Relatively Complete Counterexamples for Higher-Order Programs

Abstract

In this paper, we study the problem of generating inputs to a higher-order program causing it to error. We first study 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 in the sense of Hoare logic; completeness is reduced to the completeness of 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.

Keywords Higher-order programs; symbolic execution; contracts; testing

1. Introduction

Generating inputs that crash first-order programs is a well-known and well-studied problem in the literature on symbolic execution [Cadar et al. 2006; Godefroid et al. 2005], type systems [Foster et al. 2002], flow analysis [Xie and Aiken 2005], and software model checking [Yang et al.]. 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., Rondon et al. [2008]; Xu et al. [2009]; Kawaguchi et al. [2010]; Vytiniotis et al. [2013]; Nguyen et al. [2014]).

There are, however, a few notable exceptions which tackle the problem of counterexamples for higher programs. Perhaps the most successful has been the approach of random testing found in tools such as *QuickCheck* [Claessen and Hughes 2000; Klein et al. 2010]. While testing works well, it is not a complete method and often fails to generate inputs for which a little symbolic reasoning would go a long way (Section 2 discusses such an example in detail). 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 [Tobin-Hochstadt and Van Horn 2012; Nguyen et al. 2014]. Higher-order model checking [Kobayashi 2013] 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). Zhu and Jagannathan [2013] give 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 paper, 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 accumates a path condition as a symbolic heap. The semantics is an adaptation of Nguyen et al. [2014], where the critical technical distinction is our semantics maintains a *complete* path condition during execution. The key insight of this work is that 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, much like Hoare logic [Cook 1978].

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 seemlessly scales to handle first-class behavioral contracts [Findler and Felleisen 2002] by incorporating off-the-self semantics for contract monitoring [Dimoulas et al. 2012] 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 CPCF.

Contributions

We make the following contributions

1. We give a novel symbolic execution semantics for the PCF language which gradually refined unknown values and maintains a complete path condition.
2. We give a method of integrating a first-order solver to not only obtain a precise execution of symbolic programs, but also enable construction of higher-order counterexamples in case of errors.
3. We prove that 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 and give preliminary results of benchmarking the method of existing functional programs.

Outline

The remainder of the paper is organized as follows. We first step through a worked example of a higher-order program that consumes functional inputs (§ 2). The example is simple, but thwarts approaches to counterexample generation based on random testing. 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 a 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 (§ 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 (§ 4). We use these extensions as the basis of a tool for finding contract violations in Racket code to validate our approach (§ 5). Finally, we relate our work to the literature (§ 6) and conclude (§ 7).

2. Worked Examples

We illustrate our idea using the following incomplete OCaml program. Below, • denotes an unknown value of the appropriate type. Each subscript is unique to identify some unknown but fixed value.

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

Through symbolic execution and gradual refinement of unknown values, we are able to construct a counterexample triggering division by zero. In general, the program may have multiple unknown values, and the system tries to find one combination of such that breaks the program.

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

To find counterexamples, we first seek possible errors by running the program under an abstract semantics allowing opaque values. When execution follows different branches, it remembers assumptions associated with each path, and opaque values become partially transparent. To keep track of incremental refinements throughout execution, we allocate all values and maintain a heap mapping each location to an upperbound of their behavior. We rewrite the original execution of the main expression as follow, where the unknown value $\bullet_1$ is now allocated at location L_1.

$$(L_1\ \texttt{f}),\ [L_1 \mapsto \bullet]$$

Even though the function at L_1 is arbitrary, the only operation that impacts known program portions is applying its argument `f`. To soundly approximate all actual applications, we assume this unknown context applies `f` to yet other unknown values. The main expression therefore applies `f` to unknown values L_2 and L_3. We also refine the originally unknown value at L_1 accordingly.

$$\begin{aligned}(\texttt{f}\ L_2\ L_3),\ [L_1 &\mapsto \texttt{fun f} \to (\texttt{f}\ L_2\ L_3),\\ L_2 &\mapsto \bullet^{\texttt{int}\to\texttt{int}},\\ L_3 &\mapsto \bullet^{\texttt{int}}]\end{aligned}$$

We execute `f`'s body, substituting `g` with L_2 and `n` with L_3. Applying unknown function L_2 to unknown integer L_3 gives another unknown integer L_4. Assuming functions are pure, we model unknown functions on integers as mappings and refine L_2 to map L_3 to L_4. We straightforwardly extend primitive operations to work on abstract values. Subtraction of L_4 from `100` in turn results in another unknown integer L_5. We refine L_5 by a predicate remembering that it is the subtraction of L_4 from `100`.

$$\begin{aligned}1\ /\ L_5,\ [L_1 &\mapsto \texttt{fun f} \to (\texttt{f}\ L_2\ L_3),\\ L_2 &\mapsto \texttt{case}^{\texttt{int}\to\texttt{int}}\ [L_3 \mapsto L_4],\\ L_3 &\mapsto \bullet^{\texttt{int}},\\ L_4 &\mapsto \bullet^{\texttt{int}},\\ L_5 &\mapsto \bullet^{\texttt{int}, \texttt{fun x} \to \texttt{x = 100 -}\ L_4}]\end{aligned}$$

The resulting division evaluates to different results depending on the value of denominator $\mathtt{L}_5$. In the error case where $\mathtt{L}_5$ is zero, we have the following refinements on the heap:

$$\begin{aligned}\mathtt{error},\ [\mathtt{L}_1 &\mapsto \mathtt{fun\ f} \to (\mathtt{f\ L}_2\ \mathtt{L}_3),\\ \mathtt{L}_2 &\mapsto \mathtt{case}^{\mathtt{int}\to\mathtt{int}}\ [\mathtt{L}_3 \mapsto \mathtt{L}_4],\\ \mathtt{L}_3 &\mapsto \bullet^{\mathtt{int}},\\ \mathtt{L}_4 &\mapsto \bullet^{\mathtt{int}},\\ \mathtt{L}_5 &\mapsto \mathtt{0}^{\{\mathtt{fun\ x} \to \mathtt{x = 100 - L}_4\}}]\end{aligned}$$

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 [De Moura and Bjørner 2008] can easily solve such constraint and yield ($\mathtt{L}_3 = \mathtt{0}$, $\mathtt{L}_4 = \mathtt{100}$, $\mathtt{L}_5 = \mathtt{0}$) as a model. We then plug these values into the current heap and straightforwardly obtain the counterexample as shown in the beginning. In summary, we use execution to incrementally construct the shapes of functions, and query a first-order solver for a model for base values, from which we combine and construct higher-order counterexamples.

3. Formal Model with Symbolic PCF

This section presents a reduction system illustrating the core of our approach. Symbolic PCF (SPCF) [Tobin-Hochstadt and Van Horn 2012] extends the PCF language [Scott 1993] with *symbolic values* that can be higher-order. We present the language's syntax and semantics, integration with an external solver to improve precision, and how the semantics enables the generation of counterexamples when errors occur. Finally, we prove that our counterexample synthesis is sound and complete relative to the underlying solver.

3.1 Syntax of SPCF

Figure 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 standard expresion forms for conditions, application, primitive applications, recursion, and standard values such as natural numbers and lambdas. 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 denote an unknown but fixed and syntactically closed value [1] of type T. The system automatically annotates each opaque value written in the source code with a unique label to identify the value. In addition, it annotates each primitive application to identify an error source.

Because an SPCF expression can contain unknown values, we allocate all values and maintain a heap to keep track of their constraints. When execution proceeds through branches and primitive applications, we refine the heap at appropriate locations with stronger assumptions taken at each branch. As figure 1 shows, a heap is a finite function mapping each location L to a storable S as an upper-bound of the value's behavior at run-time. A storable S is mostly similar to a value, but an opaque storable can be further refined by arbitrary program predicates. For example, $\bullet^{\{\mathtt{nat},\ \mathtt{\lambda x.even?\ x}\}}$ denotes an unknown even natural. In addition, we use $(\mathtt{case}^T\ \overrightarrow{[L \mapsto L]})$ to denote a mapping approximating an unknown function of type $(\mathtt{nat} \to T)$. We clarify the role of this construct later when discussing the semantics of applying opaque functions.

Expressions	E	::=	$A \mid V \mid X \mid \mathsf{if}\ E\ E\ E$
		$\mid$	$E\ E \mid O\ \overrightarrow{E}^{\ L} \mid \mu X : T.E$
Values	V	::=	$\bullet_L^T \mid \lambda X : T.E \mid n$
Answers	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	::=	$\mathtt{nat} \mid T \to T$
Stores	Σ	::=	$\emptyset \mid \Sigma, L \mapsto S$
Storables	S	::=	$\bullet^{\{T\ \overrightarrow{P}\}} \mid \lambda X : T.E \mid n$
		$\mid$	$\mathtt{case}^T\ \overrightarrow{L \mapsto L}$
Variables	X, L	$\in$	*identifier*

Figure 1. Syntax of SPCF

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 $\mathtt{err}_O^L$ denoting a violation of source location L failing a precondition from primitive O

We omit typechecking rules for SPCF as they are either standard or straightforward. In the following, we assume programs are well-typed, and primitives are applied with correct arities. In addition, we omit showing types and labels for constructs such as locations and lambdas when they are irrelevant or clear from context.

3.2 Semantics of SPCF

We present the semantics of SPCF as a relation between states of the form $\langle E, \Sigma \rangle$. 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 that on concrete states, accounting for all possible instantiation of symbolic values. We formalize the notion of approximation later in section 3.6. Figure 2 presents the reduction semantics of SPCF.

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

All values are allocated on the heap and reduce to locations as shown in rules *Opq1*, *Opq2*, and *Conc*. Because opaque values stand for arbitrary but fixed values, we reuse the location if it has been previously allocated.

Rule *Prim* shows the reduction of a primitive application. Relation δ relates 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 remembering 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. Application of a λ-abstraction follows standard β-reduction.

Application of an unknown value results in several possibilities. Consider the following example:

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

Different instantiations of $\texttt{L}_1$ result in different answers. For example, in the following, (1) raises an error from within the unknown function $\texttt{L}_1$ itself, (2) triggers the division error at `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.

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

As we only analyze concrete program portions, we ignore possibility (1) and only consider cases where the application succeeds or the argument escapes to an unknown context invoking a potential error in a concrete program portion.

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 context results in the same result. Rule *AppOpq3* considers the case where the unknown context does not immediately explore its argument behavior but delay 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.

One may suspect that rule *AppOpq3* generalizes *AppOpq2*, but this is not the case. The absence of rule *AppOpq2* would require that the opaque context eventually explores the first argument before continuing with the next one, failing to model cases where it completely skips the first argument. For example, the following application would either continue wrapping the first argument (by rule *AppOpq2*) or apply it and halt on the first error (by rule *AppHavoc*) and never reach the next argument.

$$(\bullet^{(\texttt{nat}\to\texttt{nat})\to\texttt{nat}\to\texttt{nat}}\ \ (\texttt{λx.(/ 1 0)}))\ \texttt{1}$$

On the other hand, the presence of rule *AppOpq3* is clearly important to soundly model opaque applications. When the argument is higher-order, we do not assume a simple dispatch as in rule *AppOpq1* because there is no mechanism for comparing functions for equality (without applying them as in rule *AppHavoc*). A non-constant higher-order function that does not apply its argument has a more restricted structure than an arbitrary mapping.

Finally, application rules for simple mappings are straightforward. Rule *App-Case-1* shows the case where the function has been previously applied to the argument and reuses the result location. Rule *App-Case-2* shows the case where the argument is new and allocate a new location for the result.

These rules for opaque application collectively model the demonic context in previous works on higher-order symbolic execution [Tobin-Hochstadt and Van Horn 2012], but they unroll the unknown context in an incremental fashion and remember its shape to allow constructing counterexamples at the end.

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 returns another integer and raises an error. The relation also remembers appropriate refinements to arguments and results at each branch. Figure 3 presents selective rules for primitive operations `zero?`, `add1`, and `div`. Throughout this paper, we abbreviate $\texttt{λ}X.(\texttt{=}X\,E)$ 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.

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 value at L definitely satisfies predicate P, which implies that all possible instantiations of L satisfy P. In the same way, $\Sigma \vdash L : P\text{✗}$ means all instantiations of L definitely fail P. Finally, $\Sigma \vdash L : P\,\textbf{?}$ 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

$\langle \bullet^T_L, \Sigma\rangle \rightsquigarrow \langle L^T, \Sigma[L \mapsto \bullet^T]\rangle$ if $L^T \notin dom(\Sigma)$ [*Opq1*]

$\langle \bullet^T_L, \Sigma\rangle \rightsquigarrow \langle L^T, \Sigma\rangle$ if $L^T \in dom(\Sigma)$ [*Opq2*]

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

$\langle \mu X : T.E, \Sigma\rangle \rightsquigarrow \langle [\mu X : T.E/X]E, \Sigma\rangle$ [*Rec*]

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

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

$\langle (O\ L), \Sigma\rangle \rightsquigarrow \langle L', \Sigma'[L' \mapsto V]\rangle$ if $\delta(\Sigma, O, L) \ni (V, \Sigma')$ and $L' \notin dom(\Sigma')$ [*Prim*]

$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle [L_x/X]E, \Sigma\rangle$ if $\Sigma\,(L) = \lambda X.E$ [*AppLam*]

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

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

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

$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle (L_2\ (L_x\ L_1)), \Sigma[L \mapsto V, L_1 \mapsto \bullet^{T_1}, L_2 \mapsto \bullet^{T_2\to T}]\rangle$ 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*]

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

$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle L_a, \Sigma[L \mapsto \mathtt{case}\ [L_z \mapsto L_b]\ldots [L_x \mapsto L_a]]\rangle$ 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*]

Figure 2. Semantics of SPCF

$\delta(\Sigma, \mathtt{zero?}, L) \ni (1, \Sigma)$ if $\Sigma \vdash L : \mathtt{zero?}\checkmark$

$\delta(\Sigma, \mathtt{zero?}, L) \ni (0, \Sigma)$ if $\Sigma \vdash L : \mathtt{zero?}\times$

$\delta(\Sigma, \mathtt{zero?}, L) \ni (1, \Sigma[L \mapsto 0])$ if $\Sigma \vdash L : \mathtt{zero?}\,?$

$\delta(\Sigma, \mathtt{zero?}, L) \ni (0, \Sigma[L \mapsto \bullet^{\{\mathsf{nat},\, \neg\mathtt{zero?}\}}])$ if $\Sigma \vdash L : \mathtt{zero?}\,?$

$\delta(\Sigma, \mathtt{add1}, L) \ni (n+1, \Sigma)$ if $\Sigma(L) = n$

$\delta(\Sigma, \mathtt{add1}, L) \ni (\bullet^{\mathsf{nat},\, (\equiv\ (L+1))}, \Sigma)$ if $\Sigma(L) = \bullet^{\mathsf{nat}}$

$\delta(\Sigma, \mathtt{div}, L_1, L_2) \ni (m/n, \Sigma)$ if $\Sigma(L_1) = m$ and $\Sigma(L_2) = n, n \neq 0$

$\delta(\Sigma, \mathtt{div}, L_1, L_2) \ni (\bullet^{\mathsf{nat},\, (\equiv\ L_1\ /\ L_2)}, \Sigma')$ if $\Sigma(L_2) \neq n$ and $\delta(\Sigma, \mathtt{zero?}, L_2) \ni (0, \Sigma')$

$\delta(\Sigma, \mathtt{div}, L_1, L_2) \ni (\mathtt{err}_{\mathtt{div}}, \Sigma')$ if $\Sigma(L_2) \neq n$ and $\delta(\Sigma, \mathtt{zero?}, L_2) \ni (1, \Sigma')$

Figure 3. Selected Primitive Operations

make the execution sound but not very precise.) Instead of implementing our own proof system, we rely on an existing SMT solver for sophisticated reasoning of numeric values.

We define a translation $\{\!|\cdot|\!\}$ of run-time constructs into logical formulas, which is shown in figure 4. The translation of a heap is the conjunction of formulas obtained from each mapping, and the translation of each mapping is straightforward. In particular, a location pointing to a concrete number translates to the obvious assertion on equality, and a mapping ($\mathtt{case}\ \overrightarrow{L \mapsto L}$) adds constraints asserting that equal inputs imply equal outputs. Notice that the proof system only needs to handle a small number of well-understood predicates. 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 $(\lambda \mathtt{x}.(\mathtt{or}\ (P\ \mathtt{x})\ E))$ 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.

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 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.

3.5 Constructing Counterexamples

For each answer reached by evaluation, the heap contains refinements to symbolic values in order to reach such result. In particular, refinements on the heap in an error case describes 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 describing how to reproduce the current path. Indeed,

$$\{\!\{\overrightarrow{L \mapsto S}\}\!\} = \bigwedge \overrightarrow{\{\!\{L \mapsto S\}\!\}}$$
$$\{\!\{L \mapsto n\}\!\} = (L = n)$$
$$\{\!\{L \mapsto \bullet^{\mathsf{nat}}\ \overrightarrow{P}\}\!\} = \bigwedge \overrightarrow{\{\!\{L : P\}\!\}}$$
$$\{\!\{L \mapsto \mathsf{case}\ \_ \ldots [L_1 \mapsto L_2]\ \_ \ldots [L_3 \mapsto L_4]\ \_ \ldots\}\!\} = (\bigwedge\ ((L_1 = L_3) \Rightarrow (L_2 = L_4)) \ldots)$$
$$\{\!\{L : (\lambda X.\mathsf{zero?}\ X\ )\}\!\} = (L\ =\ 0)$$
$$\{\!\{L : (\lambda X.(= (X(+\ L_1\ L_2))))\}\!\} = (L = (L_1 + L_2))$$
$$\ldots$$

Figure 4. Translation of Heap

$$\frac{\{\!\{\Sigma\}\!\} \Rightarrow \{\!\{L : P\}\!\} \text{ is valid}}{\Sigma \vdash L : P \checkmark} \quad \textit{Proved}$$

$$\frac{\{\!\{\Sigma\}\!\} \wedge \{\!\{L : P\}\!\} \text{ is unsat}}{\Sigma \vdash L : P \times} \quad \textit{Refuted}$$

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

Figure 5. Proof Rules

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 whether this process always generates actual counterexamples (sound), and it always succeeds if one exists (complete). The next section clarifies these points.

3.6 Soundness and Completeness of Counterexamples

We show that our method of finding counterexamples is sound, which means that any counterexample found by the system is real. In addition, we show that the method is complete up to the underlying first-order solver, which means that if any concrete program portion can cause an error and the underlying solver always finds a model for any satisfiable formula, then we can find a counterexample.

We first define what it means for one state to instantiate or abstract another.

3.6.1 Approximation Relation

We define an approximation relation between program states of the form $\langle E, \Sigma\rangle$. We write $\langle E_1, \Sigma\rangle \sqsubseteq \langle E_2, \hat{\Sigma}\rangle$ to mean "$\langle E_2, \hat{\Sigma}\rangle$ approximates the fully concrete state $\langle E_1, \Sigma\rangle$", or "$\langle E_1, \Sigma\rangle$ refines $\langle E_2, \hat{\Sigma}\rangle$". For example, $\langle (+\ \mathsf{L}_1\ 2), [\mathsf{L}_1 \mapsto 0]\rangle \sqsubseteq \langle (+\ \mathsf{L}_3\ \bullet^{\mathsf{nat}}_{\mathsf{L}_4}), \mathsf{L}_3 \mapsto \bullet^{\mathsf{nat}}\rangle$.

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

Figure 6. Computing Concrete Labels

Because only errors from concrete program portions are important in the analysis, we parameterize the approximation relation with a set of labels marking concrete source locations. This set can be computed straightforwardly as shown in figure 6. Hence, $\langle E, \Sigma\rangle \sqsubseteq^{\overrightarrow{L}} \langle E', \hat{\Sigma}\rangle$ means that $\langle E', \hat{\Sigma}\rangle$ approximates $\langle E, \Sigma\rangle$ up to program portions marked by labels $\overrightarrow{L}$, predicting all possible errors coming from these sources, but stating nothing about errors from other portions.

Finally, to enforce that each symbolic value in the abstract state consistently approximates one concrete value, we parameterize the relation with a function F mapping each location from the abstract state to a concrete value (either a lambda or a number).

For example, in the following, the abstract state $(+\ \mathsf{L}_3\ \mathsf{L}_4)$ approximates the concrete state $(+\ \mathsf{L}_1\ \mathsf{L}_2)$, where function $\mathsf{F} = [\mathsf{L}_3 \mapsto 1, \mathsf{L}_4 \mapsto 2]$ prevents any label to ambiguously approximate two different values. In addition, the only concrete program portion considered is the application itself (labeled L).

$$\langle (+\ \mathsf{L}_1\ \mathsf{L}_2)^{\mathsf{L}}, [\mathsf{L}_1 \mapsto 1, \mathsf{L}_2 \mapsto 2]\rangle \sqsubseteq^{\mathsf{F}}_{\{\mathsf{L}\}}$$
$$\langle (+\ \mathsf{L}_3\ \mathsf{L}_4)^{\mathsf{L}}, [\mathsf{L}_3 \mapsto \bullet, \mathsf{L}_4 \mapsto \bullet]\rangle$$

Figure 7 shows selective rules for approximation between states. We omit straightforward structural rules. Rule *Loc-Loc* states that location L_2 approximates location L_1 if the value pointed to by L_2 approximates that by L_1, provided this approximation is consistent according to function F. Rule *Opq* states that an opaque application approximates all non-answer expression up to labels reachable from the arguments. Rule *Case1* states that an opaque mapping approximates any function on natural numbers as long as the function's body contains no concrete label. Rule *Case2* states that if a mapping approximates a function and we further know the result of applying the concrete function on another value, the we can refine the mapping to remember this result and obtain anoter (tighter) approximation of the function. Finally, rule *OpqErr* allows us to ignore errors coming from unknown program portions.

3.6.2 Theorems

To prove soundness, we show that constraints on the heap is sufficiently precise that all instantiations follow the same

$$\textit{Nat-Opq}\quad \frac{\Sigma \vdash n : F(P) \checkmark \ \ldots}{\langle n, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle \bullet^{\mathsf{nat}\ P \ldots}, \hat{\Sigma}\rangle}$$

$$\textit{Loc-Loc}\quad \frac{F(L_2) = \Sigma(L_1) = V \qquad \langle V, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle \hat{\Sigma}(L_2), \hat{\Sigma}\rangle}{\langle L_1, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle L_2, \hat{\Sigma}\rangle}$$

$$\textit{Opq}\quad \frac{E \neq A \qquad \hat{\Sigma}(L_0) = \bullet^{T_1 \to T} \qquad lab_\Sigma[\![E]\!] \cap \overrightarrow{L} \subseteq lab_{\hat{\Sigma}}[\![\hat{\Sigma}(L_1)]\!]}{\langle E, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle (L_0\ L_1), \hat{\Sigma}\rangle}$$

$$\textit{OpqErr}\quad \frac{L' \notin \overrightarrow{L}}{\langle \mathsf{err}^{L'}, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle E, \hat{\Sigma}\rangle}$$

$$\textit{Case1}\quad \frac{lab_\Sigma[\![E]\!] \cap \overrightarrow{L}}{\langle \lambda X : \mathsf{nat}.E, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle \mathsf{case}\ [], \hat{\Sigma}\rangle}$$

$$\textit{Case2}\quad \frac{\begin{array}{c} lab_\Sigma[\![E]\!] \cap \overrightarrow{L} = \emptyset \\ \langle L_x, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle L_y, \hat{\Sigma}\rangle \qquad \langle L_a, \Sigma'\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle L_b, \hat{\Sigma}\rangle \\ \langle ((\lambda X : \mathsf{nat}.E)\ L_x), \Sigma\rangle \rightsquigarrow^\star \langle L_a, \Sigma'\rangle \\ \langle \lambda X : \mathsf{nat}.E, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle \mathsf{case}\ [L_1 \mapsto L_2, \ldots], \hat{\Sigma}\rangle \end{array}}{\langle \lambda X : \mathsf{nat}.E, \Sigma\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle \mathsf{case}\ [L_1 \mapsto L_2, \ldots L_y \mapsto L_b], \hat{\Sigma}\rangle}$$

Figure 7. Approximation

execution branch. To prove (relative) completeness, we show that the abstract reduction semantics is sound and assume the solver can answer all our first-order queries.

Theorem 1 (Soundness of Counterexample)**.** *If* $\langle E, \emptyset\rangle \rightsquigarrow^\star \langle A, \hat{\Sigma}\rangle$ *and* $\Sigma \sqsubseteq \hat{\Sigma}$ *then* $\langle E, \Sigma\rangle \rightsquigarrow^\star \langle A', \Sigma'\rangle$ *such that* $\langle A', \Sigma'\rangle \sqsubseteq \langle A, \hat{\Sigma}\rangle$.

Proof. By lemma 1. □

.

Lemma 1 (Completeness of Refinement in Reduction)**.** *If* $\langle E_1, \hat{\Sigma}_1\rangle \rightsquigarrow \langle E_2, \hat{\Sigma}_2\rangle$ *and* $\Sigma_1 \sqsubseteq \hat{\Sigma}_2$ *and* $\langle E_1', \Sigma_1\rangle \sqsubseteq \langle E_1, \hat{\Sigma}_2\rangle$ *then* $\langle E_1', \Sigma_1\rangle \rightsquigarrow^\star \langle E_2', \Sigma_2\rangle$ *such that* $\langle E_2', \Sigma_2\rangle \sqsubseteq \langle E_2, \hat{\Sigma}_2\rangle$.

Proof. By case analysis of the reduction and approximation relations. Base cases with primitive operations are deferred to lemma 2. □

Lemma 2 (Completeness of Refinement in Primitives)**.** *If* $\delta(\hat{\Sigma}, O, L) \ni (V, \hat{\Sigma}')$ *and* $\langle L', \Sigma\rangle \sqsubseteq \langle L, \hat{\Sigma}\rangle$ *then* $\delta(\Sigma, O, L') \ni (V', \Sigma')$ *such that* $\langle V', \Sigma'\rangle \sqsubseteq \langle V, \hat{\Sigma}'\rangle$.

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

Theorem 2 (Relative Completeness of Counterexamples)**.** *If* $\langle E, \Sigma\rangle \rightsquigarrow^\star \langle \mathsf{err}^L, \Sigma'\rangle$ *and* $\langle E, \Sigma\rangle \sqsubseteq_{\{L\}} \langle E, \hat{\Sigma}\rangle$ *then* $\langle E, \hat{\Sigma}\rangle \rightsquigarrow^\star \langle \mathsf{err}^L, \hat{\Sigma}'\rangle$ *such that* $\exists \Sigma'', \Sigma'' \sqsubseteq \hat{\Sigma}'$ *and* $\langle E, \Sigma''\rangle \rightsquigarrow^\star \langle \mathsf{err}^L, \Sigma'''\rangle$.

Proof. Soundness (lemma 3) of reduction relation ensures the error is reached in the abstract program, and completeness of the solver ensures a model is produced for the error case. □

Lemma 3 (Soundness of SPCF)**.** *If* $\langle E, \emptyset\rangle \rightsquigarrow^\star \langle A, \Sigma\rangle$ *and* $\langle E, \emptyset\rangle \sqsubseteq \langle E_1, \emptyset\rangle$*, then* $\langle E_1, \emptyset\rangle \rightsquigarrow^\star \langle A', \hat{\Sigma}\rangle$ *such that* $\langle A, \Sigma\rangle \sqsubseteq \langle A', \hat{\Sigma}\rangle$.

Proof. By lemma 4. □

Lemma 4 (Soundness of Reduction)**.** *If* $\langle E_1, \Sigma_1\rangle \rightsquigarrow \langle E_2, \Sigma_2\rangle$ *and* $\langle E_1, \Sigma_1\rangle \sqsubseteq^F_{\overrightarrow{L}} \langle E_1', \hat{\Sigma}_1\rangle$ *then* $\langle E_1', \hat{\Sigma}_1\rangle \rightsquigarrow^\star \langle E_2', \hat{\Sigma}_2\rangle$ *such that* $\langle E_2, \Sigma_2\rangle \sqsubseteq^{F'}_{\overrightarrow{L}} \langle E_2', \hat{\Sigma}_2\rangle$ *and* F' *is consistent with* F.

Proof. By case analysis of the approximation and reduction relations. The only non-straightforward case comes from approximation rule *Opq*.

- Case $E_1 \neq A$ and $E' = (L_0\ L_1)$ where $\hat{\Sigma}(L_0) = \bullet$
  - If $E_1' \neq A$, the opaque application continues to approximate it by rule *Opq*.
  - If $E_1' = \mathsf{err}^{L'}$ where $L' \notin lab_{\hat{\Sigma}}[\![L_1]\!]$, any expression approximates it by rule *OpqErr*.
  - If $E_1' = \mathsf{err}^{L'}$ where $L' \in lab_{\hat{\Sigma}}[\![L_1]\!]$, lemma 5 shows the error will be reached.
  - If $E_1' = L'$, the opaque application returns a fresh opaque value and refines the unknown function at L_0 appropriately depending on the shape of $F(L_0)$.

□

Lemma 5 (Soundness of Demonic Context)**.** *If there exists a context* $\mathcal{E}$ *such that* $\langle \mathcal{E}[L], \Sigma\rangle \rightsquigarrow^\star \langle \mathsf{err}_O^{L'}, \Sigma'\rangle$*, then* $\langle (\bullet\ L), \Sigma\rangle \rightsquigarrow^\star \langle \mathsf{err}_O^{L'}, \Sigma''\rangle$.

Proof. Any context $\mathcal{E}$ triggering the error has the canonical form of $\mathcal{E} ::= [\,] \mid (\mathcal{E}\ L_i)$. Rule *AppHavoc* produces an approximation of this of the form $\mathcal{E} ::= [\,] \mid (\mathcal{E}\ \bullet_i)$. □

$$\begin{array}{lll} \text{Operations} & O ::= & \ldots \mid \texttt{box?} \mid \texttt{box} \mid \texttt{unbox} \mid \texttt{set-box!} \\ \text{Storables} & S ::= & \texttt{box}\ L \end{array}$$

Figure 11. Syntax of SPCF with mutable boxes

$$\text{Operations}\ O ::= \ \ldots \mid \texttt{nat?} \mid \texttt{proc?}$$

Figure 8. Extended Syntax for Untyped SPCF

$$\begin{array}{l} \delta(\Sigma, \texttt{add1}, L) \ni (\bullet^{\texttt{nat?}(\equiv\ \texttt{(+ 1}\ L\texttt{))}}, \Sigma') \\ \qquad \text{if } \delta(\Sigma, \texttt{nat?}, L) \ni (\texttt{1}, \Sigma') \\ \delta(\Sigma, \texttt{add1}, L) \ni (\texttt{err}_{\texttt{add1}}, \Sigma') \\ \qquad \text{if } \delta(\Sigma, \texttt{nat?}, L) \ni (\texttt{0}, \Sigma') \end{array}$$

Figure 9. Extended Primitive Operations

4. Beyond PCF

We discuss three important extensions to our system for finding counterexamples in a realistic language: dynamic typing, contracts, and mutable states.

4.1 Dynamic Typing

Dynamically typed languages defer safety checks to run-time to avoid conservative rejection of good 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 as in figure 8. Changes to the semantics are straightforward, where run-time checks are inserted into applications to ensure a function is being applied, and into primitive applications to ensure arguments have the right tags. Figure 9 shows an example of the changes to meta-function δ on operator `add1`.

Another important change occurs in rules for applying opaque functions as shown in figure 10. Rules *UAppOpq1* and *UAppHavoc* are analogous to previous rules *AppOpq2* and *AppHavoc*, and rule *UAppOpq2* is analogous to the two rules *AppOpq1* and *AppOpq3* with a dynamic check distinguishing flat and higher-order arguments.

4.2 Contracts

Contracts generalize pre-and-post conditions to higher-order specifications [Findler and Felleisen 2002], allowing programmers to express rich invariants using arbitrary code. They can either refine an existing type system [Hinze et al. 2006], or are the main mechanism for ensuring safety in an untyped language. Although the semantics of contract checking can be complex [Greenberg et al. 2010; Dimoulas et al. 2011], they introduce no new challenge into 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. By faithfully following the contract checking semantics, the system can construct counterexamples to higher-order contracts by only relying on a first-order solver.

4.3 Mutable States

We support stateful programs by extending the language with primitives for mutable boxes as shown in figure 11.

The major change appears in rules for applying unknown functions as shown in figure 12. If the argument is a mutable box, the unknown function may invoke its content and perform arbitrary changes to the box. Second, if its argument is a function, the unknown context may apply the function any number of times, affecting its internal state arbitrarily. Finally, in the presence of mutable states, the system can no longer assume that the unknown function depends solely on its argument and yields equal outputs for equal inputs, so a memoized mapping is no longer applicable. Because this last change can be too conservative for reasoning about idiomatic functional programs, where programmers often think of functions as pure and use mutable boxes judiciously, it is useful in a practical system to have a special annotation for marking an unknown function as pure [2].

One challenge introduced by mutable states is aliasing. For example, a result from applying an unknown function can either be a fresh value, or any previous value on the heap. Future side effects performed on this unknown result may or may not affect some existing mutable box. To soundly execute symbolic programs with mutable states, we modify the behavior of primitive `box?`. When an unknown value L is determined to be a mutable box, we either decide it is a distinct box ($\texttt{box}\ L'$) from any previous one on the heap, or an alias to one previous box L_i by substituting each occurence of L for L_i in the entire program. Although this process is expensive, mutable boxes are sparse in idiomatic functional programs. Programs with no invocation of `box?` (which is implicit in `unbox` and `set-box!`) do not pay for this penalty. More efficient handling of aliasing of large imperative programs is one direction of our future work.

5. Implementation

To validate our approach, we implement a tool generating counterexamples for erroneous programs written in a subset of Racket [Flatt and PLT 2010]. The implementation handles a much more realistic set of language features beyond SPCF. We discuss results on some benchmarks below.

5.1 Language features

First, our implementation supports all extensions discussed in section 4. Second, we extend the set of base values and primitive operations, such as pairs, strings and Racket's full numeric tower. This also introduces more error sources and

[2] First-class contracts can have internal states and enforce equal outputs for equal inputs, which symbolic execution can make use of.

Common precondition: $\Sigma(L) = \bullet^{\vec{P}}$ and $\delta(\Sigma, \texttt{proc?}, L) \ni (1, \Sigma')$

$$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle L_a, \Sigma'[L_a \mapsto \bullet, L \mapsto \lambda x.L_a]\rangle \quad \text{where } L_a \notin \mathit{dom}(\Sigma) \quad [\mathit{UAppOpq1}]$$

$$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle [L_x/x]E, \Sigma''\rangle \quad \text{where } L_1, L_2 \notin \mathit{dom}(\Sigma'), \quad [\mathit{UAppOpq2}]$$
$$\Sigma'' = \Sigma[L \mapsto \lambda x.E, L_1 \mapsto \bullet, L_2 \mapsto \texttt{case}\ []]$$
$$\text{and } E = (\texttt{if}\ (\texttt{proc?}\ x)\ (\lambda y.((L_1\ x)\ y))\ (L_2\ x))$$

$$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle ((L_2\ (L_x\ L_1)), \Sigma'[L \mapsto V, L_1 \mapsto \bullet, L_2 \mapsto \bullet]\rangle \quad \text{where } L_1, L_2 \notin \mathit{dom}(\Sigma) \text{ and } V = \lambda x.(L_2\ (x\ L_1)) \quad [\mathit{UAppHavoc}]$$

Figure 10. Unknown Function Application in Untyped SPCF

Common precondition: $\Sigma(L) = \bullet^{\vec{P}}$ and $\delta(\Sigma, \texttt{proc?}, L) \ni (1, \Sigma')$

$$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle [L_x/x]E, \Sigma''\rangle \quad \text{where } L_1, L_2 \notin \mathit{dom}(\Sigma') \quad [\mathit{SAppOpq1}]$$
$$\text{and } \Sigma'' = \Sigma'[L \mapsto \lambda x.E, L_1 \mapsto \bullet, L_2 \mapsto \bullet]$$
$$\text{and } E = (L_2\ (\texttt{set-box!}\ x\ (L_1\ (\texttt{unbox}\ x))))$$

$$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle [L_x/x]E, \Sigma''\rangle \quad \text{where } L_1, L_2 \notin \mathit{dom}(\Sigma') \quad [\mathit{SAppOpq2}]$$
$$\text{and } \Sigma'' = \Sigma'[L \mapsto \lambda x.E, L_1 \mapsto \bullet, L_2 \mapsto \bullet]$$
$$\text{and } E = ((L_2\ (x\ L_1))\ x)$$

$$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle [L_x/x]V, \Sigma''\rangle \quad \text{where } L_1, L_2 \notin \mathit{dom}(\Sigma') \quad [\mathit{SAppOpq3}]$$
$$\text{and } \Sigma'' = \Sigma'[L \mapsto \lambda x.V, L_1 \mapsto \bullet]$$
$$\text{and } V = \lambda y.((L_1\ x)\ y)$$

$$\langle (L\ L_x), \Sigma\rangle \rightsquigarrow \langle L_a, \Sigma'[L_a \mapsto \bullet]\rangle \quad \text{where } L_a \notin \mathit{dom}(\Sigma') \quad [\mathit{SAppOpq4}]$$

Figure 12. Unknown Function Application in Stateful SPCF

interesting counterexamples. Third, we allow user-defined structures at the top-level. Fourth, we employ a module system to let users organize code. A module can export multiple values as well as define private ones for internal use. Finally, we support more contract combinators such as conjunction, disjunction, recursion, and contracts are first class values in our implementation, meaning they can be composed at runtime.

5.2 Evaluation

We test our tool's effectiveness in finding bugs from a collection of faulty programs, obtained by either introducing subtle but easy to make programming mistakes or weakening a contracts' preconditions. The tool is able to return with a concrete counterexample for each such program in a reasonable amount of time: the most complicated invariant violation takes 2 seconds to detect, and large typical programs take less than a second.

First, we find the approach more effective in generating counterexamples than random testing, thanks to knowledge of the program's source code. For example, the tool finds a counterexample to the following program quickly and automatically:

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

In contrast, without further customization, QuickCheck fails to find the easily seen counterexample for an equivalent Haskell program after 10,000 attempts.

In general, we find the tool particularly useful in speeding up bug-finding in an untyped language. For example, it is easy for programmers to forget that Racket supports the full numeric tower [St-Amour et al. 2012], and predicate `number?` admits complex numbers. The following definition of `argmin` has a wrong contract requiring its first argument to be a function returning a number instead of a real number, which can violate `min`'s requirement of real arguments.

```
;(argmin proc lst) → any/c
;  proc : (any/c → number?)
;  lst : (and/c pair? list?)
(define (argmin f xs)
  (cond [(empty? (rest xs))
         (f (first xs))]
        [else
         (min (f (first xs))
              (argmin f (rest xs)))]))
```

Our tool generates a counterexample:

```
f  : λx.0+1i
xs : (list 0 0)
```

In contrast, contract-guided random testing in Racket does not find this error without a slight modification to the precondition, changing `(and/c pair? list?)` to `(listof any/c)`. Otherwise the tool fails to generate inputs that satisfy the precondition in the first place due to the lack of

generator for conjunctive contracts in the general case.

The 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. Even though the approach is prone to combinatorial explosion as inherent in symbolic execution, this problem rarely shows up in our tests, because a precise solver helps eliminating spurious branches greatly, and the tool stops as soon as it finds a counterexample, which is also the smallest one due to breadth-first search. Nevertheless, we encounter one benchmark triggering our tool's initial weakness. In the following program, `insert`'s contract says the function preserves the Braun-tree invariant, but the implementation does not do so. The invariant is specified using ordinary code as in the recursive function `braun-tree?`.

```
(: insert (braun-tree? any → braun-tree?))
(define (insert bt x)
  (cond
   [(node? bt)
    (node (insert (node-left bt) x)
          (node-right bt))]
   [else (node x #f #f)]))

(define (braun-tree? x)
  (or (false? x)
      (and (node? x)
           (braun-tree? (node-left x))
           (braun-tree? (node-right x))
           (let ([l (height (node-left x))]
                 [r (height (node-right x))])
             (or (= l r) (= l (add1 r)))))))
```

The smallest counterexample breaking `insert` is a tree of height 1 with its left branch full.

```
(node (node #f #f) #f)
```

Even though execution can reach the error in theory, the frontier expands quickly, preventing graph exploration to reach the desired node in a practical amount of time. This is unintuitive at first, because the counterexample is a very small tree, and a breadth-first search should not take so long to reach a tree of that shape. However, the tool performs a bread-first search on the execution graph, not just a graph of possible trees. In particular, the distance between the root to the node where `insert`'s input is concretized as the smallest counterexample is about 300 edges. However, it takes another 400 edges for the execution to detect that `insert`'s result, which is a bigger tree, does not satisfy the Braun-tree invariant. Along the way, the algorithm also considers numerous other trees that can be reached within the given number of edges, and the frontier becomes overwhelming.

Our first attempt at solving this problem is to detect states with abstract values of high depths and slow down their exploration. However, this approach is ineffective because there can be many real counterexamples generated from high-depthed abstract values, and it is not obvious what the depth threshold should be, which is likely to vary from program to program.

Instead, we inspect the stacks to identify states that are highly likely to result in errors and speed up their exploration. When we find a state whose top-most contract checking stack frame has the positive party being a concrete component, we deem it as a suspect. We then "fast-forward" the reduction of this state by a constant number of steps or until the state is no longer a suspect. Even though this property is neither neccessary nor sufficient to result in an error, we find it highly indicative in practice. In particular, for the Braun-tree example, the frontier's size ranges from 500 to 5000, and the number of suspects stays at 1 or at most 2, and one of those is exactly the state that triggers `insert`'s violation. With this heuristic, we cut down the time of executing this program from non-terminating after 2 hours down to 2 seconds. In addition, this does not seem to be a trade off for for our corpus of benchmarks: all the intial time for other programs either stay the same or reduce.

6. 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.

6.1 First-order Symbolic Execution

Symbolic execution on first-order programs is mature and has been used to find bugs in real-world programs [Cadar et al. 2006, 2008]. Cadar et al. [Cadar et al. 2006] 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 [Cadar et al. 2008].

6.2 Counterexample-guided Abstraction Refinement

Counterexample-guided abstraction refinement has been used in model checking and dependent type inference [Rondon et al. 2008; Kobayashi et al. 2011; Zhu and Jagannathan 2013], where the inference algorithm iteratively uses a counterexample given by the solver to refine preconditions attached to functions and values. In case the algoithm fails to infer a specification, the counterexample serves as a witness to a breaking input. Our work finds higher-order counterexamples by only integrating a first-order solver, and is applicable to both typed and untyped languages. In contrast, previous works on higher-order model checking do not consider symbolic functions, and dependent type inference relies on an extension to ML.

6.3 Random Testing

Random testing is a lightweight technique for finding counterexamples to program specifications through randomly

generated inputs. QuickCheck for Haskell [Claessen and Hughes 2000] 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. Heidegger and Thiemann [2010] 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. Klein et al. [2010] also extend random testing to work on higher-order stateful programs, discovering many bugs in object-oriented programs in Racket. Seidel et al. [2014] use refinement types as generators for tests, siginificantly improving code coverage.

Compared to random testing, our approach has an important advantage of being more automated. By combining symbolic execution with an SMT solver, the tool 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 [Claessen and Hughes 2000] or require user annotations to incorporate a specific analysis collecting all literals in the program to guide input construction [Heidegger and Thiemann 2010]. Type-targeted testing [Seidel et al. 2014] is more lightweight and does not neccessitate 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. In contrast, we make use of both contracts and regular code to guide the execution to seek inputs that both satisfy pre-conditions and fail post-conditions.

6.4 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 [Rondon et al. 2008] or allow arbitrary enforcements but leave unverifiable invariants as residual run-time checks [Flanagan 2006; Knowles and Flanagan 2010; Xu 2012; Tobin-Hochstadt and Van Horn 2012]. 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 [Tobin-Hochstadt and Van Horn 2012] and can be viewed as a complement to contract verification.

7. 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 and mutable states. 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 not only guarantees the writing of correct programs but also eases the understanding of faulty programs, speeding up the development of reliable software.

References

C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. EXE: automatically generating inputs of death. CCS '06, 2006.

C. Cadar, D. Dunbar, and D. Engler. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. OSDI'08, 2008.

K. Claessen and J. 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.

S. A. Cook. Soundness and completeness of an axiom system for program verification. In *SIAM Journal of Computing*, 1978.

L. De Moura and N. 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.

C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2011.

C. Dimoulas, S. Tobin-Hochstadt, and M. Felleisen. Complete monitors for behavioral contracts. In *21st European Symposium on Programming*. Springer Berlin Heidelberg, 2012.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02: Proceedings of the seventh ACM SIGPLAN International Conference on Functional Programming*. ACM, 2002.

C. Flanagan. Hybrid type checking. In *POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2006.

M. Flatt and PLT. Reference: Racket. Technical report, PLT Inc., 2010.

J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2002.

P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. *SIGPLAN Not.*, 2005.

M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. In *POPL '10: Proceedings of the 37th annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2010.

P. Heidegger and P. Thiemann. Contract-Driven testing of JavaScript code. In *Objects, Models, Components, Patterns*. Springer Berlin Heidelberg, 2010.

R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In *FLOPS'10*. Springer, 2006.

M. Kawaguchi, P. Rondon, and R. Jhala. Dsolve: Safety verification via liquid types. In *Computer Aided Verification*. Springer Berlin Heidelberg, 2010.

C. Klein, M. Flatt, and R. 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.

K. Knowles and C. Flanagan. Hybrid type checking. *ACM Trans. Program. Lang. Syst.*, 2010.

N. Kobayashi. Model checking Higher-Order programs. *J. ACM*, 2013.

N. Kobayashi, R. Sato, and H. 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, 2011.

P. C. Nguyen, S. T. Hochstadt, and D. V. Horn. Soft contract verification. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*. ACM, 2014.

P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, 2008.

D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 1993.

E. L. Seidel, N. Vazou, and R. Jhala. Type targeted testing, 2014.

V. St-Amour, S. Tobin-Hochstadt, M. Flatt, and M. Felleisen. Typing the numeric tower. In *Practical Aspects of Declarative Languages*. Springer Berlin Heidelberg, 2012.

S. Tobin-Hochstadt and D. 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.

D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. 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.

Y. Xie and A. Aiken. Scalable error detection using boolean satisfiability. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. ACM, 2005.

D. N. Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the ACM SIGPLAN 2012 Workshop on Partial Evaluation and Program Manipulation*. ACM, 2012.

D. N. Xu, S. Peyton Jones, and S. 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.

J. Yang, P. Twohey, D. Engler, and M. Musuvathi. Using model checking to find serious file system errors. In *Sixth Symposium on Operating Systems Design and Implementation*. USENIX.

H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ML. In *Conference on Verification, Model-Checking and Abstract Interpretation*, 2013.

Subject expertise?	Passing knowledge	Expert
Relevant?	Uninteresting	Compelling
Sound?	Flawed	Sound
Accept?	No	Yes
Strength of conviction?	Ambivalent	Adamant

Points for

Points against

Questions for authors

Other notes