Higher-Order Symbolic Execution via Contracts

Sam Tobin-Hochstadt   David Van Horn

Northeastern University
{samth,dvanhorn}@ccs.neu.edu

Abstract

We present a new approach to automated reasoning about higher-order programs by extending symbolic execution to use behavioral contracts as symbolic values, thus enabling *symbolic approximation of higher-order behavior*.

Our approach is based on the idea of an *abstract* reduction semantics that gives an operational semantics to programs with both concrete and symbolic components. Symbolic components are approximated by their contract and our semantics gives an operational interpretation of contracts-as-values. The result is an executable semantics that soundly predicts program behavior, including contract failures, for all possible instantiations of symbolic components. We show that our approach scales to an expressive language of contracts including arbitrary programs embedded as predicates, dependent function contracts, and recursive contracts. Supporting this rich language of specifications leads to powerful symbolic reasoning using existing program constructs.

We then apply our approach to produce a verifier for contract correctness of components, including a sound and computable approximation to our semantics that facilitates fully automated contract verification. Our implementation is capable of verifying contracts expressed in existing programs, and of justifying contract-elimination optimizations.

Categories and Subject Descriptors   D.2.4 [*Software Engineering*]: Software/Program Verification; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms   Languages, Theory, Verification

Keywords   Higher-order contracts, symbolic execution, reduction semantics

1.   Behavioral contracts as symbolic values

Whether in the context of dynamically loaded JavaScript programs, low-level native C code, widely-distributed libraries, or simply intractably large code bases, automated reasoning tools must cope with access to only part of the program. To handle missing components, the omitted portions are often assumed to have arbitrary behavior, greatly limiting the precision and effectiveness of the tool.

Of course, programmers using external components do not make such conservative assumptions. Instead, they attach *specifications* to these components, often with dynamic enforcement. These specifications increase their ability to reason about programs that are only partially known. But reasoning solely at the level of specification can also make verification and analysis challenging as well as requiring substantial effort to write sufficient specifications.

The problem of program analysis and verification in the presence of missing *data* has been widely studied, producing many effective tools that apply *symbolic execution* to non-deterministically consider many or all possible inputs. These tools typically determine constraints on the missing data, and reason using these constraints. Since the central lesson of higher-order programming is that computation *is* data, we propose symbolic execution of higher-order programs for reasoning about systems with omitted components, taking specifications to be our constraints.

Our approach to higher-order symbolic execution therefore combines specification-based symbolic reasoning about opaque components with semantics-based concrete reasoning about available components; we characterize this technique as *specifications as values*. As specifications, we adopt higher-order behavioral software contracts [17]. Contracts have two crucial advantages for our strategy. First, they provide benefit to programmers outside of verification, since they automatically and dynamically enforce their described invariants. Because of this, modern languages such as C#, Haskell, and Racket come with rich contract libraries that programmers already use [15, 17, 22]. Rather than requiring programmers to annotate code with assertions, we leverage the large body of code that already attaches contracts at code boundaries. For example, the Racket standard library features more than 4000 uses of contracts [21]. Second, the meaning of contracts as specifications is neatly captured by

```
(define-contract list/c
  (rec/c X (or/c empty? (cons/c nat? X))))
(module opaque
  (provide
    [insert (nat? (and/c list/c sorted?)
                  -> (and/c list/c sorted?))]
    [nums list/c])))
(module insertion-sort
  (require opaque)
  (define (foldl f l b)
    (if (empty? l) b
        (foldl f (cdr l) (f (car l) b))))
  (define (sort l) (foldl insert l empty))
  (provide
    [sort
     (list/c -> (and/c list/c sorted?))]))
> (sort nums)
(• (and/c list/c sorted?))
```

Figure 1. Verification of insertion sort

their dynamic semantics. As we shall see, we are able to turn the semantics of contract systems into tools for verification of programs with contracts. Verifying contracts holds promise both for ensuring correctness and improving performance: in existing Racket code, contract checks take more than half of the running time for large computations such as rendering documentation and type checking large programs [39].

Our plan is as follows: we begin with a review of contracts in the setting of Contract PCF [12] (§2). Next, we extend Contract PCF with abstract values described by specifications, producing a core model of symbolic execution for our language of higher-order contracts, which we dub Symbolic PCF with Contracts (§3). This allows us to give non-deterministic behavior to programs in which any number of modules are omitted, represented only by their specifications; here given as contracts. We accomplish this by treating contracts as *abstract values*, with the behavior of any of their possible concrete instantiations.

Contracts as abstract values provides a rich domain of symbols, including precise specifications for abstract higher-order values. These values present new complications to soundness, addressed with a *demonic context*, a universal context for discovering blame for behavioral values (§3.5).

We then extend this core calculus to a model of programs with modules—including opaque modules whose implementation is not available—and a much richer contract language (§4), modeling the functional core of Racket [19]. We show that our symbolic execution strategy soundly scales up from Symbolic PCF to this more complex language while preserving its advantages in higher-order reasoning. More-

over, the technique of describing symbolic values with contracts becomes even more valuable in an untyped setting.

As the modular semantics is uncomputable, this verification strategy is necessarily incomplete. To address this, we apply the technique of *abstracting abstract machines* [42] to derive first an abstract machine and then a computable approximation to our semantics from our reduction system (§5). We then turn our semantics into a tool for program verification which is integrated into the Racket toolchain and IDE (§6). Users can click a button and explore the behavior of their program in the presence of opaque modules, either with a potentially non-terminating semantics, or with a computable approximation. Finally, we discuss prior work in symbolic execution, verification of specifications, and analysis of higher-order programs (§7) and conclude.

Our semantics allows us to use contracts for verification in two senses: to verify that programs do not violate their contracts, and to verify rich properties of programs by expressing them as contracts. In fact, the semantics alone is, in itself, a program verifier. The execution of a modular program which runs without contract errors on any path is a verification that the concrete portions of the program never violate their contracts, no matter the instantiation of the omitted portions. This technique is surprisingly effective, particularly in systems with many layers, each of which use contracts at their boundaries. For example, the implementation of insertion sort in figure 1 is verified to live up to its contract, which states that it always produces a sorted list. This verification works despite the omitted `insert` function, used in higher-order fashion as an argument to `foldl`.

Contributions We make the following contributions:

1. We propose *abstract reduction semantics* as a technique for higher-order symbolic execution. This is a variant of operational semantics that treats specifications as values, to enable modular reasoning about partially unknown programs.

2. We give an abstract semantics for a core typed functional language with contracts that equips symbolic values, represented as sets of contracts, with an operational interpretation. This semantics allows sound reasoning about opaque program components with rich specifications by soundly predicting program behavior for all possible instantiations of those opaque components. We then scale this semantics up to model a more realistic untyped language with modules and an expressive set of contract combinators.

3. We derive a sound and computable program analysis based on our semantics that can serve as the basis for automated program verification, optimization, and static debugging.

4. We provide a prototype implementation of an interactive verification environment based on our models that successfully verifies existing programs with contracts.

2. Contracts and Contract PCF

The basic building block of our specification system is behavioral software contracts. Originally introduced by Meyer [31], contracts are executable specifications that sit at the boundary between software components. In a first-order setting, properly assessing which component violated a contract at run-time is straightforward. Matters are complicated when higher-order values such as functions or objects are included in the language. Findler and Felleisen [17] introduced the notion of *blame* and established a semantic framework for properly assessing blame at run-time in a higher-order language, providing the theoretical basis for contract systems such as Racket's.

```
(module double
  (provide [dbl ((even? -> even?)
                 -> (even? -> even?))])
  (define dbl (λ (f) (λ (x) (f (f x))))))
> (dbl (λ (x) 7))
```
---
top-level broke the contract on dbl;
expected <even?>, *given:* 7

To illustrate, consider the program above, which consists of a module and top-level expression. Module `double` provides a `dbl` function that implements twice-iterated application, operating on functions on even numbers. The top-level expression makes use of the `dbl` function, but incorrectly—`dbl` is applied to a function that produces 7.

Contract checking and blame assignment in a higher-order program is complicated by the fact that it is not decidable in general whether the argument of `dbl` is a function from and to even numbers. Thus, higher-order contracts are pushed down into delayed lower-order checks, but care must be taken to get blame right. In our example, the top-level is blamed, and rightly so, even though `even?` witnesses the violation when `f` is applied to `x` while executing `dbl`.

2.1 Contract PCF

Dimoulas et al. [12, 13] introduce Contract PCF as a core calculus for the investigation of contracts, which we take as the starting point for our model. CPCF extends PCF [33] with contracts for base values and first-class functions. We provide a brief recap of the syntax and semantics of CPCF.

Contracts for flat values, $\mathsf{flat}(E)$, employ predicates that may use the full expressive power of CPCF. Function contracts, $C_1 \mapsto C_2$, consist of a pre-condition contract C_1 for the argument to the function and a post-condition contract C_2 for the function's result. Dependent function contracts, $C_1 \mapsto \lambda X.C_2$, bind X to the argument of the function in the post-condition contract C_2, and thus express a dependency between a function's input and result. In the remainder of the paper, we treat the non-dependent function contract $C_1 \mapsto C_2$ as shorthand for $C_1 \mapsto \lambda X.C_2$ where X is fresh.

PCF with Contracts

Types	T	$::= B \mid T \to T \mid \mathsf{con}(T)$
Base types	B	$::= \mathsf{N} \mid \mathsf{B}$
Terms	E	$::= A \mid X \mid E\,E \mid \mu X\!:\!T.E \mid \text{if } E\,E\,E$
		$\mid\ O_1(E) \mid O_2(E,E) \mid \mathsf{mon}_f^{f,f}(C,E)$
Operations	O_1	$::= \mathsf{zero?} \mid \mathsf{false?} \mid \dots$
	O_2	$::= + \mid - \mid \wedge \mid \vee \mid \dots$
Contracts	C	$::= \mathsf{flat}(E) \mid C \mapsto C \mid C \mapsto \lambda X\!:\!T.C$
Answers	A	$::= V \mid \mathcal{E}[\mathsf{blame}_f^f]$
Values	V	$::= \lambda X\!:\!T.E \mid 0 \mid 1 \mid -1 \mid \dots \mid \mathsf{tt} \mid \mathsf{ff}$
Evaluation	$\mathcal{E}$	$::= [\,] \mid \mathcal{E}\,E \mid V\,\mathcal{E} \mid O_2(\mathcal{E},E) \mid O_2(V,\mathcal{E})$
contexts		$\mid\ O_1(\mathcal{E}) \mid \text{if } \mathcal{E}\,E\,E \mid \mathsf{mon}_h^{f,g}(C,\mathcal{E})$

Semantics for PCF with Contracts $\hspace{2em} E \longmapsto E'$

$$
\begin{aligned}
\text{if } \mathsf{tt}\ E_1\ E_2 &\longmapsto E_1 \\
\text{if } \mathsf{ff}\ E_1\ E_2 &\longmapsto E_2 \\
(\lambda X\!:\!T.E)\,V &\longmapsto [V/X]E \\
\mu X\!:\!T.E &\longmapsto [\mu X\!:\!T.E/X]E \\
O(\vec{V}) &\longmapsto A \text{ if } \delta(O,\vec{V}) = A \\
\mathsf{mon}_h^{f,g}(C_1 \mapsto \lambda X\!:\!T.C_2, V) &\longmapsto \\
\lambda X\!:\!T.\mathsf{mon}_h^{f,g}(C_2, (V\ &\mathsf{mon}_h^{g,f}(C_1, X))) \\
\mathsf{mon}_h^{f,g}(\mathsf{flat}(E), V) &\longmapsto \text{ if } (E\,V)\,V\ \mathsf{blame}_h^f
\end{aligned}
$$

A contract C is attached to an expression with the monitor construct $\mathsf{mon}_h^{f,g}(C,E)$, which carries three labels: f, g, and h, denoting the names of components which may be blamed for contract failure. (An implementation would synthesize these names from the source code.) The monitor checks any interaction between the expression and its context is in accordance with the contract.

Component labels play an important role in case a contract failure is detected during contract checking. In such a case, blame is assigned with the blame_g^f construct, which states that the component named f broke its contract with g.

CPCF is equipped with a standard type system for PCF plus the addition of a contract type $\mathsf{con}(T)$, which denotes the set of contracts for values of type T [12, 13]. The type system is straightforward, so for the sake of space, we defer the details to an appendix (§A.1).

The semantics of CPCF are given as a call-by-value reduction relation on programs. One-step reduction $E \longmapsto E'$ is defined above, contextually closed over evaluation contexts $\mathcal{E}$; its reflexive transitive closure is written $E \longmapsto\!\!\!\!\rightarrow E'$.

The first five cases of the reduction relation are standard for PCF. The remaining two cases implement contract checking for function contracts and flat contracts, respectively. The monitor of a function contract on a function reduces to a function that monitors its input with reversed blame labels and monitors its output with the origi-

nal blame labels.[1] The monitor of a flat contract reduces to an if-expression which tests whether the predicate holds. If it does, the value is returned. Otherwise, a contract error is signaled with appropriate blame.

3. Symbolic PCF with Contracts

We now present an extension to Contract PCF and its semantics that enriches the language with *symbolic values*, drawn from the language of contracts. The key idea of SCPCF is to take the values of CPCF as "pre"-values U and add a notion of an unknown values (of type T), written "$\bullet^T$". Purely unknown values have arbitrary behavior, but we refine unknowns by attaching a set of contracts that specify an agreement between the program and the missing component. Such refinements can guide an operational characterization of a program. Pre-values are refined by a set of contracts to form a value, $U/\mathcal{C}$, where $\mathcal{C}$ ranges over sets of contracts.

The high-level goal of the following semantics is to enable the running of programs with unknown components. The main requirement is that the results of running such computations should soundly and precisely approximate the result of running that same program after replacing an unknown with *any* allowable value. More precisely, if a program involving some value V produces an answer A, then abstracting away that value to an unknown should produce an approximation of A:

$$\text{if } \mathcal{E}[V] \longmapsto A \text{ and } \vdash V : T, \text{ then } \mathcal{E}[\bullet^T] \longmapsto A',$$

where A' "approximates" A.

Notation: Abstract (or synonymously: symbolic) values $\widehat{V}$ range over values of the form $\bullet^T/\mathcal{C}$. Whenever the refinement of a value is irrelevant, we omit the $\mathcal{C}$ set. We write $V \cdot C$ for $U/\mathcal{C} \cup \{C\}$ where $V = U/\mathcal{C}$.

The semantics given below replace that of section 2, equipping the operational semantics with an interpretation of symbolic values. (The semantics of contract checking is deferred for the moment.)

To do so requires two changes:

1. the δ relation must be extended to interpret operations when applied to symbolic values, and

2. the one-step reduction relation must be extended to the cases of (1) branching on a (potentially) symbolic value, and (2) applying a symbolic function.

3.1 Operations on symbolic values

Typically, the interpretation of operations is defined by a function δ that maps an operation and argument values to

---

[1] For simplicity, we present the so-called *lax* dependent contract rule; our implementation uses *indy* [13], obtained by replacing the contractum with:

$$\lambda X\!:\!T.\mathrm{mon}_h^{f,g}([\mathrm{mon}_h^{f,h}(C_1, X)/X]C_2, V\ \mathrm{mon}_h^{g,f}(C_1, X)).$$

Symbolic PCF with Contracts

Prevalues	$U ::= \bullet^T \mid \lambda X\!:\!T.E \mid 0 \mid 1 \mid -1 \mid \ldots \mid \mathsf{tt} \mid \mathsf{ff}$
Values	$V ::= U/\{C, \ldots\}$

Semantics for Symbolic PCF with Contracts $E \longmapsto E'$

$$\mathsf{if}\ V\ E_1\ E_2 \longmapsto E_1 \text{ if } \delta(\mathsf{false?}, V) \ni \mathsf{ff}$$
$$\mathsf{if}\ V\ E_1\ E_2 \longmapsto E_2 \text{ if } \delta(\mathsf{false?}, V) \ni \mathsf{tt}$$
$$(\lambda X\!:\!T.E)\ V \longmapsto [V/X]E$$
$$\mu X\!:\!T.E \longmapsto [\mu X\!:\!T.E/X]E$$
$$O(\vec{V}) \longmapsto A \text{ if } \delta(O, \vec{V}) \ni A$$
$$(\bullet^{T \to T'}/\mathcal{C})\ V \longmapsto \bullet^{T'}/\{[V/X]C_2 \mid C_1 \mapsto \lambda X\!:\!T.C_2 \in \mathcal{C}\}$$
$$(\bullet^{T \to T'}/\mathcal{C})\ V \longmapsto \mathsf{havoc}_T\ V$$

---

an answer, e.g. $\delta(\mathsf{add1}, 0) = 1$. The result of applying a primitive may either be a value in case the operation is defined on its given arguments, or blame in case it is not.

The extension of δ to interpret symbolic values is largely straightforward. It starts by generalizing δ from a function from an operation and values to an answer, to a relation between operations, values, and answers (or equivalently, to a function from an operation and values to *sets* of answers). This enables multiple results when a symbolic value does not convey enough information to uniquely determine a single result. For example, $\delta(\mathsf{zero?}, 0) = \{\mathsf{tt}\}$, but $\delta(\mathsf{zero?}, \bullet^{\mathsf{N}}) = \{\mathsf{tt}, \mathsf{ff}\}$. From here, all that remains is adding appropriate clauses to the definition of δ for handling symbolic values. As an example, the definition includes:

$$\delta(+, V_1, V_2) \ni \bullet^{\mathsf{N}}, \text{ if } V_1 \text{ or } V_2 = \bullet^{\mathsf{N}}/\mathcal{C}.$$

The remaining cases are similarly straightforward.

The revised reduction relation reduces an operation, non-deterministically, to any answer in the δ relation.

3.2 Branching on symbolic values

The shift from the semantics of section 2 to section 3 also involves what appears to be a cosmetic change in the reduction of conditionals, e.g., from

$$\mathsf{if}\ \mathsf{tt}\ E_1\ E_2 \longmapsto E_1$$

to

$$\mathsf{if}\ V\ E_1\ E_2 \longmapsto E_1 \text{ if } \delta(\mathsf{false?}, V) \ni \mathsf{ff}.$$

In the absence of symbolic values, the two relations are equivalent, but once symbolic values are introduced, the latter handles branching on potentially symbolic values by deferring to δ to determine if V is possibly true. Consequently branching on $\bullet^{\mathsf{B}}$ results in both E_1 and E_2 since $\delta(\mathsf{false?}, \bullet^{\mathsf{B}}) = \{\mathsf{tt}, \mathsf{ff}\}$. Without this slight refactoring for conditionals, additional cases for the reduction relation are required, and these cases would largely mimic the existing

if reductions. By reformulating in terms of δ, we enable the uniform reduction of abstract and concrete values.

3.3 Applying symbolic functions

When applying a symbolic function V, the reduction relation must take two distinct possibilities into account. The first is that although the argument to the symbolic function escapes, no failure occurs in the unknown context, so the function returns an abstract value refined by the range contracts of the function. The second is that the use of V in a unknown context results in the blame of V. To discover if blaming V is possible we rely upon a havoc function, which iteratively explores the behavior of V for possible blame. Its only purpose is to uncover blame, thus it never produces a value; it either diverges or blames V. In this simplified model, the only behavioral values are functions, so we represent all possible uses of the escaped value by iteratively applying it to unknown values. This construction represents a universal "demonic" context to discover a way to blame V if possible, and we have named the function havoc to emphasize the analogy to Boogie's havoc function [14], which serves the same purpose, but in a first-order setting.

The havoc function is indexed by the type of its argument. At base type, values do not have behavior, so havoc simply produces a diverging computation. At function type, havoc produces a function that applies its argument to an appropriately typed unknown input value and then recursively applies havoc at the result type to the output:

$$\mathsf{havoc}_B = \mu x.x$$
$$\mathsf{havoc}_{T \to T'} = \lambda x : T \to T'.\mathsf{havoc}_{T'}(x \bullet^T)$$

This means that $\mathsf{havoc}_T \, V$ either diverges or produces blame, and thus never introduces spurious results, and further that applications of havoc_T can be given whatever type is required for the context.

To see how havoc finds all possible errors in a term, consider the following function guarded by a contract (the mon function wraps a value with a contract):

$$\mathsf{mon}(\lambda f : N \to N.\mathsf{sqrt} \, (f \, 0), (\mathsf{any} \mapsto \mathsf{any}) \mapsto \mathsf{any})$$

where any is the trivial contract $\mathsf{flat}(\lambda x.\mathsf{tt})$ and sqrt has the type $N \to N$ and contract $\mathsf{flat}(\mathsf{positive?}) \mapsto \mathsf{flat}(\mathsf{positive?})$. If we then apply havoc to this term at the appropriate type, it will supply the input $\bullet^{N \to N}$ for f. When this abstract value is applied to 0, it reduces both to $\mathsf{havoc}_N \, 0$, a diverging term that produces no blame, and the symbolic number $\bullet^N$. Finally, sqrt is applied to $\bullet^N$, which both passes and fails the contract check on sqrt, since $\bullet^N$ represents both positive and non-positive numbers; the latter demonstrates the original function could be blamed.

In contrast, if the original term was wrapped in the contract $(\mathsf{any} \mapsto \mathsf{flat}(\mathsf{positive?})) \mapsto \mathsf{any}$, then the abstract value $\bullet^{N \to N}$ would have been wrapped in the contract $\mathsf{any} \mapsto$ flat(positive?). When the wrapped abstract function is applied to 0, it then produces the more precise abstract value $\bullet^N \cdot \mathsf{flat}(\mathsf{positive?})$ as the input to sqrt and fails to blame the original function.

The ability of havoc to find blame if possible is key to our soundness result.

3.4 Contract checking symbolic values

We now turn to the revised semantics for contract checking reductions in the presence of symbolic values. The key ideas are that we

1. avoid checking any contracts which a value provably satisfies, and

2. add flat contracts to a value's refinement set whenever a contract check against that value succeeds.

To implement the first idea, we add a reduction relation that sidesteps a contract check and just produces the checked value whenever the value proves it satisfies the contract. To implement the second idea, we revise the flat contract checking reduction relation to produce not just the value, but the value refined by the contract in the success branch of a flat contract check.

Contract checking, revisited

$$\mathsf{mon}_h^{f,g}(C, V) \longmapsto V \text{ if } \vdash V : C \checkmark$$
$$\mathsf{mon}_h^{f,g}(\mathsf{flat}(E), V) \longmapsto$$
$$\text{if } (E \, V) \, (V \cdot \mathsf{flat}(E)) \, \mathsf{blame}_g^f \text{ if } \nvdash V : \mathsf{flat}(E) \checkmark$$
$$\mathsf{mon}_h^{f,g}(C_1 \mapsto \lambda X : T.C_2, V) \longmapsto$$
$$\lambda X : T.\mathsf{mon}_h^{f,g}(C_2, V \, \mathsf{mon}_h^{g,f}(C_1, X))$$
$$\text{if } \nvdash V : C_1 \mapsto \lambda X : T.C_2 \checkmark$$

The judgment $\vdash V : C \checkmark$ denotes that V provably satisfies the contract C, which we read as "V proves C." Our system is parametric with respect to this provability relation and the precision of the symbolic semantics improves as the power of the proof system increases. For concreteness, we begin by considering the following simple, yet useful proof system which asserts a symbolic value proves any contract it is refined by:

$$\frac{C \in \mathcal{C}}{\vdash V/\mathcal{C} : C \checkmark}$$

As we will see subsequently, this relation can easily be extended to handle more sophisticated reasoning.

Taken together, the revised contract checking relation and proves relation allow values to remember contracts once they are checked and to avoid rechecking in subsequent computations. Consider the following program with abstract pieces:

$$\begin{aligned}
\mathsf{let} \; \mathsf{keygen} &= \mathsf{mon}(\mathsf{unit} \mapsto \mathsf{flat}(\mathsf{prime?}), \bullet) \\
\mathsf{rsa} &= \mathsf{mon}(\mathsf{flat}(\mathsf{prime?}) \mapsto (\mathsf{any} \mapsto \mathsf{any}), \bullet) \\
\mathsf{in} \; \mathsf{rsa} \, (\mathsf{keygen} \, ()) \; \text{"Plaintext"}
\end{aligned}$$

When invoking keygen, the result is an abstract value since keygen's source is not available to be verified. However, this abstract value *remembers* the prime? contract, meaning that our semantics correctly predicts that the top level application does not break rsa's contract by providing a composite number. This verifies that regardless of the implementation of keygen and rsa, which may themselves be buggy, their *composition* is verified to uphold its obligations.

3.5 Soundness

Soundness relies on the definition of approximation between terms. We write $E \sqsubseteq E'$ to mean E' *approximates* E, or conversely E *refines* E'. The basic intuition for approximation is an abstract value, which can be thought of as standing for a set of acceptable concrete values, approximates a concrete value if that value is in the set the abstract value denotes.

Since the $\bullet^T$ value stands for any value of type T, we have the following axiom:

$$\frac{\vdash V : T}{V \sqsubseteq \bullet^T}$$

(In subsequent judgments, we assume both sides of the approximation relation are typable at the same type and thus omit type annotations and judgments.) A monitored expression is approximated by its contract:

$$\overline{\mathsf{mon}(C, E) \sqsubseteq \bullet \cdot C}$$

To handle the approximation of wrapped functions, we employ the following rule, which matches the right-hand side of the reduction relation for the monitor of a function:

$$\overline{(\lambda X.\mathsf{mon}(D, (V\ \mathsf{mon}(C, X)))) \sqsubseteq \bullet \cdot C \mapsto \lambda X.D}$$

Arbitrary additional contract refinements may be introduced on the approximated value as follows:

$$\overline{V \cdot C \sqsubseteq V}$$

A contract may be eliminated when a value proves it:

$$\frac{\vdash V : C \checkmark}{V \sqsubseteq V \cdot C}$$

If an expression approximates a monitored expression, it's OK to monitor the approximating expression too:

$$\frac{\mathsf{mon}(C, E) \sqsubseteq E'}{\mathsf{mon}(C, E) \sqsubseteq \mathsf{mon}(C, E')}$$

Finally, $\sqsubseteq$ is reflexively, transitively, and compatibly closed.

Theorem 1 (Soundness of Symbolic PCF with Contracts).
If $E \sqsubseteq E'$ and $E \longmapsto\!\!\!\rightarrow A$, then there exists some A' such that $E' \longmapsto\!\!\!\rightarrow A'$ where $A \sqsubseteq A'$.

Proof. (Sketch) The proof follows from (1) a completeness result for havoc, which states that if $\mathcal{E}[V] \longmapsto\!\!\!\rightarrow \mathcal{E}'[\mathsf{blame}^\ell]$ where ℓ is not in $\mathcal{E}$, then havoc $V \longmapsto\!\!\!\rightarrow \mathcal{E}''[\mathsf{blame}^\ell]$, and (2) the following main lemma: if $E_1 \longmapsto E_1' \neq \mathcal{E}[\mathsf{blame}_g^f]$, and $E_1 \sqsubseteq E_2$, then $E_2 \longmapsto\!\!\!\rightarrow E_2'$ and $E_1' \sqsubseteq E_2'$, which is in turn proved reasoning by cases on $E_1 \longmapsto E_1'$ and appealing to auxiliary lemmas that show approximation is preserved by substitution and primitive operations. The full proof for the enriched system of section 4 is given in section 4.5. □

The soundness result achieves the high-level goal stated at the beginning of this section: we have constructed an *abstract reduction semantics* for the sound symbolic execution of programs such that their symbolic execution approximates the behavior of programs for *all possible instantiations* of the opaque components. In particular, we can verify pieces of programs by running them with missing components, refined by contracts. If the abstract program does not blame the known components, *no context can cause those components to be blamed.*

4. Symbolic Core Racket

Having developed the core ideas of our symbolic executor for programs with contracts, we extend our language to an *untyped* core calculus of modular programs with data structures and rich contracts. This forms a core model of a realistic programming language, Racket [19]. In addition to closely modeling our target language, omitting types places a greater burden on the contract system and symbolic executor. As we see in this section, ours is up to the job.

To SCPCF we add pairs, the empty list, and related operations; contracts on pairs; recursive contracts; and conjunctive and disjunctive contracts. Predicates, as before, are expressed as arbitrary programs within the language itself. Programs are organized as a set of module definitions, which associate a module name with a value and a contract. Contracts are established at module boundaries and here express an agreement between a module and the external context. The contract checking portion of the reduction semantics monitors these agreements, maintaining sufficient information to blame the appropriate party in case a contract is broken.

4.1 Syntax

The syntax of our language is given in figure 2. We write $\vec{E}$ for a possibly-empty sequence of E, and treat these sequences as sets where convenient. Portions highlighted in gray are the key extensions over SCPCF, as presented in earlier sections.

A program P consists of a sequence of modules followed by a main expression. Modules are second-class entities that name a single value along with a contract to be applied to that value; module names are in scope throughout the program. *Opaque modules* are modules whose body is •. Expressions now include module references, labeled by

$$
\begin{array}{lll}
P, Q & ::= & \vec{M}\,E \\
M, N & ::= & (\text{module } f\ C\ V) \\
E, E' & ::= & f^\ell \mid X \mid A \mid E\ E^\ell \mid \text{if } E\ E\ E \mid O\ \vec{E}^\ell \mid \mu X.E \\
      &   & \mid\ \text{mon}_\ell^{\ell,\ell}(C, E) \\
U & ::= & n \mid \text{tt} \mid \text{ff} \mid (\lambda X.E) \mid \bullet \mid (V, V) \mid \text{empty} \\
V & ::= & U/\mathcal{C} \\
C, D & ::= & X \mid C \mapsto \lambda X.C \mid \text{flat}(E) \\
     &   & \mid\ \langle C, C \rangle \mid C \vee C \mid C \wedge C \mid \mu X.C \\
O & ::= & \text{add1} \mid \text{car} \mid \text{cdr} \mid \text{cons} \mid + \mid = \mid o? \mid \ldots \\
o? & ::= & \text{nat?} \mid \text{bool?} \mid \text{empty?} \mid \text{cons?} \mid \text{proc?} \mid \text{false?} \\
A & ::= & V \mid \mathcal{E}[\text{blame}_\ell^\ell]
\end{array}
$$

Figure 2. Syntax of Symbolic Core Racket

the module they appear in; this label is used as the negative party for the module's contract. Applications are also labeled; this label is used if the application fails. Pair values and the empty list constant are standard, along with their operations. Since the language is untyped, we add standard type predicates such as nat?.

The new contract forms include pair contracts, $\langle C, D \rangle$, conjunction, $C \wedge D$, and disjunction, $C \vee D$, of contracts, and explicit recursive contracts with contract variables, $\mu X.C$.

Contract checks $\text{mon}_h^{f,g}(C, E)$, which will now be inserted automatically by the operational semantics, take all of their labels from the names of modules, with the third label h representing the module in which the contract originally appeared. As before, f represents the positive party to the contract, blamed if the expression does not meet the contract, and g is the negative party, blamed if the context does not satisfy its obligations. Whenever these annotations can be inferred from context, we omit them; in particular, in the definition of relations, it is assumed all checks of the form $\text{mon}(C, E)$ have identical annotations. We omit labels on applications whenever they provably cannot be blamed, e.g. when the operand is known to be a function.

A blame expression, $\text{blame}_{\ell'}^\ell$, now indicates that the module (or the top-level expression) named by ℓ broke its contract with ℓ', which may be the name of a module f, the top-level expression †, or the language, indicated by Λ, in the case of primitive errors.

Syntactic requirements: We make the following assumptions of initial programs P: programs are closed, every module reference and application is labeled with the enclosing module name, or † for the top-level expression, operations are applied with the correct arity, abstract values only appear in opaque module definitions, and no monitors or blame expressions appear in source programs.

We also require that recursive contracts be *productive*, meaning either a function or pair contract constructor must occur between binding and reference. We also require that contracts in the source program are closed, both with respect

to λ-bound and contract variables. Following standard practice, we will say that a contract is *higher-order* if it syntactically contains a function contract; otherwise, the contract is *flat*. Flat contracts can be checked immediately, whereas higher-order contracts potentially require delayed checks. All predicate contracts are necessarily flat.

Disjunction of contracts: For disjunctions, we require that at most one disjunct is higher-order and without loss of generality, we assume it is the right disjunct. The reason for this restriction is that we must choose at the time of the *initial* check of the contract which disjunct to use—we cannot just try both because higher-order checks must be delayed. In Racket, disjunction is thus restricted to contracts that are distinguishable in a first-order way, which we simplify to the restriction that only one can be higher-order.

4.2 Reductions

Evaluation is modeled with one-step reduction on programs, $P \longmapsto Q$. Since the module context consists solely of syntactic values, all computation occurs by reduction of the top-level expression. Thus program steps are defined in terms of top-level expression steps, carried out in the context of several module definitions. We model this with a reduction relation on expressions in a module context, which we write $\vec{M} \vdash E \longmapsto E'$. We omit the the module context where it is not used and write $E \longmapsto E'$ instead. Our reduction system is given with evaluation contexts, which are identical to those of SCPCF in section 3.

We present the definition of this relation in several parts.

4.2.1 Applications, operations, and conditionals

First, the definition of procedure applications, conditionals, and primitive operations is as usual for a call-by-value language. Primitive operations are interpreted by a δ relation (rather than a function), just as in section 3. The reduction relation for these terms is defined as follows:

Basic reductions $\hfill E \longmapsto E'$

$$
\begin{array}{lll}
((\lambda X.E)\ V)^\ell & \longmapsto & [V/X]E \\
(V\ V')^\ell & \longmapsto & \text{blame}_\Lambda^\ell \quad \text{if } \delta(\text{proc?}, V) \ni \text{ff} \\
(O\ \vec{V})^\ell & \longmapsto & A \quad\qquad \text{if } \delta(O^\ell, \vec{V}) \ni A \\
\text{if } V\ E\ E' & \longmapsto & E \quad\qquad \text{if } \delta(\text{false?}, V) \ni \text{ff} \\
\text{if } V\ E\ E' & \longmapsto & E' \quad\qquad \text{if } \delta(\text{false?}, V) \ni \text{tt}
\end{array}
$$

Again, we rely on δ not only to interpret operations, but also to determine if a value is a procedure or ff; this allows uniform handling of abstract values, which may (depending on their remembered contracts) be treated as both true and false. We add a reduction to $\text{blame}_\Lambda^\ell$ when applications are misused; the program has here broken the contract with the language, which is no longer checked statically by the type system as it was in SCPCF. Additionally, our rules for if follow the Lisp tradition, which Racket adopts, in treating all non-ff values as true.

Primitive operations (concrete values) $\qquad \delta(O^\ell, \vec{V}) \ni A$

$$\delta(\mathsf{add1}, n) \ni n + 1$$
$$\delta(+, n, m) \ni n + m$$
$$\delta(\mathsf{car}, (V, V')) \ni V$$
$$\delta(\mathsf{cdr}, (V, V')) \ni V'$$

Primitive operations (abstract values)

$\vdash V : o?\,\checkmark$	$\implies$	$\delta(o?, V) \ni \mathsf{tt}$
$\vdash V : o?\,\boldsymbol{\times}$	$\implies$	$\delta(o?, V) \ni \mathsf{ff}$
$\vdash V : o?\,\mathbf{?}$	$\implies$	$\delta(o?, V) \ni \bullet/\{\mathsf{flat}(\mathsf{bool}?)\}$
$\vdash V : \mathsf{nat}?\,\checkmark$	$\implies$	$\delta(\mathsf{add1}, V) \ni \bullet/\{\mathsf{flat}(\mathsf{nat}?)\}$
$\vdash V : \mathsf{nat}?\,\boldsymbol{\times}$	$\implies$	$\delta(\mathsf{add1}^\ell, V) \ni \mathsf{blame}^\ell_{\mathsf{add1}}$
$\vdash V : \mathsf{nat}?\,\mathbf{?}$	$\implies$	$\delta(\mathsf{add1}, V) \ni \bullet/\mathsf{flat}(\mathsf{nat}?)$
		$\wedge\ \delta(\mathsf{add1}^\ell, V) \ni \mathsf{blame}^\ell_{\mathsf{add1}}$
$\vdash V : \mathsf{cons}?\,\checkmark$	$\implies$	$\delta(\mathsf{car}, V) \ni \pi_1(V)$
$\vdash V : \mathsf{cons}?\,\boldsymbol{\times}$	$\implies$	$\delta(\mathsf{car}^\ell, V) \ni \mathsf{blame}^\ell_{\mathsf{car}}$
$\vdash V : \mathsf{cons}?\,\mathbf{?}$	$\implies$	$\delta(\mathsf{car}, V) \ni \pi_1(V)$
		$\wedge\ \delta(\mathsf{car}^\ell, V) \ni \mathsf{blame}^\ell_{\mathsf{car}}$
otherwise		$\delta(O^\ell, \vec{V}) \ni \mathsf{blame}^\ell_\Lambda$

Figure 3. Basic operations

4.2.2 Basic operations

Basic operations, as with procedures and conditionals, follow SCPCF closely. Operations on concrete values are standard, and we present only a few selected cases. Operations on abstract values are more interesting. A few selected cases are given in figure 3 as examples. Otherwise, the definition of δ for concrete values is standard and we relegate the remainder to the auxiliary materials (§A.2).

When applying base operations to abstract values, the results are potentially complex. For example, $\mathsf{add1}\ \bullet$ might produce any natural number, or it might go wrong, depending on what value $\bullet$ represents. We represent this in δ with a combination of non-determinism, where δ relates an operation and its inputs to multiple answers, as well as abstract values as results, to handle the arbitrary natural numbers or booleans that might be produced. A representative selection of the δ definition for abstract values is presented in figure 3.

The definition of δ relies on a proof system relating predicates and values, just as with contract checking. Here, $\vdash V : o?\,\checkmark$ means that V is known to satisfy $o?$, $\vdash V : o?\,\boldsymbol{\times}$ means that V is known not to satisfy $o?$, and $\vdash V : o?\,\mathbf{?}$ means neither is known. For example, $\vdash 7 : \mathsf{nat}?\,\checkmark$, $\vdash \mathsf{tt} : \mathsf{cons}?\,\boldsymbol{\times}$, and $\vdash \bullet : o?\,\mathbf{?}$ for any $o?$. (Again, our system is parametric with respect to this proof system, although we present a useful instance in section 4.3.) Finally, if no case matches, then an appropriate error is produced.

Labels on operations come from the application site of the operation in the program, e.g., $(\mathsf{add1}\ 5)^\ell$, so that the appropriate module can be blamed when primitive operations are

misused, as in the last case, and are omitted where irrelevant. When primitive operations are misused, the violated contract is on Λ, standing for the programming language itself, just as in the rule for application of non-functions.

4.2.3 Module references

To handle references to module-bound variables, we define a module environment that describes the module context $\vec{M}$. Using the module reference annotation, the environment distinguishes between self references and external references. When an external module is referenced ($f \neq g$), its value is wrapped in a contract check; a self-reference is resolved to its (unchecked) value. This distinction implements the notion of "contracts as boundaries" [17], in other words, contracts are an agreement between the module and its context, and the module can behave internally as it likes.

Module references $\qquad\qquad \vec{M} \vdash f^g \longmapsto E$

$\vec{M} \vdash f^f \longmapsto V$	if $(\mathsf{module}\ f\ C\ V) \in \vec{M}$
$\vec{M} \vdash f^g \longmapsto \mathsf{mon}^{f,g}_f(C, V)$	if $(\mathsf{module}\ f\ C\ V) \in \vec{M}$
$\vec{M} \vdash f^g \longmapsto \mathsf{mon}^{f,g}_f(C, \bullet \cdot C)$	if $(\mathsf{module}\ f\ C\ \bullet) \in \vec{M}$

4.2.4 Contract checking

With the basic rules handled, we now turn to the heart of the system, contract checking. As in section 3, as computation is carried out, we can discover properties of values that may be useful in subsequently avoiding spurious contract errors. Our primary mechanism for remembering such discoveries is to add properties, encoded as contracts, to values as soon as the computational process proves them. If a value passes a flat contract check, we add the checked contract to the value's remembered set. Subsequent checks of the same contract are thus avoided. We divide contract checking reductions into two categories, those for flat contracts and those for higher-order contracts, and consider each in turn.

Flat contracts: First, checking flat contracts is handled by three rules, presented in figure 4, depending on whether the value has already passed the relevant contract.

The first two rules consider the case where the value definitely does pass the contract, written $\vdash V : C\,\checkmark$ ("V proves C"), or does not pass, written $\vdash V : C\,\boldsymbol{\times}$ ("V refutes C"). If neither of these is the case, written $\vdash V : C\,\mathbf{?}$, the third rule implements a contract check by compiling it to an if-expression. The test is an application of the function generated by $\mathsf{FC}(C)$ to V. If the test succeeds, $V \cdot C$ is produced. Otherwise, the positive party, here f, is blamed for breaking the contract on h.

The three judgments checking the relation between values and contracts are a simple proof system; by parameterizing over these relations, we enable our system to make use of sophisticated existing decision procedures. For the moment,

Flat contract reduction $\qquad$ $\mathsf{mon}(C, V) \longmapsto E'$

$$\mathsf{mon}_h^{f,g}(C, V) \longmapsto V \cdot C \qquad \text{if } C \text{ is flat and } \vdash V : C \checkmark$$

$$\mathsf{mon}_h^{f,g}(C, V) \longmapsto \mathsf{blame}_h^f \qquad \text{if } C \text{ is flat and } \vdash V : C \text{\ding{55}}$$

$$\mathsf{mon}_h^{f,g}(C, V) \longmapsto \text{ if } (\text{FC}(C)\ V)\ (V \cdot C)\ \mathsf{blame}_h^f$$
$$\text{if } C \text{ is flat and } \vdash V : C \textbf{ ?}$$

Flat contract checking $\qquad$ $\text{FC}(C) = E$

$$\text{FC}(\mu X. C) = \mu X. \text{FC}(C)$$
$$\text{FC}(X) = X$$
$$\text{FC}(\mathsf{flat}(E)) = E$$
$$\text{FC}(C_1 \wedge C_2) = \lambda y.\text{if } (\text{FC}(C_1)\ y)\ (\text{FC}(C_2)\ y)\ \mathsf{ff}$$
$$\text{FC}(C_1 \vee C_2) = \lambda y.\text{if } (\text{FC}(C_1)\ y)\ \mathsf{tt}\ (\text{FC}(C_2)\ y)$$
$$\text{FC}(\langle C_1, C_2 \rangle) =$$
$$\lambda y.(\mathsf{and}\ (\mathsf{cons?}\ y)\ (\text{FC}(C_1)\ (\mathsf{car}\ y))\ (\text{FC}(C_2)\ (\mathsf{cdr}\ y)))$$

Figure 4. Flat contracts

the key property is that $\vdash V \cdot C : C \checkmark$ holds, just as in section 3.4, and further details are discussed in section 4.3.

Compiling flat checks to predicates: The FC metafunction, also in figure 4, takes a flat contract and produces the source code of a function, which when applied to a value produces true or false indicating whether the value passes the contract. The additional complexity over the similar rules of sections 2 and 3 handles the addition of flat contracts containing recursive contracts, disjunctive and conjunctive contracts, and pair contracts. In particular, to check disjunctive contracts, we must *test* if the left disjunct passes the contract, and branch on the result, whereas our earlier reduction rules for flat contracts simply *fail* for contracts that don't pass.

As an example, the expression $\mathsf{mon}_h^{f,g}(\mathsf{flat}(\mathsf{nat?}), V)$ reduces to if $(\mathsf{nat?}\ V)\ V\ \mathsf{blame}_h^f$, but using this reduction to check the left disjunct of $\mathsf{mon}(\mathsf{flat}(\mathsf{nat?}) \vee \mathsf{flat}(\mathsf{bool?}), \mathsf{tt})$ would cause a blame error, which is obviously not the intended result. Instead, the rules for FC generate the check if $((\mathsf{nat?}\ \mathsf{tt}) \vee (\mathsf{bool?}\ \mathsf{tt}))\ \mathsf{tt}\ \mathsf{blame}$, which then succeeds.

Higher-order contracts: The next set of reduction rules, presented in figure 5, defines the behavior of higher-order contract checks; we assume here that the contract is not flat.

In the first rule, we again use the η-expansion technique pioneered by Findler and Felleisen [17] to decompose a higher-order contract into subcomponents. This rule only applies if the contracted value V is indeed a function, as indicated by proc? (In SCPCF, this side-condition is unnecessary thanks to the type system). Otherwise, the second rule blames the positive party of a function contract when the supplied value is not a function.

The remaining rules handle higher-order contracts that are not immediately function contracts, such as pairs of function contracts. The first two are for pair contracts. If the value is determined to be a pair by cons?, then the components are

Function contract reduction $\qquad$ $\mathsf{mon}(C, V) \longmapsto E'$

$$\mathsf{mon}_h^{f,g}(C \mapsto \lambda X. D, V) \longmapsto$$
$$(\lambda X. \mathsf{mon}_h^{f,g}(D, (V\ \mathsf{mon}_h^{g,f}(C, X))))$$
$$\text{if } \delta(\mathsf{proc?}, V) \ni \mathsf{tt}$$

$$\mathsf{mon}_h^{f,g}(C \mapsto \lambda X. D, V) \longmapsto \mathsf{blame}_h^f \quad \text{if } \delta(\mathsf{proc?}, V) \ni \mathsf{ff}$$

Other higher-order contract reductions

$$\mathsf{mon}(\langle C, D \rangle, V) \longmapsto$$
$$(\mathsf{cons}\ \mathsf{mon}(C, \mathsf{car}\ V')\ \mathsf{mon}(D, \mathsf{cdr}\ V'))$$
$$\text{if } \delta(\mathsf{cons?}, V) \ni \mathsf{tt} \text{ and } V' = V \cdot \mathsf{flat}(\mathsf{cons?})$$

$$\mathsf{mon}_h^{f,g}(\langle C, D \rangle, V) \longmapsto \mathsf{blame}_h^f \qquad \text{if } \delta(\mathsf{cons?}, V) \ni \mathsf{ff}$$

$$\mathsf{mon}(\mu X. C, V) \longmapsto \mathsf{mon}([\mu X. C / X] C, V)$$

$$\mathsf{mon}(C \wedge D, V) \longmapsto \mathsf{mon}(D, \mathsf{mon}(C, V))$$

$$\mathsf{mon}(C \vee D, V) \longmapsto \text{if } (\text{FC}(C)\ V)\ (V \cdot C)\ \mathsf{mon}(D, V)$$
$$\text{if } \vdash V : C \textbf{ ?}$$

$$\mathsf{mon}(C \vee D, V) \longmapsto V \qquad\qquad \text{if } \vdash V : C \checkmark$$

$$\mathsf{mon}(C \vee D, V) \longmapsto \mathsf{mon}(D, V) \qquad \text{if } \vdash V : C \text{\ding{55}}$$

Figure 5. Higher-order contract reduction

extracted using car and cdr and checked against the relevant portions of the contract. Otherwise, then the program reduces to blame, analogous to function contracts.

The last set of rules decompose combinations of higher-order contracts. Recursive contracts are unrolled. (Productivity ensures that contracts do not unroll forever.) Conjunctions are split into their components, with the left checked before the right. For higher-order disjunctions, we rely on the invariant that only the right disjunct is higher-order and use FC for the check of the left. When possible, we omit this check by using the proof system (see the final two rules).

4.2.5 Applying abstract values

Again, application of abstract values poses a challenge, just as it did in section 3. In contrast, the system must now explore more possible behaviors of abstract operators and it can no longer be guided by the type. Fortunately, abstract values provide the tools to express the needed computation.

Applying abstract values $\qquad$ $E \longmapsto E'$ where $\widehat{V} = \bullet / \mathcal{C}$

$$\widehat{V}\ V' \longmapsto \bullet / \{ [\widehat{V}/X] D \mid (C \mapsto \lambda X. D) \in \mathcal{C} \}$$
$$\text{if } \delta(\mathsf{proc?}, \widehat{V}) \ni \mathsf{tt}$$

$$\widehat{V}\ V' \longmapsto \mathsf{havoc}\ V' \qquad \text{if } \delta(\mathsf{proc?}, \widehat{V}) \ni \mathsf{tt}$$

$$\mathsf{havoc} = \mu y.(\lambda x. \text{AMB}(\{ y\ (x\ \bullet), y\ (\mathsf{car}\ x), y\ (\mathsf{cdr}\ x) \}))$$

$$\text{AMB}(\{ E \}) = E$$
$$\text{AMB}(\{ E, E_1, \dots \}) = \text{if } \bullet\ E\ \text{AMB}(\{ E_1, \dots \})$$

The behavior of abstract values, which are created by references to opaque modules, is handled in much the same way as in SCPCF. When an abstract function is applied, there are again two possible scenarios: (1) the abstract function returns an abstract value or (2) the argument escapes into an unknown context that causes the value to be blamed. We again make use of a havoc function for discovering if the possibility of blame exists. In contrast to the typed setting of SCPCF, we need only one such value. The demonic context is a universal context that will produce blame if it there exists a context that produces blame originating from the value. If the universal demonic context cannot produce blame, only the range value is produced.

The havoc function is implemented as a recursive function that makes a non-deterministic choice as to how to treat its argument—it either applies the argument to the least-specific value, •, or selects one component of it, and then recurs on the result of its choice. This subjects the input value to all possible behavior that a context might have. Note that the demonic context might itself be blamed; we implicitly label the expressions in the demonic context with a distinguished label and disregard these spurious errors in the proof of soundness. We use the AMB metafunction to implement the non-determinism of havoc; AMB uses an if test of an opaque value, which reduces to both branches.

4.3 Proof system

Compared to the very simple proof system of section 3.4, the system for proving or refuting whether a given value satisfies a contract in Core Racket is more sophisticated, although the general principles remain the same.

In particular, we rely on three different kinds of judgments that relate values and contracts: proves, refutes, and neither. The first, $\vdash V : C \checkmark$ includes the original judgment that a value proves a contract if it remembers that contract. Additionally, we add judgements for reasoning about type predicates in the language. For example if a value is known to satisfy a particular base predicate, written $\vdash V : o? \checkmark$, then the value satisfies the contract $\mathsf{flat}(o?)$. This relies on the relation between values and predicates used above in the definition of δ, which is defined in a straightforward way.

The refutes relation is more interesting and relies on additional semantic knowledge, such as the disjointness of data types. For instance, a value that remembers it is a procedure, refutes all pair contracts and the pair? predicate contract. Other refutes judgments are straightforward based on structural decomposition of contracts and values.

The complete definition of these relations is given in the auxiliary materials (§A.3). Our implementation, described in section 6, incorporates a richer set of rules for improved reasoning. The implementation is naive but effective for basic semantic reasoning, however it essentially does no sophisticated reasoning about base type domains such as numbers, strings, or lists. The tool could immediately benefit from leveraging an external solver to decide properties of concrete values.

4.4 Improving precision via non-determinism

Since our reduction rules, and in particular the δ relation, make use of the remembered contracts on values, making these contracts as specific as possible improves precision of the results.

Improving precision via non-determinism $\widehat{V} \longmapsto \widehat{V}'$

$$
\begin{array}{lll}
\bullet/\mathcal{C} \cup \{C_1 \vee C_2\} & \longmapsto & \bullet/\mathcal{C} \cup \{C_i\} \quad i \in \{1,2\} \\
\bullet/\mathcal{C} \cup \{\mu X.C\} & \longmapsto & \bullet/\mathcal{C} \cup \{[\mu X.C/X]C\}
\end{array}
$$

The two rules above increase the specificity of abstract values. The first rule splits abstract values known to satisfy a disjunctive contract. For example, $\bullet/\{\mathsf{flat}(\mathsf{nat}?) \vee \mathsf{flat}(\mathsf{bool}?)\} \longmapsto \bullet/\mathsf{flat}(\mathsf{nat}?)$ and $\bullet/\mathsf{flat}(\mathsf{bool}?)$. This converts the imprecision of the value into non-determinism in the reduction relation, and makes subsequent uses of δ more precise on the two resulting values. Similarly, we unfold recursive contracts in abstract values; this exposes further disjunctions to split, as with a contract for lists.

As an example of the effectiveness of this simple approach, consider the list length function:

```
(module length
  (provide [len (list/c -> nat?)])
  (define (len l)
    (if (empty? l) 0 (+ 1 (len (cdr l)))))))
```

When applied to the symbolic value

$$
\bullet \cdot \mu\mathsf{x}.(\mathsf{flat}(\mathsf{empty}?) \vee \langle \mathsf{flat}(\mathsf{nat}?),\mathsf{x}\rangle)
$$

which is the definition of $\mathsf{list/c}$, we immediately unroll and split the abstract value, meaning that we evaluate the body of len in exactly the two cases it is designed to handle, with a precise result for each. Without this splitting, the test would return both tt and ff, and the semantics would attempt to take the cdr of the empty list, even though the function will never fail on concrete inputs. This provides some of the benefits of occurrence typing [41] simply by exploiting the non-determinism inherent in the reduction semantics.

4.5 Evaluation and Soundness

We now define evaluation of entire modular programs, and prove soundness for our abstract reduction semantics. One complication remains. In any program with opaque modules, any module might be referenced, and then treated arbitrarily, by one of the opaque modules. While this does not affect the value that the main expression might reduce to, it does create the possibility of blame that has not been previously predicted. We therefore place each concrete module into the previously-defined demonic context and non-deterministically choose one of these expressions to run *prior* to running the main module of the program.

The evaluation function is defined as:

$$eval(\vec{M}\,E) = \{E' \mid \vec{M} \vdash E';\ E \longmapsto E'\},$$

where $E' = \text{AMB}(\{\text{tt}, \overline{\text{havoc } f}\})$, $(\text{module } f\ C\ V) \in \vec{M}$.

Soundness, as in section 3.5, relies on the definition of approximation between terms, and its straightforward extension to modules and programs.

The approximation relation on expressions, modules, and programs is formalized below. We show only the important cases and omit the straightforward structurally recursive cases. We parametrize $\sqsubseteq$ by the module context of the abstract program to determine the opaque modules; we omit this context where it can be inferred.

Approximates $P \sqsubseteq Q,\ M \sqsubseteq_{\vec{M}} N,$ and $E \sqsubseteq_{\vec{M}} E'$

$$\frac{}{V \sqsubseteq \bullet} \qquad \frac{}{\text{mon}(C, E) \sqsubseteq \bullet \cdot C}$$

$$\frac{}{(\lambda X.\text{mon}(D, (V\ \text{mon}(C, X)))) \sqsubseteq \bullet \cdot C \mapsto \lambda X.D}$$

$$\frac{}{V \cdot C \sqsubseteq V} \qquad \frac{V \sqsubseteq V'}{V \cdot C \sqsubseteq V' \cdot C} \qquad \frac{\vdash V : C\ \checkmark}{V \sqsubseteq V \cdot C}$$

$$\frac{\vec{N} \sqsubseteq \vec{M} \quad E' \sqsubseteq E}{\vec{N}\,E' \sqsubseteq \vec{M}\,E} \qquad \frac{\text{mon}(C, E) \sqsubseteq E'}{\text{mon}(C, E) \sqsubseteq \text{mon}(C, E')}$$

$$\frac{(\text{module } f\ C\ \bullet) \in \vec{M} \text{ or } f = \dagger}{\text{blame}_g^f \sqsubseteq_{\vec{M}} E}$$

$$\frac{(\text{module } f\ C\ \bullet) \in \vec{M}}{(\text{module } f\ C\ V) \sqsubseteq_{\vec{M}} (\text{module } f\ C\ \bullet)}$$

The $\sqsubseteq$ relation is lifted to evaluation contexts $\mathcal{E}$ by structural extension; to contracts by structural extension on contracts and $\sqsubseteq$ on embedded values; to vectors by point-wise extension; and to sets of expressions by point-wise, subset extension.

With the approximation relation in place, we now state and prove our main soundness theorem.

Theorem 2 (Soundness of Symbolic Core Racket).
If $P \sqsubseteq Q$ where $Q = \vec{M}\,E$ and $A \in eval(P)$, then there exists some $A' \in eval(Q)$ where $A \sqsubseteq_{\vec{M}} A'$.

This soundness result, proved below, implies that if a program with opaque modules does not produce blame, then the known modules cannot be blamed, *regardless* of the choice of implementation for the opaque modules.

Corollary 1. *If f is the name of a concrete module in P, and $P \not\longmapsto \mathcal{E}[\text{blame}_g^f]$, then no instantiation of the opaque modules in P can cause f to be blamed.*

To prove soundness, we first establish some auxiliary lemmas.

Lemma 1. *If $\vec{V} \sqsubseteq_{\vec{M}} \vec{U}$, then $\delta(O, \vec{V}) \sqsubseteq_{\vec{M}} \delta(O, \vec{U})$.*

Proof. By inspection of δ and cases on O and $\vec{V}$. $\square$

Lemma 2. *If $E \sqsubseteq_{\vec{M}} E'$ and $V \sqsubseteq_{\vec{M}} U$, then $[V/X]E \sqsubseteq_{\vec{M}} [U/X]E'$.*

Proof. By induction on the structure of E and cases on the derivation of $E \sqsubseteq_{\vec{M}} E'$. $\square$

Lemma 3. *If $C \sqsubseteq_{\vec{M}} D$, then $\text{FC}(C) \sqsubseteq_{\vec{M}} \text{FC}(D)$.*

Proof. By induction on the structure of C and cases on the derivation of $E \sqsubseteq_{\vec{M}} E'$ and the definition of FC. $\square$

Lemma 4. *Let $E = \text{FC}(C)$, then*

1. if $\vdash V : C\ \checkmark$ and $V \sqsubseteq_{\vec{M}} U$, then $E\ U \longmapsto A \sqsupseteq \text{tt}$,
2. if $\vdash V : C\ \textbf{\textsf{X}}$ and $V \sqsubseteq_{\vec{M}} U$, then $E\ U \longmapsto A \sqsupseteq \text{ff}$.

Proof. By induction on the structure of C and cases on the derivation of $U \sqsubseteq_{\vec{M}} V$ and the definition of FC. $\square$

Lemma 5. *If $P \longmapsto P'$, $P' \neq \vec{M}\ \text{blame}_g^f$ and $P \sqsubseteq Q$, then $Q \longmapsto Q'$ and $P' \sqsubseteq Q'$.*

Proof. We split into two cases.
Case (1):

$$P = \vec{M}\ \mathcal{E}[E] \longmapsto \vec{M}\ \mathcal{E}[E']$$
$$Q = \vec{N}\ \mathcal{E}'[E'] \longmapsto \vec{N}\ \mathcal{E}'[E'']$$

where $\mathcal{E} \sqsubseteq_{\vec{N}} \mathcal{E}'$. We reason by cases on the step from E to E'.

- Case: $E = f^g$ and $(\text{module } f\ C\ E') \in \vec{M}$
  If f is transparent in $\vec{N}$, then $(\text{module } f\ C\ E') \in \vec{N}$ and we are done by simple application of the reduction rules for module references. Otherwise, $f \neq g$ and thus

  $$E' = \text{mon}_f^{f,g}(C, V) \qquad E'' = \text{mon}_f^{f,g}(C, \bullet/\{C\}),$$

  but now $E' \sqsubseteq_{\vec{N}} E''$, since $E' \sqsubseteq_{\vec{N}} \bullet/\{C\}$.
- Case:
  $\text{mon}(C \mapsto \lambda X.D, V) \longmapsto (\lambda X.\text{mon}(D, (U\ \text{mon}(C, X))))$
  where $\delta(\text{proc?}, V) \ni \text{tt}$ and $U = V/\{C \mapsto \lambda X.D\}$.
  Since E' is a redex, by $\sqsubseteq$ we have $E' = \text{mon}(C' \mapsto \lambda y.D', V')$, where $V \sqsubseteq_{\vec{N}} V'$. Then $\delta(\text{proc?}, V') \ni \text{tt}$ by lemma 1,. So $E'' = (\lambda y.\text{mon}(D', (U'\ \text{mon}(C', y))))$ where $U' = V'/\{C' \mapsto \lambda y.C'\} \sqsubseteq_{\vec{N}} V/\{C \mapsto \lambda X.D\}$ and thus $E' \sqsubseteq_{\vec{N}} E''$.
- Case: $V_1\ V_2^\ell \longmapsto \text{blame}_\Lambda^\ell$, where $\delta(\text{proc?}, V_1) \ni \text{ff}$.
  By $\sqsubseteq$, we have $E' = U_1\ U_2^\ell$ and $U_i \sqsubseteq_{\vec{N}} V_i$. By lemma 1, $\delta(\text{proc?}, U_1) \ni \text{ff}$, hence $U_1\ U_2^\ell \longmapsto \text{blame}_\Lambda^\ell$.

- Case: $V_1\, V_2^\ell \longmapsto E'$, where $\delta(\mathsf{proc?}, V_1) \ni \mathsf{tt}$.

  By $\sqsubseteq$, we have $E' = U_1\, U_2^\ell$ and $U_i \sqsubseteq_{\vec N} V_i$. By lemma 1, $\delta(\mathsf{proc?}, U_1) \ni \mathsf{tt}$.

  Either V_1 and U_1 are structurally similar, in which case the result follows by possibly relying on lemma 2, or $V_1 = (\lambda X.E_0)/\mathcal{C}$ and $U_1 = \bullet/\mathcal{C}'$. There are two possibilities for the origin of V_1: either it was blessed or it was not. If V_1 was not blessed, then $\mathcal{C}$ contains no function contracts, implying $\mathcal{C}'$ contains no function contracts, hence $E'' = \bullet$, and the result holds. Alternatively, V_1 was blessed and $\mathcal{C}$ contains a function contract $C \mapsto \lambda X.D$. But by the blessed application rule, we have $\mathcal{E} = \mathcal{E}_1[\mathsf{mon}([V_2'/X]D, [\,])]$, thus by assumption $\mathcal{E}' = \mathcal{E}_1'[\mathsf{mon}([U_2'/X]D', [\,])]$, implying $[V_2'/X]D \sqsubseteq [U_2'/X]D'$, finally giving us the needed conclusion:

  $$\mathcal{E}_1[\mathsf{mon}([V_2'/X]D, E')] \sqsubseteq_{\vec N} \mathcal{E}_1'[\mathsf{mon}([U_2'/X]D', E'')],$$

  where $E'' = \bullet/\{[U_2/X]D' \mid C' \mapsto \lambda X.D' \in \mathcal{C}'\}$.
- Case: $\mathsf{mon}(C, V) \longmapsto E'$ where C is flat.

  If $\vdash V : C\,\textbf{?}$, then the case holds by use of lemma 3. If $\vdash V : C\,\checkmark$, then the case holds by lemma 4(1). If $\vdash V : C\,\textbf{✗}$, then the case holds by lemma 4(2).
- The remaining cases are straightforward.

Case (2):

$$P = \vec M\ \mathcal{E}_1[\mathcal{E}_2[E]] \longmapsto \vec M\ \mathcal{E}_1[\mathcal{E}_2[E']]$$
$$Q = \vec N\ \mathcal{E}_1'[\mathcal{E}_2'[E']]$$

where $\mathcal{E}_1$ is the largest context such that $\mathcal{E}_1 \sqsubseteq_{\vec N} \mathcal{E}_1'$ but $\mathcal{E}_2 \not\sqsubseteq_{\vec N} \mathcal{E}_2'$.

In this case, we have $\mathcal{E}_2[E] \sqsubseteq_{\vec N} \mathcal{E}_2'[E']$, but since $\mathcal{E}_2 \not\sqsubseteq_{\vec N} \mathcal{E}_2'$, this must follow by one of the non-structural rules for $\sqsubseteq$, all of which are either oblivious to the contents of E and E', or do not relate redexes to anything. $\qquad\square$

Lemma 6. *If there exists a context $\mathcal{E}$ such that*

$$\vec M\ (\mathsf{module}\ f\ C\ V)\ \mathcal{E}[f] \longmapsto \mathsf{blame}_g^f,$$

then

$$\vec M\ (\mathsf{module}\ f\ C\ V)\ (\mathsf{havoc}\ f) \longmapsto \mathsf{blame}_g^f.$$

Proof. If there exists such an $\mathcal{E}$, then without loss of generality, it is of some minimal form $\mathcal{D}$ in

$$\mathcal{D} = [\,] \mid (\mathcal{D}\ V) \mid (\mathsf{car}\ \mathcal{D}) \mid (\mathsf{cdr}\ \mathcal{D}),$$

but then there exists a $\mathcal{D}'$ equal to $\mathcal{D}$ with all values replaced with $\bullet$ such that $\vec M\ \mathcal{D}'[V] \longmapsto \mathsf{blame}_g^f$. This is because at every reduction step, replacing some component of the redex with $\bullet$ causes at least that reduction to fire, possibly in addition to others. Further, by inspection of havoc, if $\vec M\ \mathcal{D}'[V] \longmapsto \mathsf{blame}_g^f$, then $\vec M\ (\mathsf{havoc}\ V) \longmapsto \mathsf{blame}_g^f$. $\qquad\square$

Proof of Theorem 2. By the definition of *eval*, we have $P \longmapsto A$. Let the number of steps in $P \longmapsto A$ be n. There are two cases: either $A = V$, or $A = \mathsf{blame}_{\ell'}^\ell$. If $A = V$, then we proceed by induction on n and apply lemma 5 at each step.

If $A = \mathsf{blame}_{\ell'}^\ell$ then there are two possibilities. If ℓ is the name of an opaque module in $\vec M$ or if $\ell = \dagger$, then $A \sqsubseteq_{\vec M} A'$ immediately. If $\ell = f$ is the name of a concrete module in $\vec M$, then $\mathsf{havoc}\ f \longmapsto A$ by lemma 6, and therefore $A \in eval(Q)$ by the definition of *eval*. $\qquad\square$

5. Convergence and decidability

At this point, we have constructed an abstract reduction semantics that gives meaning to programs with opaque components. The semantics is a sound abstraction of all possible instantiations of the omitted components, thus it can be used to verify that modular programs satisfy their specifications. However, in order to automatically verify programs, the semantics must converge for the program being analyzed.

We now describe how to refactor the semantics in such a way that we can accelerate and—if desired—*guarantee* convergence by introducing further orthogonal approximation into the semantics. This is accomplished in a number of ways:

- operations may widen when applied to concrete values,
- environment structure may be bounded, and
- control structure may be bounded.

In order to guarantee convergence for all possible programs, all three of these forms of approximation must be employed and are sufficient to guarantee decidability of the semantics. However, our experience suggests that such strong convergence guarantees may be unnecessary in practice. For example, we have found that a simple widening of concrete recursive function applications to their contract when applied to abstract values works well for ensuring convergence of tail-recursive programs broken into small modules. By adding a limited form of control structure approximation, we are able to automatically verify non-tail-recursive functions. Taken together, these forms of approximation do not guarantee convergence in general, yet they do induce convergence for all of the examples we have considered (see §6.2) and with fewer spurious results compared to more traditional forms of abstraction such as 0CFA and pushdown flow analysis. But rather than advocate a particular approximation strategy, we now describe how to refactor the semantics so that all these choices may be expressed. Our implementation (§6) then makes it easy to explore any of them.

5.1 Widening values

Widening the results of basic operations, i.e., those interpreted by δ, can cut down the set of base values, and if necessary can ensure finiteness of base values. Thus, we replace δ with δ':

$$\delta'(O, \vec V) \ni \mathsf{widen}(V) \iff \delta(O, \vec V) \ni V$$

where widen represents an arbitrary choice of a metafunction for mapping a value to its approximation. To avoid approximation, it can be interpreted as the identity function. To ensure finite base values, it must map to a finite range; a simple example is $\mathsf{widen}(V) = \bullet$ for all V. An example of a more refined interpretation is $\mathsf{widen}(n) = \mathsf{flat}(\mathsf{nat?})$, $\mathsf{widen}(\mathsf{cons}\ V\ U) = \mathsf{flat}(\mathsf{cons?})$, etc. For soundness, we require that $\mathsf{widen}(V) = V'$ implies $V \sqsubseteq V'$.

5.2 Bounding environment structure

The lexical environment of a program represents a source of unbounded structure. To enable approximation of the lexical environment, we first refactor the semantics as calculus of explicit substitutions [11] with a global store. Substitutions are modeled by finite maps from variables to addresses and the store maps addresses to *sets of* values, which are now represented as closures:

$$
\begin{array}{rcl}
\rho, \varrho & ::= & \emptyset \mid \rho[X \mapsto a] \\
\sigma, \varsigma & ::= & \emptyset \mid \sigma[a \mapsto \{V, \dots\}]
\end{array}
$$

Reductions that bind variables, such as function application, must allocate and extend the environment. For example,

$$(\lambda X.E)\ V^\ell \;\longmapsto\; [V/X]E$$

becomes an analogous reduction relation on closures and stores:

$$
\begin{array}{rcl}
((\lambda X.E), \rho)\ V^\ell, \sigma & \longmapsto & (E, \rho[X \mapsto a]), \sigma \sqcup [a \mapsto V] \\
& & \text{where } a = \mathsf{alloc}(\sigma, X)
\end{array}
$$

and the interpretation of $\sigma \sqcup [a \mapsto V]$ is σ' s.t. $\sigma'(b) = \sigma(b)$ if $a \neq b$ and $\sigma'(a) = \sigma(a) \cup \{V\}$.

Since reduction operates over closures, there is an additional case needed to handle variable references:

$$(X, \rho), \sigma \;\longmapsto\; V, \sigma \text{ if } V \in \sigma(\rho(X))$$

The alloc metafunction provides a point of control which regulates the approximation of environment structure. To ensure finite environment approximation, the metafunction must map to a finite set of addresses (for a fixed program). A simple finite abstraction is $\mathsf{alloc}(\sigma, X) = 0$ for all σ and X. This abstraction maps all bindings to a single location, thus conflating all bindings in a program. Although highly imprecise, this is a sound approximation, and in fact *any* instantiation of alloc is sound [42]. A more refined abstraction is $\mathsf{alloc}(\sigma, X) = X$, which provides a finite abstraction of environment structure similar to 0CFA in which multiple bindings of the same variable are conflated. To avoid approximation, $\mathsf{alloc}(\sigma, X)$ should chose a fresh address not in σ. Consequently, the store maps all addresses to singleton sets of values and the environment-based reduction semantics corresponds precisely with the original.

5.3 Bounding control structure

The remaining source of unbounded structure stems from the control component of a program. Bounding the environment structure was achieved by (1) making substitutions explicit as environments and (2) threading environments through a store which could be bounded. An analogous approach is taken for control: (1) evaluation contexts are explicated as continuations and (2) continuations are threaded through the store.

The resulting semantics is an abstract machine that operates over a triple comprised of a closure, a store, and a continuation. Transitions take three forms: decomposition steps, which search for the next redex and push continuations if needed; plug steps which return a value to a context and pop continuations if needed; and contraction steps, which implement the s notion of reduction.

We write continuations κ as single evaluation context frames with embedded addresses representing (a pointer to) the surrounding context. So for example $E\ a$ represents $E\ \mathcal{E}$ where a points to a continuation representing $\mathcal{E}$. A simple decompose case is:

$$
\begin{array}{rcl}
\langle (E\ E', \rho), \sigma, \kappa \rangle & \longmapsto & \langle (E, \rho), \sigma \sqcup [a \mapsto \kappa], a\ (E', \rho) \rangle \\
& & \text{where } a = \mathsf{alloc}(\sigma, \kappa)
\end{array}
$$

Notice that this transition searches for the next redex in the left side of an application and allocates a pointer to the given context in order to push on the argument to be evaluated later. Similar to variable binding, continuations are allocated using alloc and joined in the store, allowing for multiple continuations to reside in a single location. The corresponding plug rule pops the current continuation frame and non-deterministically chooses a continuation:

$$\langle V, \sigma, a\ (E', \rho) \rangle \;\longmapsto\; \langle V\ (E', \rho), \sigma, \kappa \rangle \text{ where } \kappa \ni \sigma(a)$$

The contraction rule simply applies the reduction relation on explicit substitutions:

$$
\begin{array}{rcl}
\langle (E, \rho), \sigma, \kappa \rangle & \longmapsto & \langle (E', \varrho), \varsigma, \kappa \rangle \\
& \multicolumn{2}{l}{\text{if } ((E, \rho), \sigma) \longmapsto ((E', \varrho), \varsigma)}
\end{array}
$$

To ensure a finite approximation of control, alloc must map to a finite set of addresses. A simple finite abstraction is $\mathsf{alloc}(\kappa, \sigma) = 0$. A more refined finite abstraction of control is to use a frame abstraction: $\mathsf{alloc}(E\ a, \sigma) = E\ [\ ]$ and likewise for other continuation forms. To avoid approximation, $\mathsf{alloc}(\kappa, \sigma)$ should produce an address not in σ.

We have now restructured our semantics as a machine model with three distinct points of control over approximation: basic operations, environments, and control. The full definition of the machine is derived following the outline of Van Horn and Might [42]; the complete details are included in the Redex model accompanying the implementation.

We now establish the correspondence between the previous reduction semantics and the machine model when no approximation occurs. Let $\longmapsto_{CESK}$ denote the machine transition relation under the exact interpretations of widen and alloc. Let $\mathcal{U}$ be the straightforward recursive "unload" function that maps a closure and store to the closed term it represents.

Lemma 7 (Correspondence). *If $P \longmapsto Q$, then there exists ς such that $\langle P, \varnothing, \varnothing \rangle \longmapsto_{CESK} \varsigma$ and $\mathcal{U}(\varsigma) = Q$.*

We now relate any approximating variant of the machine to its exact counterpart. Let $\longmapsto_{\widehat{CESK}}$ denote the machine transition under any sound interpretation of widen. We define an *abstraction map* as a structural abstraction of the state-space of the exact machine to its approximate counterpart. The key case is on stores:

$$\alpha(\sigma) = \lambda\hat{a}.\bigsqcup_{\alpha(a)=\hat{a}}\{\alpha(\sigma(a))\}$$

The $\sqsubseteq$ relation is lifted to machine states as the point-wise, element-wise, component-wise, and member-wise lifting.

Theorem 3 (Soundness). *If $\varsigma \longmapsto_{CESK} \varsigma'$ and $\alpha(\varsigma) \sqsubseteq \hat{\varsigma}$, then there exists $\hat{\varsigma}'$ such that $\hat{\varsigma} \longmapsto_{\widehat{CESK}} \hat{\varsigma}'$ and $\alpha(\varsigma') \sqsubseteq \hat{\varsigma}'$.*

We have now established any instantiation of the machine is a sound approximation to the exact machine, which in turn correspond to the original reduction semantics. Furthermore, we can prove decidability of the semantics for finite instantiations of widen and alloc:

Theorem 4 (Decidability). *If widen and alloc have finite range for a program P, then $\langle P, \varnothing, \varnothing \rangle \longmapsto_{\widehat{CESK}} \varsigma$ is decidable for any ς.*

The proofs of these theorems closely follow those given by Van Horn and Might [42].

6. Implementation

To validate our approach, we have implemented a prototype interactive program verification environment, as seen in figure 6. We can take the example from section 2, define the relevant modules, and explore the behavior of different choices for the main expression.

Programs are written with the `#lang var <options>` header, where `<options>` range over a visualization mode: `trace`, `step`, or `eval`; a model mode: `term` or `machine`; and an approximation mode: `approx` or `exact`. Following the header, programs are written in a subset of Racket, consisting of a series of module definitions and a top-level expression.

The visualization mode controls how the state space is explored. The choices are simply running the program to completion with a read-eval-print loop, visualizing a directed graph of the state space labeled by transitions, or with an interactive step-by-step exploration. The model mode selects
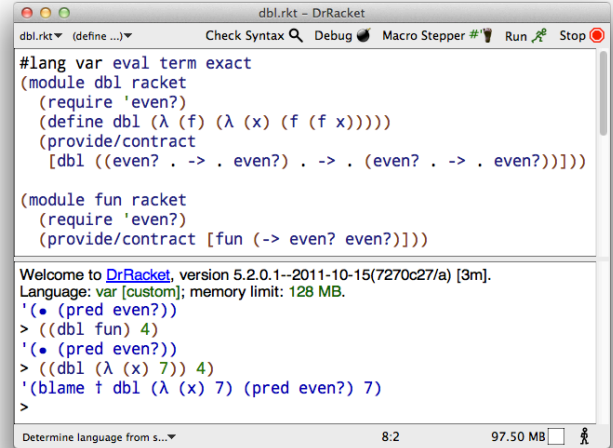
Figure 6. Interactive program verification environment

whether to use the `term` or the `machine` as the underlying model of computation.

Finally, the approximation mode selects what, if any, approximation should be used. The `exact` mode uses no approximation; allocation always returns fresh addresses and no widening is used for base values. The `approx` mode uses a default mode of approximation that has proved useful in verifying programs (discussed below).

6.1 Implementation extensions

Our prototype includes significant extensions to the system as described above.

First, we make numerous extensions in order to verify existing Racket programs. For example, modules are extended to include multiple definitions and functions may accept zero or more arguments. The latter complicates the reduction relation as new possibilities arise for errors due to arity mismatches. Second, we add additional base values and operations to the model to support more realistic programs. Third, we make the implementation of contract checking and reduction more sophisticated, improving running time and simplifying visualization. Fourth, we implement several techniques to reduce the size of the state space explored in practice, including abstract garbage collection [32]. Abstract GC enables naive allocation strategies to perform with high precision. Additionally, we widen contracted recursive functions to their contracts on recursive calls; this implements a form of induction that is highly effective at increasing convergence. Fifth, we add simpler rules to model non-recursive functions and non-dependent contracts. This brings the model closer to programmers expectation of the semantics of the language, and simplifies visualizations. Sixth, we include several more contract combinators, such as `and/c` for contract conjunction; `atom/c` for expressing equality with atomic values; `one-of/c` for

finite enumerations; `struct/c` for structures; and `listof` and `non-empty-listof` for lists of values..

Finally, we provide richer blame information, as can be seen in the screen shot of figure 6. Our system reports the full complement of information available in Racket's production contract library, which reports the failure of `dbl` as:

```
> ((dbl (λ (x) 7)) 4)
top-level broke (even? → even?) →
                 (even? → even?)
on dbl; expected <even?>, given: 7
```

Our prototype is available at `github.com/samth/var` .

6.2 Verified examples

We have verified a number of example programs, which fall into three categories:

- small programs with rich contracts such as the example of `insertion-sort` from the introduction, where verifying contract correctness is close to full functional verification;

- tricky-to-verify programs with simple contracts such as Wright and Cartwright's tautology checker, where our tool proves not only that the function satisfies its `boolean? -> boolean?` contract, but also that it has no internal run-time type errors; and

- several larger graphical, interactive video games developed according to the program-by-design [16] approach that stresses data- and contract-driven program design.

In the third category, we were able to automatically verify contract correctness for non-trivial existing programs, including two, Snake and Tetris, developed for our undergraduate programming course. The programs make use of higher-order functions such as folds and maps and construct anonymous (upward) functions. Their contracts include finite enumerations, structures, and recursive (ad-hoc) unions. Here is the key contract definition for the Snake game:

```
(define-contract snake/c
  (struct/c snake
    (one-of/c 'up 'down 'left 'right)
    (non-empty-listof
      (struct/c posn nat? nat?))))
```

We are able to automatically verify these stated invariants, such as that snakes have at least one segment and that it moves in one of four possible directions.

For these examples, we found that simply widening the results of recursive calls with abstract arguments is sufficient to ensure convergence in the semantics given an appropriate fine-grained module decomposition. All of our verified examples are available with our implementation.

7. Related work

The analysis and verification of programs and specifications has been a research topic for half a century; we survey only closely related work here.

Symbolic execution Symbolic execution [24] is the idea of running a program, but with abstract inputs. The technique can be used either for testing, to avoid the need to specify certain test data, or for verification and analysis. Over the past 35 years, it has been used for numerous testing and verification tasks. There has been a particular upsurge in interest in the last ten years [8, 9], as high-performance SAT and SMT solvers have made it possible to eliminate infeasible paths by checking large sets of constraints.

Most approaches to symbolic execution focus on abstracting first order data such as numbers, typically with constraints such as inequalities on the values. In this paper, we present an approach to symbolic execution based on contracts as symbols, which scales straightforwardly to higher-order values. Despite this focus on higher-order values, the remembered contracts maintained by our system let us constrain symbolic execution to feasible evaluations; using an external solver to decide relationships such as $\vdash V : C \checkmark$ is an important area of future work.

Recently, under the heading of *concolic execution* [20, 36], symbolic execution has been paired with test generation to analyze software more effectively. We believe that we could effectively use our system as the framework for such a system, by nondeterminstically reducing abstract values to concrete instances.

The only work on higher-order symbolic execution that we are aware of is by Thiemann [40] on eliminating redundant pattern matches. Thiemann considers only a very restricted form of symbols: named functions partially applied to arguments, constructors, and a top value. This approximation is only sound for a purely functional language, and thus while we could incorporate it into our current symbolic model of Racket, further extensions to handle mutable state rule out the technique. It is unclear whether redundancy elimination would benefit from contracts as symbol.

Verification of first-order contracts Over the past ten years, there has been enormous success verifying modular first-order programs, as demonstrated by tools such as the SLAM and Spec# projects [3, 6, 14]. These tools typically operate by abstracting first-order programs in languages such as C to simpler systems such as automata or boolean programs, then model-checking the results for violations of specified contracts.

However, these approaches do not attempt to handle the higher-order features of languages such as Racket, Python, Scala, and Haskell. For instance, the boolean program abstraction employed by SLAM [2] is inherently first-order: variables can take only boolean values. Our system, in contrast, scales to higher-order language features.

Despite the fundamental difference, there are important similarities between this work and ours. The systems all employ nondeterminism extensively to reason about unknown behavior, and abstract the environment by allowing it to take arbitrary actions; as in the `havoc` statement in Boogie [14] which we generalize to the `havoc` function for placing higher-order functions in an arbitrary context.

Additionally, we believe that the techniques used in these existing first-order tools could improve precision for first-order predicate checks in our system; exploring this is an important avenue for future work.

Verification of higher-order contracts The most closely related work to ours is the modular set-based analysis based on contracts of Meunier et al. [29, 30] and the static contract checking of Xu et al. [43, 44].

Meunier et al. take a program analysis approach, generating set constraints describing the flow of values through the program text. When solved, the analysis maps source labels to sets of abstract values to which that expression may evaluate. Meunier's system is more limited than ours in several significant ways.

First, the set-based analysis is defined as a separate semantics, which must be manually proved to correspond to the concrete semantics. This proof requires substantial support from the reduction semantics, making it significantly and artificially more complex by carrying additional information used only in the proof. Despite this, the system is unsound, since it lacks an analogue of `havoc`. This unsoundness has been verified in Meunier's prototype.

Second, while our semantics allows the programmer to choose how much to make opaque and how much to make concrete, Meunier's system always treats the entire rest of the program opaquely from the perspective of each module.

Third, our language of contracts is much more expressive: we consider disjunction and conjunction of contracts, dependent function contracts, and data structure contracts. Our ability to statically reason about contract checks is significantly greater—Meunier's system includes only the simplest of our rules for $\vdash V : C\checkmark$.

Finally, Meunier approximates conditionals by the union of its branches' approximation; the test is ignored. This seemingly minor point becomes significant when considering predicate contracts. Since predicate contracts reduce to conditionals, this effectively approximates all predicates as both holding and not holding, and thus *all predicate contracts may both fail and succeed*.

Xu et al. [44] describe a static contract verification system for Haskell. Their approach is to compile contract checks into the program, using a transformation modeled on Findler and Felleisen [17], simplify the program using the GHC optimizer, and examine the result to see if any contract checks are left in the residual program. In subsequent work, Xu [43] applies this approach to OCaml, providing a formal account of the simplifier employed, and extend-

ing simplification by using an SMT solver as an oracle for some simplification steps. In both systems, if not all contract checks are eliminated by simplification, the system reports them as potentially failing.

Our approach extends that of Xu et al. in three crucial ways. First, our symbolic execution-based approach allows us to consider full executions of programs, rather than just a simplification step. Second, Xu et al. considers a significantly restricted contract language, omitting conjunction, disjunction, and recursive contracts, as well as contracts that may not terminate, may fail, or include calls to unknown functions. As we saw in section 4, these extensions add significant complexity and expressiveness to the system. Third, as with Meunier et al.'s work, the user has no control over what is precisely analyzed; indeed, Xu et al. *inline* all non-contracted functions.

Blume and McAllester [7] provide a semantic model of contracts which includes a definition of when a term is **Safe**, which is when it can never be caused to produce blame. We use a related technique to verify that modules cannot be blamed, by constructing the `havoc` context. However, we do not attempt to construct a semantic model of contracts; instead we merely approximate the run-time behaviors of programs with contracts.

Abstract interpretation Abstract interpretation provides a general theory of semantic approximation [10] that relates concrete semantics to an abstract semantics that interprets programs over a domain of abstract values. Our approach is very much an instance of abstract interpretation. The reachable state semantics of CPCF is our concrete semantics, with the semantics of SCPCF as an abstract interpretation defined over the union of concrete values and abstract values represented as sets of contracts. In a first-order setting, contracts have been used as abstract values [14]. Our work applies this idea to behavioral contracts and higher-order programs.

Combining expressions with specifications Giving semantics to programs combined with specifications has a long history in the setting of program refinements [23]. Our key innovations are (a) treating specifications as abstract values, rather than as programs in a more abstract language, (b) applying abstract reduction to modular program analysis, as opposed to program derivation or by-hand verification, and (c) the use of higher-order contracts as specifications.

Type inference and checking can be recast as a reduction semantics [27], and doing so bears a conceptual resemblance to our contracts-as-values reduction. The principal difference is that Kuan et al. are concerned with producing a *type*, and so all expressions are reduced to types before being combined with other types. Instead, we are concerned with *values*, and thus contracts are maintained as specification values, but concrete values are not abstracted away.

Also related to our specification-as-values notion of reduction is Reppy's [34] variant of 0CFA that uses "a more refined representation of approximate values", namely types.

The analysis is modular in the sense that all module imports are approximated by their type, whereas our approach allows more refined analysis whenever components are not opaque. Reppy's analysis can be considered as an instance of our framework by applying the techniques of section 5 and thus could be derived from the semantics of the language rather than requiring custom design.

Modular program analysis Shivers [38], Serrano [37], and Ashley and Dybvig [1] address modularity (in the sense of open-world assumptions of missing program components) by incorporating a notion of an *external* or *undefined* value, which is analogous to always using the abstract value • for unknown modules, and therefore allowing more descriptive contracts can be seen as a refinement of the abstraction on missing program components.

Another sense of the words *modular* and *compositional* is that program components can be analyzed in isolation and whole programs can be analyzed by combining these component-wise analysis results. Flanagan [18] presents a set-based analysis in this style for analyzing untyped programs, with many similar goals to ours, but without considering specifications and requiring the whole program before the final analysis is available. Banerjee and Jensen [4, 5] and Lee et al. [28] take similar approaches to type-based and 0CFA-style analyses, respectively.

Other approaches to higher-order verification Kobayashi et al. [25, 26] have recently proposed approaches to verification of temporal properties of higher-order programs based on model checking. This work differs from ours in four important respects. First, it addresses temporal properties while we focus on behavioral properties. Second, it uses externally-provided specifications, whereas we use contracts, which programmers already add to their programs. Third, and most importantly, our system handles opaque components, while model-checking approaches are whole-program. Fourth, it operates on higher-order recursion schemes, a computational model with less power than CPCF, the basis of our development.

Rondon et al. [35] present Liquid Types, an extension to the type system of OCaml which incorporates dependent refinement types, and automatically discharges the obligations using a solver. This naturally supports the encoding of some uses of contracts, but restricts the language of refinements to make proof obligations decidable. We believe that a combination of our semantics with an extension to use such a solver to decide the $\vdash V : C \checkmark$ relation would increase the precision and effectiveness of our system.

8. Conclusion

We have presented a technique for verifying modular higher-order programs with behavioral software contracts. Contracts are a powerful specification mechanism that are already used in existing languages. We have shown that by us-

ing contracts as abstract values that approximate the behavior of omitted components, a reduction semantics for contracts becomes a verification system. Further, we can scale this system both to a rich contract language, allowing expressive specifications, as well as to a computable approximation for automatic verification derived directly from our semantics. Our central lesson is that abstract reduction semantics can turn the semantics of a higher-order programming language with executable specifications into a symbolic executor and modular verifier for those specifications.

References

[1] J. Michael Ashley and R. Kent Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM Trans. on Program. Lang. Syst.*, 20(4):845–868, 1998.

[2] Thomas Ball, Rupak Majumdar, Todd Millstein, and Sriram K. Rajamani. Automatic predicate abstraction of C programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 203–213, 2001.

[3] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. A decade of software model checking with SLAM. *Commun. ACM*, 54(7):68–76, 2011.

[4] Anindya Banerjee. A modular, polyvariant and type-based closure analysis. In *Proceedings of the International Conference on Functional Programming*, pages 1–10, 1997.

[5] Anindya Banerjee and Thomas Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathematical. Structures in Comp. Sci.*, 13(1):87–124, 2003.

[6] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Commun. ACM*, 54(6):81–91, 2010.

[7] Matthias Blume and David McAllester. Sound and complete models of contracts. *J. Funct. Program.*, 16(4–5):375–414, 2006.

[8] Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. EXE: automatically generating inputs of death. In *Proceedings of the Conference on Computer and Communications Security*, pages 322–335, 2006.

[9] Cristian Cadar, Daniel Dunbar, and Dawson Engler. KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs. In *Proceedings of the Conference on Operating Systems Design and Implementation*, pages 209–224, 2008.

[10] 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 Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[11] P. L. Curien. An abstract framework for environment machines. *Theoretical Computer Science*, 82(2):389–402, 1991.

[12] Christos Dimoulas and Matthias Felleisen. On contract satisfaction in a higher-order world. *ACM Trans. on Program. Lang. Syst.*, 33, 2011.

[13] Christos Dimoulas, Robert B. Findler, Cormac Flanagan, and Matthias Felleisen. Correct blame for contracts: no more scapegoating. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 215–226, 2011.

[14] 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*, pages 10–30, 2011.

[15] Manuel Fähndrich, Michael Barnett, and Francesco Logozzo. Embedded contract languages. In *Proceedings of the Symposium on Applied Computing*, pages 2103–2110, 2010.

[16] Matthias Felleisen, Robert B. Findler, Matthew Flatt, and Shriram Krishnamurthi. *How to design programs: an introduction to programming and computing*. MIT Press, 2001.

[17] 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*, pages 48–59, 2002.

[18] Cormac Flanagan. *Effective Static Debugging via Componential Set-Based Analysis*. PhD thesis, Rice University, 1997.

[19] Matthew Flatt and PLT. Reference: Racket. Technical Report PLT-TR-2010-1, PLT Inc., 2010.

[20] 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*, pages 213–223. ACM, 2005.

[21] M. Greenberg. personal communication.

[22] Ralf Hinze, Johan Jeuring, and Andres Löh. Typed contracts for functional programming. In *Functional and Logic Programming*, volume 3945 of *LNCS*, chapter 15, pages 208–225. 2006.

[23] Ralph Johan, Abo Akademi, and J. Von Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.

[24] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, 1976.

[25] Naoki Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. In *Proceedings of the Symposium on Principles of Programming Languages*, pages 416–428, 2009.

[26] Naoki Kobayashi, Ryosuke Sato, and Hiroshi Unno. Predicate abstraction and CEGAR for higher-order model checking. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 222–233, 2011.

[27] George Kuan, David MacQueen, and Robert B. Findler. A rewriting semantics for type inference. In *Proceedings of the European Symposium on Programming*, volume 4421, 2007.

[28] Oukseh Lee, Kwangkeun Yi, and Yunheung Paek. A proof method for the correctness of modularized 0CFA. *Info. Proc. Letters*, 81:179–185, 2002.

[29] Philippe Meunier. *Modular Set-Based Analysis from Contracts*. PhD thesis, Northeastern University, 2006.

[30] 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*, pages 218–231, 2006.

[31] Bertrand Meyer. *Eiffel : The Language*. Prentice Hall, 1991.

[32] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the International Conference on Functional Programming*, pages 13–25, 2006.

[33] G. Plotkin. LCF considered as a programming language. *Theoretical Computer Science*, 5(3):223–255, 1977.

[34] John Reppy. Type-sensitive control-flow analysis. In *ML '06: Proceedings of the 2006 Workshop on ML*, pages 74–83, 2006.

[35] Patrick M. Rondon, Ming Kawaguci, and Ranjit Jhala. Liquid types. In *Programming Languages Design and Implementation*, PLDI '08, pages 159–169. ACM, 2008.

[36] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: a concolic unit testing engine for C. *SIGSOFT Softw. Eng. Notes*, 30(5):263–272, 2005.

[37] Manuel Serrano. Control flow analysis: a functional languages compilation paradigm. In *Proceedings of the Symposium on Applied Computing*, pages 118–122, 1995.

[38] Olin Shivers. *Control-flow analysis of higher-order languages*. PhD thesis, Carnegie Mellon University, 1991.

[39] T. Stephen Strickland, Sam Tobin-Hochstadt, Robert Bruce Findler, and Matthew Flatt. Chaperones and impersonators: Runtime support for reasonable interposition. In *OOPSLA '12: Object-Oriented Programming, Systems, Languages, and Applications*, 2012.

[40] Peter Thiemann. Higher-Order redundancy elimination. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*, pages 73–84, 1994.

[41] Sam Tobin-Hochstadt and Matthias Felleisen. Logical types for untyped languages. In *Proceedings of the International Conference on Functional Programming*, pages 117–128, 2010.

[42] David Van Horn and Matthew Might. Abstracting abstract machines. In *Proceedings of the International Conference on Functional Programming*, pages 51–62, 2010.

[43] Dana N. Xu. Hybrid contract checking via symbolic simplification. In *Proceedings of the Workshop on Partial Evaluation and Program Manipulation*, pages 107–116, 2012.

[44] 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*, pages 41–52, 2009.