathsf{if}\ [\,]\ C_1'\ C_2')]} \qquad \frac{C \sqsubseteq_F C' \qquad \mathcal{E} \sqsubseteq_F \mathcal{E}' \qquad L \neq \mathsf{L}_\bullet}{\mathcal{E}[([\,]\ C)^L] \sqsubseteq_F \mathcal{E}'[([\,]\ C')^L]}$$

$$\frac{W \sqsubseteq_F W' \qquad \mathcal{E} \sqsubseteq_F \mathcal{E}' \qquad L \neq \mathsf{L}_\bullet}{\mathcal{E}[(W\ [\,])^L] \sqsubseteq_F \mathcal{E}'[(W'\ [\,])^L]}$$

$$\frac{\mathcal{E} \sqsubseteq_F \mathcal{E}' \qquad \mathrm{P} \sqsubseteq_F \mathrm{P}'}{\mathcal{E}[(\mathsf{set!}\ (X, \mathrm{P})\ [\,])] \sqsubseteq_F \mathcal{E}'[(\mathsf{set!}\ (X, \mathrm{P}')\ [\,])]}$$

$$\frac{\mathcal{E} \sqsubseteq_F \mathcal{E}' \qquad M \sqsubseteq_F M'}{\mathcal{E}[(\mathsf{rt}_X^{\vec{Y}}\ S\ M\ \Phi\ [\,])] \sqsubseteq_F \mathcal{E}'[(\mathsf{rt}_X^{\vec{Y}}\ S\ M'\ \Phi\ [\,])]}$$

$$\frac{\mathcal{E} \sqsubseteq_F \mathcal{E}' \qquad C \sqsubseteq_F \mathcal{E}' \qquad \mathsf{L}_\bullet \notin \{L, L'\}}{\mathcal{E}[(\mathtt{mon}_{L'}^{L}\ [\,]\ C)] \sqsubseteq_F \mathcal{E}'[(\mathtt{mon}_{L'}^{L}\ [\,]\ C')]}$$

$$\frac{\mathcal{E} \sqsubseteq_F \mathcal{E}' \qquad W \sqsubseteq_F W' \qquad \mathsf{L}_\bullet \notin \{L, L'\}}{\mathcal{E}[(\mathtt{mon}_{L'}^{L}\ W\ [\,])] \sqsubseteq_F \mathcal{E}'[(\mathtt{mon}_{L'}^{L}\ W'\ [\,])]} \qquad \frac{\mathcal{E} \sqsubseteq_F \mathcal{E}'}{\mathcal{E} \sqsubseteq_F \mathcal{E}[(\bullet\ [\,])]}$$

$$\frac{\mathsf{restricted}_\mathsf{F}(C) \qquad \mathcal{E} \sqsubseteq_F \mathcal{E}'[(\bullet\ W')^L]}{\mathcal{E}[([\,]\ C)^{\mathsf{L}_\bullet}] \sqsubseteq_F \mathcal{E}'[(\bullet\ W')^L]}$$

$$\frac{\mathsf{restricted}_\mathsf{F}(W) \qquad \mathcal{E} \sqsubseteq_F \mathcal{E}'[(\bullet\ W')^L]}{\mathcal{E}[(W\ [\,])^{\mathsf{L}_\bullet}] \sqsubseteq_F \mathcal{E}'[(\bullet\ W')^L]}$$

$$\frac{\mathsf{restricted}_\mathsf{F}(C_1) \qquad \mathsf{restricted}_\mathsf{F}(C_2) \qquad \mathcal{E} \sqsubseteq_F \mathcal{E}'[(\bullet\ W')^L]}{\mathcal{E}[(\mathsf{if}\ [\,]\ C_1\ C_2)] \sqsubseteq_F \mathcal{E}'[(\bullet\ W')^L]}$$

$$\frac{\mathsf{restricted}_\mathsf{F}(\mathrm{P}) \qquad \mathcal{E} \sqsubseteq_F \mathcal{E}'[(\bullet\ W')^L]}{\mathcal{E}[(\mathsf{set!}\ (X, \mathrm{P})\ [\,])] \sqsubseteq_F \mathcal{E}'[(\bullet\ W')^L]}$$

$$\frac{\Sigma \sqsubseteq_F \Sigma' \qquad V \sqsubseteq_F V'}{\Sigma \sqcup [\alpha \mapsto V] \sqsubseteq_F \Sigma' \sqcup [F(\alpha) \mapsto V']}$$

$$\frac{C \sqsubseteq_F C' \qquad M \sqsubseteq_F M' \qquad \Phi \sqsubseteq_F \Phi' \qquad \Sigma \sqsubseteq_F \Sigma}{(C, M, \Phi, \Sigma) \sqsubseteq_F (C', M', \Phi', \Sigma')}$$

$$\frac{\mathsf{restricted}_\mathsf{F}(C) \qquad \mathcal{E} \sqsubseteq_F \mathcal{E}'[(\bullet\ [\,])] \qquad M \sqsubseteq_F M' \qquad \Phi \sqsubseteq_F \Phi' \qquad \Sigma \sqsubseteq_F \Sigma'}{(\mathcal{E}[C], M, \Phi, \Sigma) \sqsubseteq_F (\mathcal{E}'[(\bullet\ W)], M', \Phi', \Sigma')}$$

Figure A.7: Approximation between machine components

$$\frac{}{\mathsf{restricted_F}(N)} \qquad \frac{}{\mathsf{restricted_F}(O)} \qquad \frac{\mathsf{restricted_F}(\mathrm{P})}{\mathsf{restricted_F}(\mathsf{Clo}(X, E^{\bullet}, \mathrm{P})\Phi)}$$

$$\frac{}{\mathsf{restricted_F}(\{\})} \qquad \frac{\mathsf{restricted_F}(\mathrm{P}) \qquad F(\alpha) = \bullet}{\mathsf{restricted_F}(\mathrm{P}[X \mapsto \alpha])} \qquad \frac{\mathsf{restricted_F}(V)}{\mathsf{restricted_F}((V, S))}$$

$$\frac{\mathsf{restricted_F}(\mathrm{P})}{\mathsf{restricted_F}((E^{\bullet}, \mathrm{P}))} \qquad \frac{\mathsf{restricted_F}(C_1) \qquad \mathsf{restricted_F}(C_2)}{\mathsf{restricted_F}((C_1\, C_2)^{\mathsf{L}\bullet})}$$

$$\frac{\mathsf{restricted_F}(C_1) \qquad \mathsf{restricted_F}(C_2) \qquad \mathsf{restricted_F}(C_3)}{\mathsf{restricted_F}((\mathsf{if}\ C_1\, C_2\, C_3))}$$

$$\frac{\mathsf{restricted_F}(C) \qquad \mathsf{restricted_F}(\mathrm{P})}{\mathsf{restricted_F}((\mathsf{set!}\ (X, \mathrm{P})\ C))}$$

Figure A.8: Restriction on runtime components instantiated by unknown code

$$\textsc{Prim}\ \frac{}{\mathrm{P} \vdash O \Downarrow O} \qquad \textsc{Base}\ \frac{}{\mathrm{P} \vdash N \Downarrow N} \qquad \textsc{Lam}\ \frac{}{\mathrm{P} \vdash (\lambda\ (\overrightarrow{X})\ E) \Downarrow \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})}$$

$$\textsc{Var}\ \frac{}{\mathrm{P} \vdash X \Downarrow \mathrm{P}(X)} \qquad \textsc{App-Clo}\ \frac{\mathrm{P} \vdash E \Downarrow \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}') \qquad \mathrm{P} \vdash \overrightarrow{E_x} \Downarrow \overrightarrow{V_x} \qquad \mathrm{P}'[\overrightarrow{X \mapsto V_x}] \vdash E' \Downarrow \mathrm{A}}{\mathrm{P} \vdash (E\ \overrightarrow{E_x}) \Downarrow \mathrm{A}}$$

$$\textsc{SC-Err}\ \frac{}{\mathrm{P}, \bot \vdash E \blacklozenge \mathsf{error}^{\mathsf{SC}}} \qquad \textsc{SC-Prim}\ \frac{}{\mathrm{P}, M \vdash O \blacklozenge O} \qquad \textsc{SC-Base}\ \frac{}{\mathrm{P}, M \vdash N \blacklozenge N}$$

$$\textsc{SC-Lam}\ \frac{}{\mathrm{P}, M \vdash (\lambda\ (\overrightarrow{X})\ E) \blacklozenge \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})} \qquad \textsc{SC-Var}\ \frac{}{\mathrm{P}, M \vdash X \blacklozenge \mathrm{P}(X)}$$

$$\textsc{SC-If-T}\ \frac{\mathrm{P}, M \vdash E \blacklozenge 0 \qquad \mathrm{P}, M \vdash E_1 \blacklozenge \mathrm{A}}{\mathrm{P}, M \vdash (\mathsf{if0}\ E\ E_1\ E_2) \blacklozenge \mathrm{A}}$$

$$\textsc{SC-If-F}\ \frac{\mathrm{P}, M \vdash E \blacklozenge V \text{ where } V\ 0 \qquad \mathrm{P}, M \vdash E_2 \blacklozenge \mathrm{A}}{\mathrm{P}, M \vdash (\mathsf{if0}\ E\ E_1\ E_2) \blacklozenge \mathrm{A}}$$

$$\textsc{SC-App-Clo}\ \frac{\mathrm{P}, M \vdash E \blacklozenge \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}') \qquad \mathrm{P}, M \vdash \overrightarrow{E_x} \blacklozenge \overrightarrow{V_x} \qquad \mathrm{P}'[\overrightarrow{X \mapsto V_x}], \mathit{update}(M, \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \overrightarrow{V_x}) \vdash E' \blacklozenge A}{\mathrm{P}, M \vdash (E\ \overrightarrow{E_x}) \blacklozenge A}$$

Figure A.9: Standard and Terminating semantics of λ_{SCT}.

$$
\begin{array}{rl}
\text{[Expressions]} & E ::= O \mid N \mid (\lambda\ (\overrightarrow{X})\ E) \mid X \\
& \quad \mid (E\ \overrightarrow{E}) \mid (\mathsf{term/c}\ E) \\
\text{[Value Literals]} & N ::= 0 \mid -1 \mid 1 \mid \ldots \\
\text{[Primitives]} & O ::= + \mid \mathsf{cons} \mid \mathsf{car} \mid \mathsf{cdr} \mid \ldots \\
\text{[Values]} & V ::= O \mid N \mid (V, V) \mid \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P}) \\
& \quad \mid \mathsf{term/c}(\mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})) \\
\text{[Size-change Table]} & M \in V \rightharpoonup \overrightarrow{V} \times G \\
\text{[Size-change Graph]} & G \in \mathcal{P}(\mathbb{N} \times R \times \mathbb{N}) \\
\text{[Change]} & R ::= \downarrow \mid \overline{\downarrow}
\end{array}
$$

Figure A.10: Syntax of λ_{CSCT}.

$$\textsc{Prim}\ \frac{}{\mathrm{P} \vdash O \Downarrow O} \qquad \textsc{Base}\ \frac{}{\mathrm{P} \vdash N \Downarrow N} \qquad \textsc{Lam}\ \frac{}{\mathrm{P} \vdash (\lambda\ (\overrightarrow{X})\ E) \Downarrow \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})}$$

$$\textsc{Var}\ \frac{}{\mathrm{P} \vdash X \Downarrow \mathrm{P}(X)} \qquad \textsc{Wrap-Lam}\ \frac{\mathrm{P} \vdash E \Downarrow \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})}{\mathrm{P} \vdash (\mathsf{term/c}\ E) \Downarrow \mathsf{term/c}(\mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P}))}$$

$$\textsc{Wrap-Prim}\ \frac{\mathrm{P} \vdash E \Downarrow O}{\mathrm{P} \vdash (\mathsf{term/c}\ E) \Downarrow O} \qquad \textsc{App-Clo}\ \frac{\mathrm{P} \vdash E \Downarrow \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}') \qquad \mathrm{P} \vdash \overrightarrow{E_x} \Downarrow \overrightarrow{V_x} \qquad \mathrm{P}'[\overrightarrow{X \mapsto V_x}] \vdash E' \Downarrow \mathrm{A}}{\mathrm{P} \vdash (E\ \overrightarrow{E_x}) \Downarrow \mathrm{A}}$$

$$\textsc{App-Term}\ \frac{\mathrm{P} \vdash E \Downarrow \mathsf{term/c}(\mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}')) \qquad \mathrm{P} \vdash \overrightarrow{\mathrm{P}_x} \Downarrow \overrightarrow{V_x} \qquad \mathrm{P}'[\overrightarrow{X \mapsto V_x}], \mathit{update}(\{\}, \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \overrightarrow{V_x}) \vdash E' \blacktriangleright \mathrm{A}}{\mathrm{P} \vdash (E\ \overrightarrow{E_x}) \Downarrow \mathrm{A}}$$

$$\textsc{SC-Err}\ \frac{}{\mathrm{P}, \bot \vdash E \blacktriangleright \mathsf{error}^{\mathsf{SC}}} \qquad \textsc{SC-Prim}\ \frac{}{\mathrm{P}, M \vdash O \blacktriangleright O} \qquad \textsc{SC-Base}\ \frac{}{\mathrm{P}, M \vdash N \blacktriangleright N}$$

$$\textsc{SC-Lam}\ \frac{}{\mathrm{P}, M \vdash (\lambda\ (\overrightarrow{X})\ E) \blacktriangleright \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})} \qquad \textsc{SC-Var}\ \frac{}{\mathrm{P}, M \vdash X \blacktriangleright \mathrm{P}(X)}$$

$$\textsc{SC-Wrap-Lam}\ \frac{\mathrm{P}, M \vdash E \blacktriangleright \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P})}{\mathrm{P}, M \vdash (\mathsf{term/c}\ E) \blacktriangleright \mathsf{term/c}(\mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P}))} \qquad \textsc{SC-Wrap-Prim}\ \frac{\mathrm{P}, M \vdash E \blacktriangleright O}{\mathrm{P}, M \vdash (\mathsf{term/c}\ E) \blacktriangleright O}$$

$$\textsc{SC-App-Clo}\ \frac{\mathrm{P}, M \vdash E \blacktriangleright \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}') \qquad \mathrm{P}, M \vdash \overrightarrow{E_x} \blacktriangleright \overrightarrow{V_x} \qquad \mathrm{P}'[\overrightarrow{X \mapsto V_x}], \mathit{update}(M, \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \overrightarrow{V_x}) \vdash E' \blacktriangleright \mathrm{A}}{\mathrm{P}, M \vdash (E\ \overrightarrow{E_x}) \blacktriangleright \mathrm{A}}$$

$$\textsc{SC-App-Term}\ \frac{\mathrm{P}, M \vdash E \blacktriangleright \mathsf{term/c}(\mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}')) \qquad \mathrm{P}, M \vdash \overrightarrow{E_x} \blacktriangleright \overrightarrow{V_x} \qquad \mathrm{P}'[\overrightarrow{X \mapsto V_x}], \mathit{update}(M, \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \overrightarrow{V_x}) \vdash E' \blacktriangleright \mathrm{A}}{\mathrm{P}, M \vdash (E\ \overrightarrow{E_x}) \blacktriangleright \mathrm{A}}$$

Figure A.11: Semantics of λ_{CSCT}.

$$\textsc{CC-Prim}\quad \frac{}{\mathrm{P}, M \vdash O \Downarrow O, \{M\}} \qquad \textsc{CC-Base}\quad \frac{}{\mathrm{P}, M \vdash N \Downarrow N, \{M\}}$$

$$\textsc{CC-Lam}\quad \frac{}{\mathrm{P}, M \vdash (\lambda\ (\overrightarrow{X})\ E) \Downarrow \mathsf{Clo}(\overrightarrow{X}, E, \mathrm{P}), \{M\}}$$

$$\textsc{CC-App}\quad \frac{\mathrm{P}, M \vdash E \Downarrow \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \{M' \ldots\} \qquad \mathrm{P}, M \vdash \overrightarrow{E_x} \Downarrow \overrightarrow{V_x}, \overrightarrow{\{M_x \ldots\}} \qquad \mathrm{P}'[\overrightarrow{X \mapsto V_x}], \mathit{ext}(M, \mathsf{Clo}(\overrightarrow{X}, E', \mathrm{P}'), \overrightarrow{V_x}) \vdash E' \Downarrow A, \{M'' \ldots\}}{\mathrm{P}, M \vdash (E\ \overrightarrow{E_x}) \Downarrow A, \{M' \ldots\} \cup \overrightarrow{\{M_x \ldots\}} \cup \{M'' \ldots\}}$$

Figure A.12: Call-sequence Semantics of λ_{CSCT}.

Bibliography

[1] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *ICFP '10: International Conference on Functional Programming*. ACM, September 2010.

[2] Aseem Rastogi, Nikhil Swamy, Cédric Fournet, Gavin Bierman, and Panagiotis Vekris. Safe & efficient gradual typing for typescript. In *ACM SIGPLAN Notices*, volume 50, pages 167–180. ACM, 2015.

[3] Ambrose Bonnaire-Sergeant, Rowan Davies, and Sam Tobin-Hochstadt. Practical optional types for clojure. In *European Symposium on Programming*, pages 68–94. Springer, 2016.

[4] Tim Freeman and Frank Pfenning. Refinement types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. ACM, June 1991.

[5] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008.

[6] Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon P. Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM, August 2014.

[7] Cormac Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, volume 41. ACM, January 2006.

[8] Philippe Meunier, Robert B. Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 2006.

[9] Sam Tobin-Hochstadt and David Van Horn. Higher-order symbolic execution via contracts. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2012.

[10] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: Automatically generating inputs of death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*. ACM, 2006.

[11] Patrice Godefroid, Nils Klarlund, and Koushik Sen. DART: Directed automated random testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2005.

[12] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, May 2002.

[13] Yichen Xie and Alex Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 2005.

[14] Junfeng Yang, Paul Twohey, Dawson Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *Sixth Symposium on Operating Systems Design and Implementation*. USENIX, 2004.

[15] Dana S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121(1-2), December 1993.

[16] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, June 2011.

[17] Tachio Terauchi. Dependent types from counterexamples. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2010.

[18] Phúc C. Nguyễn, Sam Tobin-Hochstadt, and David Van Horn. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2014.

[19] Koen Claessen and John Hughes. QuickCheck: A lightweight tool for random testing of haskell programs. In *Proceedings of the Fifth ACM SIGPLAN International Conference on Functional Programming*. ACM, 2000.

[20] Dana N. Xu, Simon Peyton Jones, and Simon Claessen. Static contract checking for Haskell. In *POPL '09: Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2009.

[21] Dana N. Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. ACM, 2012.

[22] Dimitrios Vytiniotis, Simon Peyton Jones, Koen Claessen, and Dan Rosén. Halo: Haskell to logic through denotational semantics. In *Proceedings of the 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2013.

[23] Kenneth Knowles and Cormac Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 32(2), February 2010.

[24] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2010.

[25] David Darais, Nicholas Labich, Phúc C. Nguyễn, and David Van Horn. Abstracting definitional interpreters (functional pearl). *Proc. ACM Program. Lang.*, 1(ICFP):12:1–12:25, August 2017.

[26] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Trans. Program. Lang. Syst.*, 19(1), January 1997.

[27] Chin Soon Lee, Neil D. Jones, and Amir M. Ben-Amram. The size-change principle for program termination. In *Proceedings of the 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '01, pages 81–92, New York, NY, USA, 2001. ACM.

[28] Damien Sereni and Neil D. Jones. Termination analysis of Higher-Order functional programs. In *Programming Languages and Systems, Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005, Proceedings*, volume 3780. Springer, 2005.

[29] Alexander Krauss. Certified size-change termination. In *International Conference on Automated Deduction*, pages 460–475. Springer, 2007.

[30] Panagiotis Manolios and Daron Vroon. Termination analysis with calling context graphs. In *International Conference on Computer Aided Verification*, pages 401–414. Springer, 2006.

[31] Nikhil Swamy, Catalin Hritcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoué, and Santiago Zanella-Béguelin. Dependent types and multi-monadic effects in F*. In *43rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 256–270. ACM, January 2016.

[32] Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G Scott, Ryan R Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with smt. *Proceedings of the ACM on Programming Languages*, 2(POPL):53, 2017.

[33] Ming Kawaguchi, PatrickM Rondon, and Ranjit Jhala. Dsolve: Safety verification via liquid types. In *Computer Aided Verification*. Springer Berlin Heidelberg, 2010.

[34] Casey Klein, Matthew Flatt, and Robert B. Findler. Random testing for higher-order, stateful programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, 2010.

[35] Naoki Kobayashi. Model checking Higher-Order programs. *J. ACM*, 60(3), June 2013.

[36] He Zhu and Suresh Jagannathan. Compositional and lightweight dependent type inference for ML. In *Conference on Verification, Model-Checking and Abstract Interpretation*, 2013.

[37] Stephen A. Cook. Soundness and completeness of an axiom system for program verification. In *SIAM Journal of Computing*, 1978.

[38] Robert B. Findler and Matthias Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*. ACM, September 2002.

[39] Christos Dimoulas, Sam Tobin-Hochstadt, and Matthias Felleisen. Complete monitors for behavioral contracts. In *21st European Symposium on Programming*, volume 7211. Springer-Verlag, 2012.

[40] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer-Verlag, 2008.

[41] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010.

[42] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Proceedings of the 8th international conference on Functional and Logic Programming*, volume 3945. Springer, January 2006.

[43] Michael Greenberg, Benjamin C. Pierce, and Stephanie Weirich. Contracts made manifest. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2010.

[44] Christos Dimoulas, Robert B. Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: No more scapegoating. In *Proceedings of the 38th Annual ACM Symposium on Principles of Programming Languages*. ACM, January 2011.

[45] John Hughes. Personal communication, 2015.

[46] Vincent St-Amour, Sam Tobin-Hochstadt, Matthew Flatt, and Matthias Felleisen. Typing the numeric tower. In *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 2012.

[47] 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*. USENIX Association, 2008.

[48] Phillip Heidegger and Peter Thiemann. Contract-Driven testing of JavaScript code. In *Objects, Models, Components, Patterns*. Springer Berlin Heidelberg, 2010.

[49] EricL Seidel, Niki Vazou, and Ranjit Jhala. Type targeted testing. In *Programming Languages and Systems*. Springer Berlin Heidelberg, 2015.

[50] Bertrand Meyer. *Eiffel : The Language*. Prentice Hall, October 1991.

[51] Tim Disney. contracts.coffee, July 2013.

[52] R. Plosch. Design by contract for Python. In *Proceedings of the Joint Asia Pacific Software Engineering Conference*. IEEE, December 1997.

[53] Thomas H. Austin, Tim Disney, and Cormac Flanagan. Virtual values for language extension. In *Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, October 2011.

[54] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert B. Findler, and Matthew Flatt. Chaperones and impersonators: Run-time support for reasonable interposition. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, October 2012.

[55] Rich Hickey, Michael Fogus, and contributors. core.contracts, July 2013.

[56] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 1977.

[57] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd International Symposium on Programming*, April 1976.

[58] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007.

[59] R. Majumdar and K. Sen. Hybrid concolic testing. In *Software Engineering, 2007. ICSE 2007. 29th International Conference on*. IEEE, 2007.

[60] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[61] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for c. *SIGSOFT Softw. Eng. Notes*, 30(5), 2005.

[62] Matthew Might. Abstract interpreters for free. In *International Symposium on Static Analysis*, SAS, pages 407–421. Springer, 2011.

[63] Thomas Gilray, Michael D. Adams, and Matthew Might. Allocation characterizes polyvariance: A unified methodology for polyvariant control-flow analysis. In *Proceedings of the 21st ACM SIGPLAN International Conference on Functional Programming*, ICFP, pages 407–420. ACM, 2016.

[64] Clark Barrett, ChristopherL Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In *Computer Aided Verification*. Springer, 2011.

[65] Matthew Might and Panagiotis Manolios. A posteriori soundness for non-deterministic abstract interpretations. In *Proceedings of the 10th International Conference on Verification, Model Checking, and Abstract Interpretation*. Springer-Verlag, 2009.

[66] Thomas Gilray, Steven Lyde, Michael D. Adams, Matthew Might, and David Van Horn. Pushdown control-flow analysis for free. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 691–704. ACM, 2016.

[67] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[68] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the 11th ACM SIGPLAN International Conference on Functional Programming*, September 2006.

[69] Manuel Fähndrich and Francesco Logozzo. Static contract checking with abstract interpretation. In *Proceedings of the 2010 International Conference on Formal Verification of Object-Oriented Software*. Springer, 2011.

[70] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The spec# experience. *Communications of the ACM*, 54(6):81–91, June 2011.

[71] P. Müller and J. N. Ruskiewicz. Using debuggers to understand failed verification attempts. In *Formal Methods*, FM, pages 73–87. Springer, 2011.

[72] Niki Vazou, PatrickM Rondon, and Ranjit Jhala. Abstract refinement types. In *European Symposium on Programming*. Springer Berlin Heidelberg, 2013.

[73] Jessica Gronski and Cormac Flanagan. Unifying hybrid types and contracts. In *Trends in Functional Programming*, pages 54–70. Citeseer, 2007.

[74] Taro Sekiyama and Atsushi Igarashi. Stateful manifest contracts. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*, POPL, pages 530–544. ACM, 2017.

[75] Taro Sekiyama, Yuki Nishida, and Atsushi Igarashi. Manifest contracts for datatypes. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL, pages 195–207. ACM, 2015.

[76] Ravi Chugh, David Herman, and Ranjit Jhala. Dependent types for JavaScript. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*. ACM, October 2012.

[77] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements: A logic for duck typing. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 2012.

[78] C. H. Luke Ong. On Model-Checking trees generated by Higher-Order recursion schemes. In *21st Annual IEEE Symposium on Logic in Computer Science (LICS'06)*. IEEE, 2006.

[79] 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*. ACM, January 2009.

[80] Naoki Kobayashi, Naoshi Tabuchi, and Hiroshi Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. In *Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 2010.

[81] Robin P Neatherway, Steven J Ramsay, and Chih-Hao Luke Ong. A traversal-based algorithm for higher-order model checking. In *Proceedings of the 17th*

Annual International Conference on Functional Programming, ICFP, pages 353–364. ACM, 2012.

[82] Takeshi Tsukada and Naoki Kobayashi. Untyped recursion schemes and infinite intersection types. In *Proceedings of the 13th International Conference on Foundations of Software Science and Computational Structures*. Springer-Verlag, 2010.

[83] Naoki Kobayashi and Atsushi Igarashi. Model-Checking Higher-Order programs with recursive types. In *European Symposium on Programming*. Springer Berlin Heidelberg, 2013.

[84] C. H. Luke Ong and Steven J. Ramsay. Verifying higher-order functional programs with pattern-matching algebraic data types. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, January 2011.

[85] Naoki Kobayashi and C. H. Luke Ong. A type system equivalent to the modal Mu-Calculus model checking of Higher-Order recursion schemes. In *2009 24th Annual IEEE Symposium on Logic In Computer Science*. IEEE, August 2009.

[86] Naoki Kobayashi. Model-checking higher-order functions. In *Proceedings of the 11th ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*. ACM, 2009.

[87] John C Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the 17th Annual IEEE Symposium on Logic in Computer Science*, LICS, pages 55–74. IEEE, 2002.

[88] Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Computer science logic*, CSL, pages 1–19. Springer, 2001.

[89] Lars Birkedal, Bernhard Reus, Jan Schwinghammer, and Hongseok Yang. A simple model of separation logic for Higher-Order store. In *Automata, Languages and Programming*, volume 5126. Springer Berlin Heidelberg, 2008.

[90] James Laird. A fully abstract trace semantics for general references. *Automata, Languages and Programming*, pages 667–679, 2007.

[91] Gogul Balakrishnan and Thomas Reps. Recency-Abstraction for Heap-Allocated storage static analysis. In *Proceedings of the 13th international conference on Static Analysis*, volume 4134, chapter 15. Springer Berlin / Heidelberg, 2006.

[92] Matthew Might and Olin Shivers. Environment analysis via $ Delta$CFA. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2006.

[93] Alan M. Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London Mathematical Society*, s2-42(1):230–265, 1937.

[94] Martin Davis. *Computability and Unsolvability*. McGraw-Hill, 1958.

[95] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Terminator: beyond safety. In *International Conference on Computer Aided Verification*, pages 415–418. Springer, 2006.

[96] Jan Midtgaard. Control-flow analysis of functional programs. *ACM Comput. Surv.*, 44(3), June 2012.

[97] Neil D Jones and Nina Bohr. Termination analysis of the untyped λ-calculus. In *International Conference on Rewriting Techniques and Applications*, pages 1–23. Springer, 2004.

[98] Jeffrey Mark Siskind and Barak A. Pearlmutter. First-class nonstandard interpretations by opening closures. In *Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '07, pages 71–76, New York, NY, USA, 2007. ACM.

[99] David Van Horn and Matthew Might. Abstracting abstract machines: a systematic approach to higher-order program analysis. *Communications of the ACM*, 54, September 2011.

[100] John Clements and Matthias Felleisen. A tail-recursive machine with stack inspection. *ACM Trans. Program. Lang. Syst.*, 26(6), November 2004.

[101] Jacob Burnim, Nicholas Jalbert, Christos Stergiou, and Koushik Sen. Looper: Lightweight detection of infinite loops at runtime. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 161–169. IEEE Computer Society, 2009.

[102] Michael Carbin, Sasa Misailovic, Michael Kling, and Martin C Rinard. Detecting and escaping infinite loops with jolt. In *European Conference on Object-Oriented Programming*, pages 609–633. Springer, 2011.

[103] Michael Kling, Sasa Misailovic, Michael Carbin, and Martin Rinard. Bolt: on-demand infinite loop escape in unmodified binaries. In *ACM SIGPLAN Notices*, volume 47, pages 431–450. ACM, 2012.

[104] James Bailey, Alexandra Poulovassilis, and Peter Newson. A dynamic approach to termination analysis for active database rules. In John Lloyd, Veronica Dahl, Ulrich Furbach, Manfred Kerber, Kung-Kiu Lau, Catuscia Palamidessi, Luís Moniz Pereira, Yehoshua Sagiv, and Peter J. Stuckey, editors, *Computational Logic — CL 2000*, pages 1106–1120, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[105] Yi-Dong Shen, Jia-Huai You, Li-Yan Yuan, Samuel S. P. Shen, and Qiang Yang. A dynamic approach to characterizing termination of general logic programs. *ACM Trans. Comput. Logic*, 4(4):417–430, October 2003.

[106] Michael Codish and Cohavit Taboch. A semantic basis for termination analysis of logic programs and its realization using symbolic norm constraints. In *Proceedings of the 6th International Joint Conference on Algebraic and Logic Programming*, ALP '97-HOA '97, pages 31–45, London, UK, UK, 1997. Springer-Verlag.

[107] Damien Sereni. *Termination analysis of higher-order functional programs.* PhD thesis, Oxford University, 2006.

[108] Michael Codish, Vitaly Lagoon, and Peter J. Stuckey. Testing for termination with monotonicity constraints. In Maurizio Gabbrielli and Gopal Gupta, editors, *Logic Programming*, pages 326–340, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[109] Niki Vazou, Eric L Seidel, and Ranjit Jhala. From safety to termination and back: Smt-based verification for lazy languages. *arXiv preprint arXiv:1401.6227*, 2014.

[110] Hongwei Xi. Dependent types for program termination verification. *Higher-Order and Symbolic Computation*, 15(1):91–131, 2002.

[111] Sven Eric Panitz and Manfred Schmidt-Schauß. Tea: Automatically proving termination of programs in a non-strict higher-order functional language. In Pascal Van Hentenryck, editor, *Static Analysis*, pages 345–360, Berlin, Heidelberg, 1997. Springer Berlin Heidelberg.

[112] Ashutosh Gupta, Thomas A. Henzinger, Rupak Majumdar, Andrey Rybalchenko, and Ru-Gang Xu. Proving non-termination. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, POPL '08, pages 147–158. ACM, 2008.

[113] Helga Velroyen and Philipp Rümmer. Non-termination checking for imperative programs. In *International Conference on Tests and Proofs*, pages 154–170. Springer, 2008.

[114] Jürgen Giesl, Peter Schneider-Kamp, and René Thiemann. Aprove 1.2: Automatic termination proofs in the dependency pair framework. In *International Joint Conference on Automated Reasoning*, pages 281–286. Springer, 2006.

[115] Jürgen Giesl, René Thiemann, Stephan Swiderski, and Peter Schneider-Kamp. Proving termination by bounded increase. In *International Conference on Automated Deduction*, pages 443–459. Springer, 2007.

[116] Jürgen Giesl, Matthias Raffelsieper, Peter Schneider-Kamp, Stephan Swiderski, and René Thiemann. Automated termination proofs for haskell by term rewriting. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 33(2):7, 2011.

[117] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. The dependency pair framework: Combining techniques for automated termination proofs. In *International Conference on Logic for Programming Artificial Intelligence and Reasoning*, pages 301–331. Springer, 2005.

[118] Thomas Arts and Jürgen Giesl. Termination of term rewriting using dependency pairs. *Theoretical Computer Science*, 236(1-2):133–178, 2000.

[119] Jürgen Giesl, René Thiemann, and Peter Schneider-Kamp. Proving and disproving termination of higher-order functions. In Bernhard Gramlich, editor, *Frontiers of Combining Systems*, pages 216–231, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[120] René Thiemann and Jürgen Giesl. Size-change termination for term rewriting. In *International Conference on Rewriting Techniques and Applications*, pages 264–278. Springer, 2003.

[121] Byron Cook, Andreas Podelski, and Andrey Rybalchenko. Termination proofs for systems code. In *ACM SIGPLAN Notices*, volume 41, pages 415–426. ACM, 2006.

[122] Andreas Podelski and Andrey Rybalchenko. A complete method for the synthesis of linear ranking functions. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 239–251. Springer, 2004.

[123] Andreas Podelski and Andrey Rybalchenko. Transition invariants. In *Logic in Computer Science, 2004. Proceedings of the 19th Annual IEEE Symposium on*, pages 32–41. IEEE, 2004.

[124] Aliaksei Tsitovich, Natasha Sharygina, Christoph M Wintersteiger, and Daniel Kroening. Loop summarization and termination analysis. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 81–95. Springer, 2011.

[125] Matthias Heizmann, Neil D Jones, and Andreas Podelski. Size-change termination and transition invariants. In *International Static Analysis Symposium*, pages 22–50. Springer, 2010.

[126] Elvira Albert, Puri Arenas, Michael Codish, Samir Genaim, Germán Puebla, and Damiano Zanardini. Termination analysis of java bytecode. In *International Conference on Formal Methods for Open Object-Based Distributed Systems*, pages 2–18. Springer, 2008.

[127] Josh Berdine, Byron Cook, Dino Distefano, and Peter W O'hearn. Automatic termination proofs for programs with shape-shifting heaps. In *International Conference on Computer Aided Verification*, pages 386–400. Springer, 2006.

[128] Fausto Spoto, Fred Mesnard, and Étienne Payet. A termination analyzer for java bytecode based on path-length. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 32(3):8, 2010.

[129] Marc Brockschmidt, Richard Musiol, Carsten Otto, and Jürgen Giesl. Automated termination proofs for java programs with cyclic data. In *International Conference on Computer Aided Verification*, pages 105–122. Springer, 2012.

[130] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM (JACM)*, 50(5):752–794, 2003.

[131] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: from scripts to programs. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*. ACM, 2006.

[132] Jacob Matthews and Robert B. Findler. Operational semantics for multi-language programs. In *POPL '07: Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. ACM, 2007.

[133] Jeremy G Siek and Walid Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, volume 6, pages 81–92, 2006.

[134] Jessica Gronski, Kenneth Knowles, Aaron Tomb, Stephen N Freund, and Cormac Flanagan. Sage: Hybrid checking for flexible specifications. In *Scheme and Functional Programming Workshop*, volume 6, pages 93–104, 2006.

[135] Asumu Takikawa, Daniel Feltey, Ben Greenman, Max S New, Jan Vitek, and Matthias Felleisen. Is sound gradual typing dead? In *ACM SIGPLAN Notices*, volume 51, pages 456–468. ACM, 2016.

[136] Ben Greenman and Matthias Felleisen. A spectrum of type soundness and performance. *Proceedings of the ACM on Programming Languages*, 2(ICFP):71, 2018.