Abstract Reduction Semantics for Modular Higher-Order Contract Verification

Sam Tobin-Hochstadt David Van Horn

Abstract

We contribute a new approach to the modular verification of higher-order programs that leverages behavioral software contracts as a rich source of symbolic values. Our approach is based on the idea of an *abstract* reduction semantics that gives meaning to programs with missing or opaque components. Such components are approximated by their contract and our semantics gives an operational interpretation of contracts-as-values. The result is a executable semantics that soundly approximates all possible instantiations of opaque components, including contract failures. It enables automated reasoning tools that can verify the contract correctness of components for all possible contexts. We show that our approach scales to an expressive language of contracts including arbitrary programs embedded as predicates, dependent function contracts, and recursive contracts. We argue that handling such a feature-rich language of specifications leads to powerful symbolic reasoning that utilizes existing program assertions. Finally, we derive a sound and computable approximation to our semantics that facilitates fully automated contract verification.

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. However, programmers who use these components do not make such conservative assumptions. Instead, they attach *specifications* to these components. These specifications increase our 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.

To tackle these problems, we combine specification-based symbolic reasoning about opaque components with semantics-based concrete reasoning about available components. Our approach to modular program verification is based on computing with *specifications as values*. As specifications, we adopt higher-order behavioral software contracts. 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 which programmers already use [9, 15, 17]. 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 [16]. Second, the meaning of contracts as specifications is neatly captured by their dynamic semantics. As we shall see, we are able to leverage the semantics of contract systems into tools for verification of programs with contracts.

Our plan is as follows: we give a reduction semantics for the core of a higher-order programming language that includes modules and contracts (§4). Next, we take a symbolic execution approach to making our semantics *modular* (§5). This allows us to give non-deterministic behavior to programs in which any number of the component 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.

Symbolic execution and refinement calculi have a long history of semantics with abstract elements; 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, which we address with a *demonic context*, a universal context for discovering blame for behavioral values (§6).

We note that this semantics 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 immediately allows us to use contracts for verification in two senses: to verify that programs do not violate their contracts, and verifying rich properties of programs by expressing them as contracts. This technique is surprisingly effective, particularly in systems with many layers, each of which use contracts at their boundaries. For example, the following tail-recursive implementation of insertion sort is verified to live up to its contract, which states that it always produces a sorted list.

As the modular semantics is uncomputable, this verification strategy is necessarily incomplete. To address this, we apply the technique of *abstracting abstract machines* [34] to derive first an abstract machine and then a computable approximation to our semantics directly from our reduction system (§7).

Finally, we turn our semantics into a tool for program verification which is integrated into the Racket [15] toolchain and IDE (§8). Users can click a button and explore the behavior of their program in the presence of opaque modules, either with a non-deterministic and uncomputable semantics, or with a computable approximation.

```
(define-contract list/c
  (rec/c X (or/c empty? (cons/c nat? X))))
(module insert (nat? (and/c list/c (pred sorted?))
                 -> (and/c list/c (pred sorted?)))
  •)
(module insertion-sort
  (list/c (and/c list/c sorted?)
       -> (and/c list/c sorted?))
  (λ (l acc)
     (if (empty? l) acc
         (insertion-sort (cdr l)
                         (insert (car l) acc)))))
(module l list/c •)
(insertion-sort l empty)
```

Contributions

- We propose *abstract reduction semantics*, a variant of operational semantics that treats specifications as values, to enable modular reasoning about partial programs.
- We give a semantics for a representative core of an untyped higher-order language with a rich language of contracts.
- We give an abstract semantics that equips symbolic values represented as sets of contracts with an operational interpretation, allowing reasoning about opaque program components.
- We prove the semantics soundly predicts program behavior for all possible instantiations of opaque components.
- We derive a sound and computable program analysis based on the abstract reduction semantics that can serve as the basis for automated program verification, optimization, and debugging.
- We provide a prototype implementation of an interactive verification environment based on our theoretical models.

We begin by giving background on the key technical ideas we employ (§2) and then give a whirlwind tour of the technical development (§3), which presents the essence of our approach to contract verification before we delve into the full development.

2. Our key technical tools

In this section, we provide background on contracts, and introduce our key techniques and design choices.

Contracts The basic building block of our specification system is behavioral software contracts. Originally introduced by Meyer [26], 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 [12] introduce the notion of *blame* and establish 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 [15].

```
(module dbl (even? -> even?) -> (even? -> even?)
  (λ (x) (λ (f) (f (f x)))))
```

top-level broke the contract on `dbl`*; expected* `<even?>`*, given:* `7`

To illustrate, consider the `dbl` program above, which consists of a module and top-level expression. Module `dbl` 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 more complex since it is not decidable whether the argument of `dbl` is a function that consumes and produces even numbers. 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`.

Semantics as a verification tool Muchnick and Jones [28] describe program analysis as "a tool for discovering properties the run-time behavior of a program without actually running it." Of course, actually running the program is the best predictor of its behavior. We instead run programs to verify them, but with components omitted. While uncomputable in general, this is effective for proving properties of programs relative to opaque components, and supports abstraction to recover computability when needed. Put simply, we make it possible to "actually run" modular programs, producing a sound prediction of their behavior.

Abstract values and nondeterminism Our verification strategy is aimed at programs which are partially opaque, so we must determine the operational behavior of opaque values. We choose to make them into abstract values, which are pervasively non-deterministic in their behavior. An example is that a test of an abstract boolean might take the then *or* the else branch of an `if` expression.

Universal contexts To verify a module will not error and produce blame in the presence of opaque components requires quantifying over all possible components to check the module. Instead, we make use of the nondeterminism described above to create a *universal context*, one that will cause a term to error if possible.

Run, don't analyze The theme of the above two ideas is to perform verification by execution, whether it be in a particular context or with particular inputs. Throughout our development, we avoid analyses of programs in favor of running them. This us leads to use δ to implement tests in `if` expressions and contract checking, and to run modules in a universal context to check them for errors.

3. A whirlwind tour

In this section, we illustrate the kernel of our approach by slimming the technical development to its bare essentials. We start by giving a semantics for a core language with functions and contracts.

$$\begin{array}{lcl}
E, F & ::= & x \mid A \mid (E\ E) \mid (\texttt{if}\ E\ E\ E) \mid (o\ E) \mid (C \Leftarrow^{\ell,\ell} E) \\
U, V & ::= & n \mid \texttt{\#t} \mid \texttt{\#f} \mid (\lambda_x x.E) \mid ((C \dashrightarrow C) \Leftarrow^{\ell,\ell} V) \\
C, D & ::= & C \rightarrow C \mid \lfloor \lambda_x x.E \rfloor \\
o & ::= & \texttt{proc?} \mid \texttt{false?} \mid \texttt{add1} \mid \ldots \\
A & ::= & V \mid \texttt{blame}^{\ell}
\end{array}$$

Our language includes usual elements such as constants and recursive functions, as well as a contract check form $(C \Leftarrow E)$, that checks E produces a value satisfying C. Contract checks represent an agreement between and expression and its context: the expression must *produce* a value meeting the specification C, while the context must *use* the value only according to C. The parties of the agreement are named by labels on the check, $(C \Leftarrow^{\ell,\ell'} E)$; ℓ refers to the expression and ℓ' to the context. A contract system enforces the agreement and *blames* the offending party when a check fails.

Contracts are written either as *predicates*, $\lfloor \lambda_y x.E \rfloor$, which are arbitrary functions, or as *function contracts*, constructed from a pair of domain and range contracts. A value satisfies a predicate contract only if the predicate holds on the value. Predicates import the full computational power of the language into the specification language of contracts. Function contracts specify behavioral properties of function values. A function satisfies a $C \rightarrow D$ contract if given values satisfying C, it produces values satisfying D. Function contracts may be thought of as specifying pre- and post-conditions. However, in a higher-order setting it is impossible to verify pre- and post-conditions against a function value. Instead, checks are delayed until the function is applied, which pushes pre- and post-conditions down to lower levels until only first-order properties, encoded as predicates, can be checked. The mechanism for delaying pre- and post-condition checking is the "blessed" function value, written $((C \dashrightarrow D) \Leftarrow V)$, which represents a value that has been partially checked against $C \rightarrow D$; in particular V has been checked for being a function (and not a number or boolean).

We formalize the semantics as a reduction relation, $\mathbf{c}$:

Basic reductions $E\ \mathbf{c}\ F$

$$\begin{array}{rcll}
((\lambda_y x.E)\ V) & \mathbf{c} & [(\lambda_y x.E)/y][V/x]E & \\
(o\ V) & \mathbf{c} & A & \text{if } \delta(o, V) \ni A \\
(\texttt{if}\ V\ E\ F) & \mathbf{c} & E & \text{if } \delta(\texttt{false?}, V) \ni \texttt{\#f} \\
(\texttt{if}\ V\ E\ F) & \mathbf{c} & F & \text{if } \delta(\texttt{false?}, V) \ni \texttt{\#t}
\end{array}$$

The first case implements function application via substitution. The second case uses a δ relation to interpret primitives. The third and fourth case handle conditional branching in the usual way, but rely on δ to determine if the test value is true or false. We follow LISP tradition and treat all non-`#f` values as true. (For this simple model, we assume erroneous programs, such as `(5 7)`, get stuck.) To implement contract monitoring, we add the following:

Contract checking $\hfill E\ \mathbf{c}\ F$

$$\begin{array}{rcl}
(\lfloor E\rfloor \Leftarrow^{\ell,\ell'} V) & \mathbf{c} & (\texttt{if}\ (E\ V)\ V\ \texttt{blame}^{\ell}) \\
(C \rightarrow D \Leftarrow^{\ell,\ell'} V) & \mathbf{c} & ((C \dashrightarrow D) \Leftarrow^{\ell,\ell'} V) \\
 & & \text{if } \delta(\texttt{proc?}, V) \ni \texttt{#t} \\
(((C \dashrightarrow D) \Leftarrow^{\ell,\ell'} V)\ U) & \mathbf{c} & (D \Leftarrow^{\ell,\ell'} (V\ (C \Leftarrow^{\ell',\ell} U)))
\end{array}$$

Checking a predicate is implemented by applying the predicate and producing the value when it holds and blaming ℓ otherwise. Checking a function against a function contract produces a blessed function. When a blessed function is applied, the delayed contract checking is resumed by checking the pre-condition against the argument, applying the function, and checking the post-condition against the result. It is critical for proper blame assignment that the check annotations are swapped when checking the argument.

All of these rules rely heavily on the δ metafunction, instead of using syntactic tests on, e.g., values tested in `if` expressions. This, along with the choice to make δ map to *sets* of values, sets up our subsequent development, where δ is non-deterministic.

At this point, we have a standard semantics for a simple, yet representative core of a higher-order language with contracts. We now aim to construct an extension of the semantics that can give meaning to programs that are *missing components*. The key idea is "holes" in the program, written $\bullet$, represent unknown, missing components. Purely unknown values have arbitrary behavior, but we will refine holes 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 program *even in the presence of the unknown*. So for example, if we know $\bullet$ satisfies $\lfloor\texttt{nat?}\rfloor \rightarrow \lfloor\texttt{nat?}\rfloor$ in $(\bullet\ \texttt{5})$, we conclude the application produces an unknown value satisfying $\lfloor\texttt{nat?}\rfloor$.

We extend values to include the opaque value $\bullet$ and values refined by a set of contracts, written $V/\mathcal{C}$. To give an operational semantics to *abstract values*, which are values $\widehat{V}$ of the form $\bullet/\mathcal{C}$, we need two things: (1) the δ relation must be extended to interpret operations when applied to (partially) unknown values and (2) we need to extend $\mathbf{c}$ to the case of applying an abstract function, i.e. an opaque value that may represent a function. The extension to δ is straightforward—when an operation is applied to an abstract value, the result set may include multiple distinct values, and/or blame.

Applying abstract values $\hfill E\ \widehat{\mathbf{c}}\ F$

$$\begin{array}{rcll}
(\widehat{V}\ U) & \widehat{\mathbf{c}} & \bullet/\{D \mid C \rightarrow D \in \mathcal{C}\} & \text{if } \delta(\texttt{proc?}, \widehat{V}) \ni \texttt{#t} \\
(\widehat{V}\ U) & \widehat{\mathbf{c}} & ((\lambda_y x.(y\ (x\ \bullet)))\ U) & \text{if } \delta(\texttt{proc?}, \widehat{V}) \ni \texttt{#t}
\end{array}$$

When applying an abstract function, $\widehat{\mathbf{c}}$ relates the term to two possible results. The first is an abstract value refined by the range contracts of the function. The second produces no result, but recursively applies the argument U. While the first possibility represents a successful function application, the second simulates the argument escaping to an unknown context. 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 $\bullet$. This construction represents a universal *demonic* context that will discover a way to blame U if possible. Its only purpose is to uncover blame.

We also revise the contract checking reductions so that values *remember which contracts they have satisfied*. A subsequent check of a value against a contract it is known to satisfy always passes, thus the semantics becomes more precise as it reduces.

Contract checking $\hfill E\ \widehat{\mathbf{c}}\ F$

$$\begin{array}{rcl}
(C \Leftarrow^{\ell,\ell'} V/\mathcal{C}) & \widehat{\mathbf{c}} & V/\mathcal{C} \text{ if } C \in \mathcal{C} \\
(\lfloor E\rfloor \Leftarrow^{\ell,\ell'} V/\mathcal{C}) & \widehat{\mathbf{c}} & (\texttt{if}\ (E\ V/\mathcal{C})\ V'\ \texttt{blame}^{\ell}) \text{ if } \lfloor E\rfloor \notin \mathcal{C} \\
 & & \text{where } V' = V/\mathcal{C} \cup \{\lfloor E\rfloor\} \\
(C \rightarrow D \Leftarrow^{\ell,\ell'} V) & \widehat{\mathbf{c}} & ((C \dashrightarrow D) \Leftarrow^{\ell,\ell'} V') \text{ if } \delta(\texttt{proc?}, V) \ni \texttt{#t} \\
 & & \text{where } V' = V/\{C \rightarrow D\}
\end{array}$$

We have now constructed an *abstract reduction semantics* that 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.*

In the remainder of the paper, we scale these ideas up to an expressive language of modules and contracts.

4. Semantics of modules and contracts

Having seen the crucial ideas, we now present the semantics our language with modules and a rich language of contracts in full[1]

To our language with first-class, higher-order recursive procedures, conditionals, base values and operations we add pairs, and to our language of contracts we add dependent higher-order contracts, contracts on pairs, recursive contracts, and the conjunction and disjunction of 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 not upheld.

4.1 Syntax

The syntax of our language is given below. We write $\boldsymbol{E}$ for a possibly-empty sequence of E, and treat these sequences as sets where convenient. Portions highlighted in gray are internal to the semantics and cannot appear in source programs. Applications are labeled by the module in which they appear.

$$\begin{array}{lcl}
P, Q & ::= & \boldsymbol{M}\,E \\
M, N & ::= & (\texttt{module}\ f\ C\ V) \\
E, F & ::= & x \mid f^{\ell} \mid A \mid (E\ E)^{\ell} \mid (\texttt{if}\ E\ E\ E) \mid (o\ \boldsymbol{E})^{\ell} \\
 & \mid & (C \Leftarrow^{f,f}_{f} E) \\
U, V & ::= & n \mid \texttt{#t} \mid \texttt{#f} \mid (\lambda_x x.E) \mid (V, V) \mid \texttt{empty} \\
 & \mid & ((C \dashrightarrow x.C) \Leftarrow^{f,f}_{f} V) \\
C, D & ::= & x \mid C \rightarrow x.C \mid \lfloor \lambda_x x.E \rfloor \mid \langle C,C \rangle \\
 & \mid & C \wedge C \mid C \vee C \mid \mu x.C \\
o & ::= & \texttt{add1} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{+} \mid \texttt{=} \mid \texttt{cons} \mid o? \\
o? & ::= & \texttt{nat?} \mid \texttt{bool?} \mid \texttt{empty?} \mid \texttt{cons?} \mid \texttt{proc?} \mid \texttt{false?} \\
A & ::= & V \mid \texttt{blame}^{\ell}_{\ell}
\end{array}$$

Contract checks, written $(C \Leftarrow^{f,g}_{h} E)$, check that E evaluates to a value satisfying C. The additional label h represents the module in which the contract originally appeared. Tracking this third label provides two benefits. First, it allows the semantics to report the

[1] Throughout, we assume a basic familiarity with reduction semantics and abstract machines and refer the reader to *Semantics Engineering with PLT Redex* [11] for background, notation, and terminology.

source of contract that was violated in the case of an error; high-quality error reporting is a feature of production contract system implementations. Second, it supports the *indy* semantics of dependent contracts, as proposed by Dimoulas et al. [7]. 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 occurrences of checks of the form $(C \Leftarrow E)$ have the same annotations. We omit labels on applications whenever they provably cannot be blamed, e.g. when the operand is known to be a function.

We use the following syntactic shorthands throughout: the non-dependent function contract $(\lambda x.E)$ is short for $(\lambda_z x.E)$ where z is not free in E, $\lfloor f^g \rfloor$ is short for $\lfloor \lambda x.(f^g\ x)^g \rfloor$, and likewise, $\lfloor o? \rfloor$ is short for $\lfloor \lambda x.(o?\ x) \rfloor$, $E; F$ is short for $((\lambda x.E)\ F)$ where x is not free in E, $C \rightarrow D$ is short for $C \rightarrow x.D$ where x is not free in D, and $((C \dashrightarrow D) \Leftarrow V)$ is short for $((C \dashrightarrow x.D) \Leftarrow V)$ where x is not free in D.

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

Syntactic requirements We make the following assumptions of initial, well-formed programs, P: programs are closed, every module reference and application is labeled with the enclosing module's name, or † if in the top-level expression, recursive contracts are productive, operations are applied with the correct arity, abstract values only appear in opaque module definitions.

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. For disjunctions, we require at most one of the disjuncts is higher-order and without loss of generality, we assume it is the right disjunct.

4.2 Reductions

Computation is modeled as a small-step reduction relation on programs, $P \longmapsto_{\mathbf{c}} Q$. Evaluation is defined as the set of reachable program states using the reflexive, transitive closure of the step relation:

$$eval_{\mathbf{c}}(P) = \{Q \mid P \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{c}} Q\},$$

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 such steps by two notions of reduction:

- The $\mathbf{c}$ reduction relation carries out procedure applications, conditionals, primitive operations, and contract checking. Procedure application, conditionals and primitive operations are defined as usual for a call-by-value language. Primitive operations are interpreted by a δ relation. Reductions for contract checking take two forms: those for checking a value against a flat contract and those for checking higher-order contracts. For flat contracts, we generate programs that decide if the given value is satisfies the contract. Higher-order contracts, on the other hand, are decomposed.
- The $\Delta(\boldsymbol{M})$ relation, which is a function of the module context of a program, resolves module references to module definitions. Self references are resolved to their unchecked definitions, while external references are wrapped in contract checks.

Together, the two model the reduction steps of a top-level expression, when closed over evaluation contexts, $\mathcal{E}$:

Reduction in context $\quad P \longmapsto_{\mathbf{c}} Q$

$$\begin{array}{rcll}
\boldsymbol{M}\ \mathcal{E}[E] & \longmapsto_{\mathbf{c}} & \boldsymbol{M}\ \mathcal{E}[E] & \text{if } E\ \mathbf{c}\ F \\
\boldsymbol{M}\ \mathcal{E}[E] & \longmapsto_{\mathbf{c}} & \boldsymbol{M}\ \mathcal{E}[F] & \text{if } (E, F) \in \Delta(\boldsymbol{M}) \\
\boldsymbol{M}\ \mathcal{E}[\texttt{blame}^f_h] & \longmapsto_{\mathbf{c}} & \boldsymbol{M}\ \texttt{blame}^f_h & \text{if } \mathcal{E} \neq [\,]
\end{array}$$

$$\mathcal{E} ::= [\,] \mid (\mathcal{E}\ E) \mid (V\ \mathcal{E}) \mid (\texttt{if}\ \mathcal{E}\ E\ E) \mid (o\ \boldsymbol{V}\ \mathcal{E}\ \boldsymbol{E}) \mid (C \Leftarrow \mathcal{E})$$

4.3 Applications, operations, and conditionals

The $\mathbf{c}$ relation is defined on closed expressions. For applications, operations, and conditionals, reduction is defined as follows:

Basic reductions $\quad E\ \mathbf{c}\ F$

$$\begin{array}{rcll}
((\lambda_y x.E)\ V)^{\ell} & \mathbf{c} & [(\lambda_y x.E)/y][V/x]E & \\
(V\ U)^{\ell} & \mathbf{c} & \texttt{blame}^{\ell}_{\Lambda} & \text{if } \delta(\texttt{proc?}, V) \ni \texttt{\#f} \\
(o\ \boldsymbol{V}) & \mathbf{c} & A & \text{if } \delta(o, \boldsymbol{V}) \ni A \\
(\texttt{if}\ V\ E\ F) & \mathbf{c} & E & \text{if } \delta(\texttt{false?}, V) \ni \texttt{\#f} \\
(\texttt{if}\ V\ E\ F) & \mathbf{c} & F & \text{if } \delta(\texttt{false?}, V) \ni \texttt{\#t}
\end{array}$$

Again, we rely on δ not only to interpret operations, but also to determine if a value is a procedure or #f; this is to set up our subsequent abstract reduction which will re-use the $\mathbf{c}$ relation by simply extending the δ relation to interpret abstract values. We add a reduction to $\texttt{blame}^f_{\Lambda}$ when applications are misused; the program has here broken the contract with the language.

4.4 Contract checking

We divide contract checking reductions into two categories. First, checking flat contracts is handled by a single rule, which delegates to the FC metafunction.

Flat contract reduction $\quad E\ \mathbf{c}\ F$

$$\begin{array}{rcll}
(C \Leftarrow^{f,g}_{h} V) & \mathbf{c} & (\texttt{if}\ (E\ V)\ V\ \texttt{blame}^f_h) & (*) \\
 & & \text{where } C \text{ is flat and } E = \textsc{fc}(C) &
\end{array}$$

This rule implements a contract check by compiling it to an `if` expression. The test is an application of the function generated by $\textsc{fc}(C)$ to V. If the test succeeds, V is produced, otherwise, the positive party to the contract, here f is blamed, noting that the original contract came from h.

The FC metafunction, defined below, 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 rule of section 3 is to handle the addition of flat contracts containing conjunction, disjunction, and pair contracts. As an example, the check expression $(\lfloor \texttt{nat?} \rfloor \Leftarrow^{f,g}_{h} V)$ reduces to $(\texttt{if}\ ((\lambda x.(\texttt{nat?}\ x))\ V)\ V\ \texttt{blame}^f_h)$

Flat contract checking $\quad \textsc{fc}(C) = V$

$$\begin{array}{rll}
\textsc{fc}(\mu x.C) = & \lambda_x y.E \text{ where } E = \textsc{fc}(C) & \\
\textsc{fc}(x) = & \lambda y.(x\ y) & \\
\textsc{fc}(\lfloor \lambda_y z.E \rfloor) = & \lambda_y z.E & \\
\textsc{fc}(C_1 \wedge C_2) = & \lambda y.(\texttt{and}\ (E_1\ y)\ (E_2\ y)) & \text{where } E_i = \textsc{fc}(C_i) \\
\textsc{fc}(C_1 \vee C_2) = & \lambda y.(\texttt{or}\ (E_1\ y)\ (E_2\ y)) & \text{where } E_i = \textsc{fc}(C_i) \\
\textsc{fc}(\langle C_1, C_2 \rangle) = & \lambda y.(\texttt{and}\ (\texttt{cons?}\ y)\ (E_1\ (\texttt{car}\ y))\ (E_2\ (\texttt{cdr}\ y))) & \\
 & \text{where } E_i = \textsc{fc}(C_i) &
\end{array}$$

The next set of reduction rules defines the behavior of higher-order contract checks; we assume in each case that the checked contract is not flat.

Reduction for function contracts $E\ \mathbf{c}\ F$

$$\begin{array}{rcll}
(((C \dashrightarrow x.D) \Leftarrow^{f,g}_{h} V)\ U) & \mathbf{c} & ([U/x]D \Leftarrow^{f,g}_{h} (V\ (C \Leftarrow^{g,f}_{h} U))) & \\
(C \rightarrow x.D \Leftarrow V) & \mathbf{c} & ((C \dashrightarrow x.D) \Leftarrow V) & (*) \\
 & & \text{if } \delta(\texttt{proc?}, V) \ni \texttt{\#t} & \\
(C \rightarrow x.D \Leftarrow^{f,g}_{h} V) & \mathbf{c} & \texttt{blame}^{f}_{h} \text{ if } \delta(\texttt{proc?}, V) \ni \texttt{\#f} &
\end{array}$$

The first rule creates *blessed functions*. These values represent a function which has been wrapped with a function contract, but where the domain and range contracts have not yet been applied to the arguments or results of the function. In the first, we apply a blessed function, producing a function application of the encapsulated function, where the argument is checked against the domain contract and the result is checked against the range. Note that the argument is substituted into the range contract to support dependent contracts. In the second rule, a value that is a function as determined by the `proc?` primitive is wrapped to produce a blessed function. The third rule blames the positive party of a function contract when the supplied value is not in fact a function.

Higher-order pair contract reductions $E\ \mathbf{c}\ F$

$$\begin{array}{rcl}
(\langle C,D\rangle \Leftarrow V) & \mathbf{c} & (\texttt{cons}\ (C \Leftarrow (\texttt{car}\ V))\ (D \Leftarrow (\texttt{cdr}\ V)))\ (*) \\
 & & \text{if } \delta(\texttt{cons?}, V) \ni \texttt{\#t} \\
(\langle C,D\rangle \Leftarrow^{f,g}_{h} V) & \mathbf{c} & \texttt{blame}^{f}_{h} \text{ if } \delta(\texttt{cons?}, V) \ni \texttt{\#f}
\end{array}$$

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 extracted using `car` and `cdr` and checked against the relevant portions of the contract. If the value is not a pair, then the program reduces to blame, analogous to the case for function contracts.

Other higher-order contract reductions $E\ \mathbf{c}\ F$

$$\begin{array}{rcll}
(\mu x.C \Leftarrow V) & \mathbf{c} & ([\mu x.C/x]C \Leftarrow V) & \\
(C \wedge D \Leftarrow V) & \mathbf{c} & (D \Leftarrow (C \Leftarrow V)) & \\
(C \vee D \Leftarrow V) & \mathbf{c} & (\texttt{if}\ (E\ V)\ V\ (D \Leftarrow V)) & (*) \\
 & & \text{where } E = \textsc{fc}(C) &
\end{array}$$

The last three rules decompose combinations of higher-order contracts. Recursive contracts are unrolled and conjunctions are decomposed into successive checks. For higher-order contract disjunctions, we make use of the invariant that only the right disjunct is higher-order and use FC to implement the check for the left.

4.5 Module references

To handle references to module-bound variables, we define a module environment that describes the module context $\boldsymbol{M}$. Using the module reference annotation, the environment distinguishes between self references and external references. When an external module is referenced, 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" [12], in other words, contracts are an agreement between the module and its context, and the module can behave internally as it likes.

Module environment $(f^g, E) \in \Delta(\boldsymbol{M})$

$$\begin{array}{rl}
\Delta(\boldsymbol{M}) = \{(f^f, V) & \mid (\texttt{module}\ f\ C\ V) \in \boldsymbol{M}\} \\
\cup\ \{(f^g, (C \Leftarrow^{f,g}_{f} V)) & \mid (\texttt{module}\ f\ C\ V) \in \boldsymbol{M}, f \neq g\}
\end{array}$$

4.6 Basic operations

Typically, the interpretation of operations is defined by a function δ that maps an operation and argument values to a result. So for example, you might have $\delta(\texttt{add1}, 0) = 1$. 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. We do the same with a slight twist: we choose to model δ more generally as a *relation* between an operation, arguments, and a result. The example now becomes $(\texttt{add1}^{\ell}, 0, 1) \in \delta$, which we also write $\delta(\texttt{add1}^{\ell}, 0) \ni 1$ to be suggestive of the standard notation. Additionally, we define here only the $\widetilde{\delta}$ relation which is a subset of the full δ relation; the remainder, $\widehat{\delta}$ handles abstract values, see section 5.5. A few selected cases are given below as examples. Otherwise, the definition of $\widetilde{\delta}$ is standard and we relegate the remainder to an appendix.

Primitive operations $\widetilde{\delta}(o^{\ell}, \boldsymbol{V}) \ni A$

$$\begin{array}{rcl@{\qquad}rcl}
\widetilde{\delta}(\texttt{add1}, n) & \ni & n+1 & \widetilde{\delta}(\texttt{+}, n, m) & \ni & n+m \\
\widetilde{\delta}(\texttt{car}, (U,V)) & \ni & U & & \ldots & \\
\widetilde{\delta}(\texttt{cdr}, (U,V)) & \ni & V & \widetilde{\delta}(o^{\ell}, \boldsymbol{V}) & \ni & \texttt{blame}^{\ell}_{\Lambda}
\end{array}$$

Labels on operations come from the application site of the operation in the program, e.g. $(\texttt{add1}\ 5)^{\ell}$ so that the appropriate module can be blamed when primitive operations are misused, as in the last case, and are omitted whenever they are 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.

5. Contracts as abstract values

The previous section establishes as semantics for programs with modules and contracts. We now extend the semantics to incorporate opaque components, i.e. modules whose implementations are omitted, written $(\texttt{module}\ f\ C\ \bullet)$. Our semantics gives non-deterministic behavior to these components, bounded by their specifications, that is, their declared contracts.

$$V, U \quad += \quad \bullet \mid V/\mathcal{C}$$

To implement this idea, we add two new possibilities for values to our language. The first is an abstract value, written $\bullet$—this is a value about which we know nothing. The second is a value which we know to have satisfied a set of contracts, written $V/\mathcal{C}$. This knowledge about values is necessary for precise reasoning about abstract values—once we know that a particular abstract value satisfies $\lfloor\texttt{even?}\rfloor$, that contract will not fail in future when reapplied to the same value.

In the following, we assume that $(V/\mathcal{C})/\mathcal{C}' = V/\mathcal{C} \cup \mathcal{C}'$ and $V = V/\emptyset$. Here, $\mathcal{C}$ ranges over sets of contracts. Without loss of generality, we assume all conjunctions in $\mathcal{C}$ are flattened into the set. We more generally refer to an *abstract value* for a value of the form $\bullet/\mathcal{C}$, while a *concrete value* is any value $V/\mathcal{C}$ where $V \neq \bullet$. We let $\widehat{V}$ range over abstract values, $\widetilde{V}$ over concrete values, and V over their union.

5.1 Opaque module references

Abstract values are introduced by reference to modules whose implementation is not available, in which case we model the missing component by its specification. A module whose implementation is not available is *opaque* and *transparent* when it is available. References to module-defined variables are now resolved through the $\widehat{\Delta}(\boldsymbol{M})$ relation, which resolves references to transparent modules are handled just as before, and resolves opaque modules to the check of an abstract value that consists solely of the module's contract.

Module environment $(f^g, E) \in \widehat{\Delta}(\boldsymbol{M})$

$$\begin{array}{ll}
\widehat{\Delta}(\boldsymbol{M}) = \{(f^f, \widetilde{V}) & \mid (\texttt{module}\ f\ C\ \widetilde{V}) \in \boldsymbol{M}\} \\
\quad \cup \{(f^g, (C \Leftarrow^{f,g}_{f} \widetilde{V})) & \mid (\texttt{module}\ f\ C\ \widetilde{V}) \in \boldsymbol{M}, f \neq g\} \\
\quad \cup \{(f^g, (C \Leftarrow^{f,g}_{f} \bullet/\{C\})) & \mid (\texttt{module}\ f\ C\ \bullet) \in \boldsymbol{M}\}
\end{array}$$

5.2 Remembering contracts

As computation is carried out, we can discover properties of abstract values that may be useful in subsequently avoiding spurious program execution. Our primary mechanism for remembering such discoveries is to add properties, encoded as contracts, to values (both abstract and concrete) as soon as the computational process proves them. So for example, if a value passes a flat contract check, we want to add the checked contract to the value's remembered set. Subsequent checks of the same contract will be avoided.

To accomplish this, we modify the rules that check first-order properties of values to remember these properties. In particular, we replace the rules in section 4 marked with $(*)$ with the following:

Remembering contracts $E\ \widehat{\mathbf{c}}\ F$

$$\begin{array}{rcl}
(C \Leftarrow^{f,g}_{h} V) & \widehat{\mathbf{c}} & (\texttt{if}\ (E\ V)\ U\ \texttt{blame}^{f}_{h}) \\
\multicolumn{3}{c}{\text{where } C \text{ is flat, } E = \textsc{fc}(C) \text{ and } U = V/\{C\} \text{ and } V \vdash^{\sim} C} \\
(C \Leftarrow^{f,g}_{h} V) & \widehat{\mathbf{c}} & V/\{C\} \text{ where } C \text{ is flat, and } V \vdash C \\
(C \Leftarrow^{f,g}_{h} V) & \widehat{\mathbf{c}} & \texttt{blame}^{f}_{h} \text{ where } C \text{ is flat, and } V \nvdash C \\
(C \rightarrow x.D \Leftarrow V) & \widehat{\mathbf{c}} & ((C \dashrightarrow x.D) \Leftarrow U) \\
\multicolumn{3}{r}{\text{where } \delta(\texttt{proc?}, V) \ni \texttt{\#t} \text{ and } U = V/\{C \rightarrow x.D\}} \\
(\langle C,D \rangle \Leftarrow V) & \widehat{\mathbf{c}} & (\texttt{cons}\ (C \Leftarrow (\texttt{car}\ U))\ (D \Leftarrow (\texttt{cdr}\ U))) \\
\multicolumn{3}{r}{\text{where } \delta(\texttt{cons?}, V) \ni \texttt{\#t} \text{ and } U = V/\{\lfloor \texttt{cons?} \rfloor\}} \\
(C \vee D \Leftarrow V) & \widehat{\mathbf{c}} & (\texttt{if}\ (E\ V)\ U\ (D \Leftarrow V)) \\
\multicolumn{3}{r}{\text{where } E = \textsc{fc}(C) \text{ and } U = V/\{C\} \text{ and } V \vdash^{\sim} C} \\
(C \vee D \Leftarrow V) & \widehat{\mathbf{c}} & V/\{C\} \text{ where } C \text{ is flat and } V \vdash C \\
(C \vee D \Leftarrow V) & \widehat{\mathbf{c}} & (D \Leftarrow V) \text{ where } C \text{ is flat and } V \nvdash C
\end{array}$$

Each of these rules is similar to its earlier counterpart except that it replaces V with U, a new value that extends the set of remembered contracts on V. We then use the remembered contracts to optimize the reductions for flat contract checking by adding a proof system, written $V \vdash C$, when a value V definitely satisfies the contract C. We provide the full details of the this proof system in section 5.5; for the moment, the key property is that $V/C \vdash C$ holds. Similarly, $V \nvdash C$ indicates that V definitely does not satisfy C, and $V \vdash^{\sim} C$ indicates that neither of these relations holds.

5.3 Reduction with abstract values

Having created abstract values via reference to opaque modules, we must determine how they behave in computation. Fortunately, in many cases our use of δ in the definition of reduction accomplishes this automatically. For example, if $\delta(\texttt{false?}, \widehat{V}) \ni \texttt{\#t}$, then $(\texttt{if}\ \widehat{V}\ E\ F) \longmapsto_{\mathbf{c}} F$—no additional rules are required.

Function application, however, is not interpreted by δ. We therefore endow abstract functions, i.e. abstract values that answer `#t` to `proc?`, with reductions that describe their behavior when applied.

When an abstract value is applied as function $(\widehat{V}\ U)$, the argument U crosses into an unknown component. It may be treated arbitrary, so long as the component lives up to any commitments its made on the domain of its inputs. So for example, applying the function $\widehat{V} = \bullet/\{\lfloor \texttt{cons?} \rfloor \rightarrow \lfloor \texttt{nat?} \rfloor\}$, $\widehat{V}$ may treat its input U arbitrary, so long as it always treats it as a pair.

There are three possibilities that may occur in when applying an abstract function: (1) the abstract function may produce a result satisfying the specification of the functions output, (2) the function may make use of its argument according to the specification of the functions input, but if the input contains functions, this potentially uncovers blame, or (3) the function errors internally or diverges. The third case we ignore, since we do not attempt to predict the behavior of components whose implementations we do not have.

We handle the other two possibilities by non-deterministically considering both. In the case of (1), we simply produce an abstract value that remembers all range contracts of the function. We handle (2) by first placing the argument in a *demonic context*, then returning the same value as in case (1). 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 successful range value is produced.

Applying abstract values $E\ \widehat{\mathbf{c}}\ F$

$$\begin{array}{lll}
(\bullet/\mathcal{C}\ U)\ \widehat{\mathbf{c}}\ \bullet/\{[V/x]D \mid C \rightarrow x.D \in \mathcal{C}\} & \text{if } \delta(\texttt{proc?}, \bullet/\mathcal{C}) \ni \texttt{\#t} \\
(\widehat{V}\ U)\ \widehat{\mathbf{c}}\ (\texttt{DEMONIC}\ U) & \text{if } \delta(\texttt{proc?}, \widehat{V}) \ni \texttt{\#t}
\end{array}$$

$$\texttt{DEMONIC} = (\lambda_y x.\textsc{amb}(\{(y\ (x\ \bullet)), (y\ (\texttt{car}\ x)), (y\ (\texttt{cdr}\ x))\}))$$

$$\begin{array}{rcl}
\textsc{amb}(\{E\}) & = & E \\
\textsc{amb}(\{E\} \uplus \mathcal{E}) & = & (\texttt{if}\ \bullet\ E\ F) \text{ where } F = \textsc{amb}(\mathcal{E})
\end{array}$$

The demonic context 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, $\bullet$, 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 `demonic`; AMB uses an `if` test of an opaque value, which reduces to both branches.

5.4 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 $E\ \widehat{\mathbf{c}}\ F$

$$\begin{array}{rcll}
\bullet/\mathcal{C} \uplus \{C_1 \vee C_2\} & \widehat{\mathbf{c}} & \bullet/\mathcal{C} \cup \{C_i\} & i \in \{1,2\} \\
\bullet/\mathcal{C} \uplus \{\mu x.C\} & \widehat{\mathbf{c}} & \bullet/\mathcal{C} \cup \{[\mu x.C/x]C\} &
\end{array}$$

These two rules increase the specificity of abstract values. The first splits abstract values known to satisfy a disjunctive contract. For example, $\bullet/\{\lfloor \texttt{nat?} \rfloor \vee \lfloor \texttt{bool?} \rfloor\}\ \widehat{\mathbf{c}}\ \bullet/\lfloor \texttt{nat?} \rfloor$ and $\bullet/\lfloor \texttt{bool?} \rfloor$. This reifies the non-determinism of the value into non-determinism in the reduction relation, and makes subsequent uses of δ more precise on the two produced values. Similarly, we unfold recursive contracts in abstract values; this exposes further disjunctions to split, as with a contract for lists.

5.5 Base operations on abstract values

When applying base operations to abstract values, the results are potentially complex. For example, $(\texttt{add1}\ \bullet)$ might produce any natural number, or it might go wrong, depending on what value $\bullet$ represents. We represent this in the abstract version of δ, written $\widehat{\delta}$, with a combination of non-determinism, where $\widehat{\delta}$ 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.

The definition of $\widehat{\delta}$ relies on a proof system relating predicates and values, discussed below. Here, $V \vdash o?$ means that V is known

Base operations on abstract values $\widehat{\delta}(o^\ell, \boldsymbol{V}) \ni A$

$$\begin{array}{lcl}
V \vdash o? & \Longrightarrow & \widehat{\delta}(o?, V) \ni \texttt{\#t} \\
V \nvdash o? & \Longrightarrow & \widehat{\delta}(o?, V) \ni \texttt{\#f} \\
V \vdash^{\sim} o? & \Longrightarrow & \widehat{\delta}(o?, V) \ni \bullet/\{\lfloor \texttt{bool?} \rfloor\} \\
V \vdash \texttt{nat?} & \Longrightarrow & \widehat{\delta}(\texttt{add1}, V) \ni \bullet/\{\lfloor \texttt{nat?} \rfloor\} \\
V \nvdash \texttt{nat?} & \Longrightarrow & \widehat{\delta}(\texttt{add1}^\ell, V) \ni \texttt{blame}^\ell_{\texttt{add1}} \\
V \vdash^{\sim} \texttt{nat?} & \Longrightarrow & \widehat{\delta}(\texttt{add1}, V) \ni \bullet/\lfloor \texttt{nat?} \rfloor \\
 & & \widehat{\delta}(\texttt{add1}^\ell, V) \ni \texttt{blame}^\ell_{\texttt{add1}} \\
V \vdash \texttt{cons?} & \Longrightarrow & \widehat{\delta}(\texttt{car}, V) \ni \pi_1(V) \\
V \nvdash \texttt{cons?} & \Longrightarrow & \widehat{\delta}(\texttt{car}^\ell, V) \ni \texttt{blame}^\ell_{\texttt{car}} \\
V \vdash^{\sim} \texttt{cons?} & \Longrightarrow & \widehat{\delta}(\texttt{car}, V) \ni \pi_1(V) \\
 & & \widehat{\delta}(\texttt{car}^\ell, V) \ni \texttt{blame}^\ell_{\texttt{car}}
\end{array}$$

$$\pi_i(\bullet/\mathcal{C}) = \bullet/\{C_i \mid \langle C_1, C_2 \rangle \in \mathcal{C}\} \text{ for } i \in \{1,2\}$$
$$\pi_i(V_1, V_2) = V_i \text{ for } i \in \{1,2\}$$

Value proves or refutes base predicate $V \vdash o?$ and $V \nvdash o?$

$$\frac{\widetilde{\delta}(o?, V) \ni \texttt{\#t}}{V \vdash o?} \qquad \frac{C \vdash o?}{V/\mathcal{C} \cup \{C\} \vdash o?}$$

$$\frac{\widetilde{\delta}(o?, V) \ni \texttt{\#f}}{V \nvdash o?} \qquad \frac{C \nvdash o?}{V/\mathcal{C} \cup \{C\} \nvdash o?}$$

Value proves or refutes contract $V \vdash C$ and $V \nvdash C$

$$\frac{C \in \mathcal{C}}{V/\mathcal{C} \vdash C} \qquad \frac{V \vdash o?}{V \vdash \lfloor o? \rfloor}$$

$$\frac{V \nvdash o?}{V \nvdash \lfloor o? \rfloor} \qquad \frac{V \nvdash C \quad V \nvdash D}{V \nvdash C \vee D} \qquad \frac{V \nvdash C \text{ or } V \nvdash D}{V \nvdash C \wedge D}$$

$$\frac{C \in \{\lfloor \texttt{cons?} \rfloor, \lfloor \texttt{nat?} \rfloor, \lfloor \texttt{false?} \rfloor, \lfloor \texttt{bool?} \rfloor\} \quad V \vdash \texttt{proc?}}{V \nvdash C}$$

$$\frac{V \nvdash \texttt{proc?}}{V \nvdash C \rightarrow x.D} \qquad \frac{V \nvdash \texttt{cons?}}{V \nvdash \langle C, D \rangle} \qquad \frac{\pi_1(V) \nvdash C \text{ or } \pi_2(V) \nvdash D}{V \nvdash \langle C, D \rangle}$$

$$\frac{U \nvdash C \text{ or } V \nvdash D}{(U, V) \nvdash \langle C, D \rangle} \qquad \frac{V \nvdash [\mu x.C/x]C}{V \nvdash \mu x.C}$$

to satisfy $o?$, $V \nvdash o?$ means that V is known not to satisfy $o?$, and $V \vdash^{\sim} o?$ means V neither is known. For example, $7 \vdash$ `nat?` and $\bullet \vdash^{\sim} o?$ for any $o?$.

Projections are defined on all values that do not refute `cons?`. When a value is an actual pair, $\pi_{\{1,2\}}$ does the usual thing and accesses the left or right pair value component, respectively. For abstract values, $\pi_{\{1,2\}}$ constructs an abstract value that is the projection of all the left (or right) pair contract components, respectively. So for example, $\pi_1(\bullet/\{\langle C_1, C_2 \rangle, \langle D_1, D_2 \rangle\}) = \bullet/\{C_1, D_1\}$. Any non-pair contracts are ignored, so for example if only `cons?` is known about a value, the projection produces an abstract value with no constraints: $\pi_1(\bullet/\{\lfloor \texttt{cons?} \rfloor\} = \bullet$.

We now define the our proof system used by $\widehat{\delta}$, along with the $V \vdash C$ judgment used earlier in flat contract checking. We begin with the relation between values and base predicates. Here, we simply consult $\widetilde{\delta}$ where possible, and also examine the contracts that V remembers by referencing the subsequent relation between contracts and predicates. Note that consulting only $\widetilde{\delta}$ is necessary, as $\widehat{\delta}$ relies on the judgment we are defining. The judgment for $\nvdash$ is similar.

Second, we define a relation between values and contracts, used earlier in the definition of checking of flat contracts. The intuition is that $V \vdash C$ then V satisfies C, with the opposite relation $V \nvdash C$ implying that V does not satisfy C. Again, these rules are simple inclusions and exclusions based on injectivity of the base types of our semantics. One complication to note is that the $V \nvdash C$ relation is *co*inductive, to handle recursive contracts properly.

Both of these judgments rely on an auxiliary judgment between contracts and predicates, $C \vdash o?$, which we use merely as for $V \vdash o?$. The key intuition is that any value satisfying given C contract implies $o?$ holds on that value. The $\nvdash$ counterpart relation has a similar intuition—a value satisfying the given contract implies that $o?$ does not hold on that value. These rules are simple inclusions and exclusions based on injectivity of the base types of our semantics, and are deferred to the appendix.

5.6 Abstract reduction

We are now equipped to define our full notion of reduction for programs with opaque components.

Abstract reduction in context $P \longmapsto_{\widehat{\mathbf{c}}} Q$

$$\begin{array}{rcll}
\boldsymbol{M}\ \mathcal{E}[E] & \longmapsto_{\widehat{\mathbf{c}}} & \boldsymbol{M}\ \mathcal{E}[E] & \text{if } E\ \widehat{\mathbf{c}}\ F \\
\boldsymbol{M}\ \mathcal{E}[E] & \longmapsto_{\widehat{\mathbf{c}}} & \boldsymbol{M}\ \mathcal{E}[F] & \text{if } (E, F) \in \widehat{\Delta}(\boldsymbol{M}) \\
\boldsymbol{M}\ \mathcal{E}[\texttt{blame}^f_h] & \longmapsto_{\widehat{\mathbf{c}}} & \boldsymbol{M}\ \texttt{blame}^f_h & \text{if } \mathcal{E} \neq [\,]
\end{array}$$

However, there is one remaining complication. 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 complicate 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 now defined as:

$$eval_{\widehat{\mathbf{c}}}(\boldsymbol{M}E) = \{Q \mid \boldsymbol{M}\ F;\ E \longmapsto\!\!\!\!\to_{\widehat{\mathbf{c}}} Q\},$$

where $F = \textsc{amb}(\{\texttt{\#t}, \overline{(\texttt{DEMONIC}\ \boldsymbol{f})}\})$, $(\texttt{module}\ f\ C\ V) \in \boldsymbol{M}$.

6. Soundness

In this section we establish the soundness of our abstract reduction semantics. The soundness theorem states the abstract semantics approximates the concrete behavior of a program for all possible instantiations of opaque components. Soundness implies an important corollary for the verification of modules with respect to their specification: any concrete module that is not blamed during abstract reduction is *contract correct*; in all possible contexts, the module cannot be blamed.

The key intuition of the claim is that any program that has a set of opaque modules can be instantiated with arbitrary implementations, making the program fully concrete. Running the instantiated program under the concrete semantics may produce an answer which is either a value or blame. The abstract semantics semantics is sound in the following sense: running the uninstantiated, opaque program under the abstract semantics may produce a set of answers. If the concrete program produced a value, the opaque program produces an abstraction of that value. If the concrete program produced blame of a module that is *not* the instantiation of an opaque module, the abstract program produces the same blame.

This soundness result implies, for example, that if a program with opaque modules does not produce blame, the known modules cannot be blamed, regardless of the choice of implementation for the missing components.

Soundness relies on the definition of *approximation* between values. The basic intuition for approximation is that two concrete values approximate each other only if they are identical, but an abstract value, which can be thought of as standing for a set of acceptable concrete values, approximates a concrete value if that value in the set the abstract value denotes. For example, $\bullet/\lfloor\texttt{bool?}\rceil$ approximates `#t`, but not 7. By extension, an abstract value approximates another abstract value if it stands for a superset of values.

The approximation relation on expressions, modules, and programs is formalized below. We show only the important cases and omit the straightforward structurally recursive cases.

Approximates $\qquad P \sqsubseteq Q$, $M \sqsubseteq_{\boldsymbol{M}} N$, and $E \sqsubseteq_{\boldsymbol{M}} F$

$$(C \Leftarrow E) \sqsubseteq_{\boldsymbol{M}} \bullet/\mathcal{C} \cup \{C\} \qquad \frac{V \vdash C}{V \sqsubseteq_{\boldsymbol{M}} \bullet/\mathcal{C} \cup \{C\}}$$

$$((C \dashrightarrow x.D) \Leftarrow E) \sqsubseteq_{\boldsymbol{M}} \bullet/\mathcal{C} \cup \{C \rightarrow x.D\}$$

$$\frac{(\texttt{module}\ f\ C\ \bullet) \in \boldsymbol{M} \text{ or } f = \dagger}{\texttt{blame}^f_g \sqsubseteq_{\boldsymbol{M}} E} \qquad \frac{(C \Leftarrow E) \sqsubseteq_{\boldsymbol{M}} F}{(C \Leftarrow E) \sqsubseteq_{\boldsymbol{M}} (C \Leftarrow F)}$$

$$\frac{C \in \mathcal{C} \implies D \in \mathcal{C}'.D \sqsubseteq_{\boldsymbol{M}} C}{V/\mathcal{C} \sqsubseteq_{\boldsymbol{M}} \bullet/\mathcal{C}'} \qquad \frac{\boldsymbol{N} \sqsubseteq_{\boldsymbol{M}} \boldsymbol{M} \quad F \sqsubseteq_{\boldsymbol{M}} E}{\boldsymbol{N}F \sqsubseteq \boldsymbol{M}E}$$

$$\frac{(\texttt{module}\ f\ C\ \bullet) \in \boldsymbol{M}}{(\texttt{module}\ f\ C\ V) \sqsubseteq_{\boldsymbol{M}} (\texttt{module}\ f\ C\ \bullet)}$$

We lift $\sqsubseteq$ to evaluation contexts $\mathcal{E}$ by structural extension. We lift $\sqsubseteq$ to contracts by structural extension on contracts and $\sqsubseteq$ on embedded values. We lift $\sqsubseteq$ 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.

For space, we omit straightforward proofs of auxiliary lemmas.

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

Lemma 2. *If $E \sqsubseteq_{\boldsymbol{M}} F$ and $V \sqsubseteq_{\boldsymbol{M}} U$, then $[V/x]E \sqsubseteq_{\boldsymbol{M}} [U/x]F$.*

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

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

1. *if $V \vdash C$ and $V \sqsubseteq_{\boldsymbol{M}} U$, then $(E\ U) \longmapsto\!\!\!\!\twoheadrightarrow_{\hat{\mathbf{c}}} A \sqsupseteq$ `#t`,*
2. *if $V \nvdash C$ and $V \sqsubseteq_{\boldsymbol{M}} U$, then $(E\ U) \longmapsto\!\!\!\!\twoheadrightarrow_{\hat{\mathbf{c}}} A \sqsupseteq$ `#f`.*

Lemma 5. *If $P \longmapsto_{\hat{\mathbf{c}}} P'$, $P' \neq \boldsymbol{M}\ \texttt{blame}^f_g$ and $P \sqsubseteq Q$, then $Q \longmapsto\!\!\!\!\twoheadrightarrow_{\hat{\mathbf{c}}} Q'$ and $P' \sqsubseteq Q'$.*

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

$$P = \boldsymbol{M}\ \mathcal{E}[E] \longmapsto_{\hat{\mathbf{c}}} \boldsymbol{M}\ \mathcal{E}[E']$$
$$Q = \boldsymbol{N}\ \mathcal{E}'[F] \longmapsto_{\hat{\mathbf{c}}} \boldsymbol{N}\ \mathcal{E}'[F']$$

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

- Case: $E = f^g$ and $(f^g, E') \in \Delta(\boldsymbol{M})$
  If f is transparent in $\boldsymbol{N}$, then $(f^g, E') \in \widehat{\Delta}(\boldsymbol{N})$ and we are done. Otherwise, $f \neq g$ and thus
  $$E' = (C \Leftarrow^{f,g}_{f} V) \qquad F' = (C \Leftarrow^{f,g}_{f} \bullet/\{C\}),$$
  but now $E' \sqsubseteq_{\boldsymbol{N}} F'$, since $E' \sqsubseteq_{\boldsymbol{N}} \bullet/\{C\}$.
- Case: $(C \rightarrow x.D \Leftarrow V)\ \hat{\mathbf{c}}\ ((C \dashrightarrow x.D) \Leftarrow U)$ where $\delta(\texttt{proc?}, V) \ni$ `#t` and $U = V/\{C \rightarrow x.D\}$.
  Since F is a redex, by $\sqsubseteq$ we have $F = (C' \rightarrow y.D' \Leftarrow V')$, where $V \sqsubseteq_{\boldsymbol{N}} V'$. By lemma 1, $\delta(\texttt{proc?}, V') \ni$ `#t`. So $F' = ((C' \dashrightarrow y.D') \Leftarrow U')$ where $U' = V'/\{C' \rightarrow y.C'\} \sqsubseteq_{\boldsymbol{N}} V/\{C \rightarrow x.D\}$ and thus $E' \sqsubseteq_{\boldsymbol{N}} F'$.
- Case: $(V_1\ V_2)^\ell\ \hat{\mathbf{c}}\ \texttt{blame}^\ell_\Lambda$, where $\delta(\texttt{proc?}, V_1) \ni$ `#f`.
  By $\sqsubseteq$, we have $F = (U_1\ U_2)^\ell$ and $U_i \sqsubseteq_{\boldsymbol{N}} V_i$. By lemma 1, $\delta(\texttt{proc?}, U_1) \ni$ `#f`, hence $(U_1\ U_2)^\ell\ \hat{\mathbf{c}}\ \texttt{blame}^\ell_\Lambda$.
- Case: $(V_1\ V_2)^\ell\ \hat{\mathbf{c}}\ E'$, where $\delta(\texttt{proc?}, V_1) \ni$ `#t`.
  By $\sqsubseteq$, we have $F = (U_1\ U_2)^\ell$ and $U_i \sqsubseteq_{\boldsymbol{N}} V_i$. By lemma 1, $\delta(\texttt{proc?}, U_1) \ni$ `#t`.
  Either V_1 and U_1 are structurally similar, in which case the result follows by possibly relying on lemma 2, or $V_1 = (\lambda_y x.E_0)/\mathcal{C}$ 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 $F' = \bullet$, and the result holds. Alternatively, V_1 was blessed and $\mathcal{C}$ contains a function contract $C \rightarrow x.D$. But but by the blessed application rule, we have $\mathcal{E} = \mathcal{E}_1[([V_2'/x]D \Leftarrow [\ ])]$, thus by assumption $\mathcal{E}' = \mathcal{E}_1'[([U_2'/x]D' \Leftarrow [\ ])]$, implying $[V_2'/x]D \sqsubseteq [U_2'/x]D'$, finally giving us the needed conclusion:
  $$\mathcal{E}_1[([V_2'/x]D \Leftarrow E')] \sqsubseteq_{\boldsymbol{N}} \mathcal{E}_1'[([U_2'/x]D' \Leftarrow F')],$$
  where $F' = \bullet/\{[U_2/x]D' \mid C' \rightarrow x.D' \in \mathcal{C}'\}$.
- Case: $(C \Leftarrow V)\ \hat{\mathbf{c}}\ E'$ where C is flat.
  If $V \vdash^{\sim} C$, then the case holds by use of lemma 3. If $V \vdash C$, then the case holds by lemma 4(1). If $V \nvdash C$, then the case holds by lemma 4(2).
- The remaining cases are straightforward.

Case (2):

$$P = \boldsymbol{M}\ \mathcal{E}_1[\mathcal{E}_2[E]] \longmapsto_{\hat{\mathbf{c}}} \boldsymbol{M}\ \mathcal{E}_1[\mathcal{E}_2[E']]$$
$$Q = \boldsymbol{N}\ \mathcal{E}_1'[\mathcal{E}_2'[F]]$$

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

In this case, we have $\mathcal{E}_2[E] \sqsubseteq_{\boldsymbol{N}} \mathcal{E}_2'[F]$, but since $\mathcal{E}_2 \not\sqsubseteq_{\boldsymbol{N}} \mathcal{E}_2'$, 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. ☐

Lemma 6. *If $\boldsymbol{M}E \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{c}} \boldsymbol{M}A$, then $\boldsymbol{M}E \longmapsto\!\!\!\!\twoheadrightarrow_{\hat{\mathbf{c}}} \boldsymbol{M}A/\mathcal{C}$.*

Lemma 7. *If there exists a context $\mathcal{E}$ such that*

$$\boldsymbol{M}\ (\texttt{module}\ f\ C\ V)\ \mathcal{E}[f] \longmapsto\!\!\!\!\twoheadrightarrow_{\mathbf{c}} \texttt{blame}^f_g,$$

then

$$\boldsymbol{M}\ (\texttt{module}\ f\ C\ V)\ (\texttt{DEMONIC}\ f) \longmapsto\!\!\!\!\twoheadrightarrow_{\hat{\mathbf{c}}} \texttt{blame}^f_g.$$

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

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

but then there exists a $\mathcal{D}'$ equal to $\mathcal{D}$ with all values replaced with $\bullet$ such that $\boldsymbol{M}\ \mathcal{D}'[V] \longmapsto\!\!\!\!\twoheadrightarrow_{\hat{\mathbf{c}}} \texttt{blame}^f_g$. 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 `DEMONIC`, if $\boldsymbol{M}\ \mathcal{D}'[V] \longmapsto\!\!\!\!\twoheadrightarrow_{\hat{\mathbf{c}}} \texttt{blame}^f_g$, then $\boldsymbol{M}\ (\texttt{DEMONIC}\ V) \longmapsto\!\!\!\!\twoheadrightarrow_{\hat{\mathbf{c}}} \texttt{blame}^f_g$. ☐

Theorem 1 (Soundness)**.** *If $P \sqsubseteq Q$ where $Q = \boldsymbol{M}E$ and $A \in eval_{\hat{\mathbf{c}}}(P)$ there exists some $B \in eval_{\hat{\mathbf{c}}}(Q)$ where $A \sqsubseteq_{\boldsymbol{M}} B$.*

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

If $A = \mathtt{blame}^{\ell}_{\ell'}$ then there are two possibilities. If ℓ is the name of an opaque module in $\boldsymbol{M}$ or if $\ell = \dagger$, then $A \sqsubseteq_{\boldsymbol{M}} B$ immediately. If $\ell = f$ is the name of a concrete module in $\boldsymbol{M}$, then $(\mathtt{DEMONIC}\ f) \longmapsto\!\!\!\!\twoheadrightarrow_{\widehat{c}} A$ by lemma 7, and therefore $A \in eval_{\widehat{c}}(Q)$ by the definition of $eval_{\widehat{c}}$. ☐

With the soundness theorem in hand, we can immediately conclude our desired result: that our abstract reduction semantics verifies the absence of blame for concrete modules.

Corollary 1. *If $P \not\longmapsto\!\!\!\!\twoheadrightarrow_{\widehat{c}} \mathtt{blame}^{f}_{g}$ for some concrete f, then no instantiation of the opaque modules in P can cause f to be blamed.*

7. Computable approximation

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 modular programs satisfy their specifications. However, the abstract semantics subsumes the concrete semantics and consequently cannot, in general, be used to *automatically* verify contract correctness of programs since the abstract semantics of a program is undecidable.

In this section, we describe a modular program analysis based on our previous abstract reduction semantics that enables precisely this kind of automation at the expense of precision.

Program analysis aims to soundly *and computably* predict the behavior of programs. Our abstract semantics enjoys the first property, but lacks the second. To achieve decidability we will construct a finite-state approximation to the execution of programs with potentially opaque components. The resulting analysis makes sound predictions and can be used as the basis for automatic program verification and analysis.

We adopt the *abstracting abstract machines* [34] approach to deriving a program analysis, which provides a recipe for going from a high-level semantics to a low-level computable approximation. The recipe is to first derive a first-order state-transition system (an abstract machine) that precisely characterizes the semantics, then to refactor the machine to thread any recursive structure through the machine's store. Finally the machine, which has a potentially infinite state-space, is approximated by bounding the size of the store. Store updates are interpreted as joins to a set of possible values residing at a location, and store dereferences are interpreted as a non-deterministic choice among the values in a store location. Since all recursive structure has been threaded through the store and store is of finite size, the resulting machine has a finite state-space, while the non-determinism of store dereference leads to a straightforward soundness argument.

7.1 An abstract machine for analysis

We begin by deriving an abstract machine for the reduction semantics of section 5. The machine is a variant of the CESK machine [10]; machine states consist of a control string (an expression), an environment that closes free variables in the control string, a store mapping addresses to heap-allocated values, and a continuation, which represents the evaluation context of the control string. Environments are finite maps from variables to addresses and stores are finite maps from addresses to values. Values are pairs of terms and environments, called closures, or pairs of closures. We let a and b range over value pointers (addresses that resolve to values in the store), and k and i range over continuation pointers (addresses that resolve to continuations).

Environments and stores

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

Values are extended with pairs of closures and blessed pointers. The former is needed to represent pairs in an environment model. The latter are needed to ensure programs generate only a finite amount of new syntax when run.

Values

$$\begin{array}{rcl}
V & += & ((V,\rho),(V,\rho)) \\
 & \mid & ((C \dashrightarrow x.C) \Leftarrow^{f,f}_{f} a)
\end{array}$$

Continuations

$$\begin{array}{rcl|l}
\kappa, \iota & ::= & \mathsf{mt} & \mathsf{op}^{\ell}(o,k) \\
 & \mid & \mathsf{ar}^{\ell}(E,\rho,k) & \mathsf{opl}^{\ell}(o,E,\rho,k) \\
 & \mid & \mathsf{fn}^{\ell}(V,\rho,k) & \mathsf{opr}^{\ell}(o,V,\rho,k) \\
 & \mid & \mathsf{if}(E,E,\rho,k) & \mathsf{chk\text{-}or}^{f,f}_{f}(V,\rho,C,\rho,k) \\
 & \mid & \mathsf{chk}^{f,f}_{f}(C,\rho,k) & \mathsf{chk\text{-}cons}^{f,f}_{f}(C,\rho,V,\rho,k)
\end{array}$$

Continuations represent evaluation contexts as follows: mt represents $[\ ]$; $\mathsf{ar}^{\ell}(E,\rho,k)$ represents $\mathcal{E}[([\ ]\ E')^{\ell}]$, where ρ closes E to represent E' and k points to a continuation κ that represents $\mathcal{E}$; and so on. The continuations forms are standard with the exception of those for contract checking. The $\mathsf{chk}^{f,g}_{h}(C,\rho,k)$ continuation represents an evaluation context $(C' \Leftarrow^{f,g}_{h} [\ ])$ where ρ closes any embedded terms in the contract C to represent C'. The $\mathsf{chk\text{-}or}$ and $\mathsf{chk\text{-}cons}$ special purpose variants that are used to represent continuations of disjunctive and pair contracts. For example, $\mathsf{chk\text{-}cons}^{f,g}_{h}(C,\varrho,V,\rho,k)$ represents $(\mathtt{cons}\ [\ ]\ (C' \Leftarrow^{f,g}_{h} (\mathtt{cdr}\ V')))$, where ϱ closes terms embedded in C to represent C', while ρ closes V to represent V'.

The machine transition system is defined on page 10. Although the definition is formidable, it can be derived in a systematic way.

States of the machine are represented as quadruples of expressions, environments, continuations, and stores:

$$\Sigma ::= \langle E, \rho, \kappa, \sigma \rangle$$

We give a representative selection of machine transition rules on page 10. The full machine definition is given in the supplemental material. We now define the evaluation function of the machine, which computes the set of reachable machine states:

$$eval^{\mathtt{M}}_{\widehat{c}}(P) = \{\Sigma' \mid inj(P) \longmapsto\!\!\!\!\twoheadrightarrow \Sigma'\},$$
$$\text{where } inj(\boldsymbol{M}E) = \langle E, \emptyset, \mathsf{mt}, \emptyset \rangle.$$

7.2 Allocation and approximation

The machine is parameterized by an allocation strategy embodied in the alloc function:

$$(\boldsymbol{i}, \boldsymbol{b}) = \mathsf{alloc}(\Sigma),$$

which produces a vector of continuation and value pointers.

When allocation produces only fresh addresses that are not in use in the store, the machine carries out program execution exactly as in the reduction semantics. The machine therefore can act as low-level interpreter for our language. On the other hand, if allocation produces addresses from some finite set, the machine's state-space is bounded, causing the machine to approximate the actual execution of a program. This restriction implies alloc may allocate an address already in use. To ensure soundness, the machine interprets store updates as joins which place multiple values in any given store location. When dereference the store, a non-deterministic choice is made as to which value is fetched.

Abstract machine (selected transitions) $\Sigma = \langle E, \rho, \kappa, \sigma\rangle \longmapsto \langle F, \varrho, \iota, \varsigma\rangle$, *where* $(\boldsymbol{i}, \boldsymbol{b}) = \mathsf{alloc}(\Sigma)$

$$\begin{array}{lcll}
\langle f^g, \rho, \kappa, \sigma\rangle & \longmapsto & \langle V, \emptyset, \mathsf{chk}^{f,g}_h(C, \emptyset, i_1), \sigma[i_1 \mapsto \kappa]\rangle & \text{if } (f^f, (C \Leftarrow^{f,g}_h V)) \in \widehat{\Delta}(\boldsymbol{M}) \\
\langle (C \Leftarrow^{f,g}_h E), \rho, \kappa, \sigma\rangle & \longmapsto & \langle E, \rho, \mathsf{chk}^{f,g}_h(C, \rho, i_1), \sigma[i_1 \mapsto \kappa]\rangle & \\
\langle V, \rho, \mathsf{fn}^\ell(((C \dashrightarrow x.D) \Leftarrow^{f,g}_h a), \varrho, k), \sigma\rangle & \longmapsto & \langle V, \rho, \mathsf{chk}^{g,f}_h(C, \varrho, i_1), \sigma[i_1 \mapsto \mathsf{fn}^\ell(U, \varrho', i_2), & \\
 & & \qquad i_2 \mapsto \mathsf{chk}^{f,g}_h(D, \varrho[x \mapsto b_1], k), & \\
 & & \qquad b_1 \mapsto (V, \rho)]\rangle & \\
 & & \text{where } (U, \varrho') \in \sigma(a) & \\
\langle V, \rho, \mathsf{chk}^{f,g}_h(C \rightarrow x.D, \varrho, k), \sigma\rangle & \longmapsto & \langle ((C \dashrightarrow x.D) \Leftarrow^{f,g}_h b_1), \varrho, \kappa, \sigma[b_1 \mapsto (V, \rho)]\rangle & \text{if } \delta(\texttt{proc?}, V) \ni \texttt{\#t}, \kappa \in \sigma(k) \\
\langle V, \rho, \mathsf{fn}^\ell(\bullet/\mathcal{C}, \varrho, k), \sigma\rangle & \longmapsto & \langle \bullet/\{[V/x]D \mid C \rightarrow x.D \in \mathcal{C}\}, \rho, \kappa, \sigma\rangle & \text{if } \delta(\texttt{proc?}, \bullet/\mathcal{C}) \ni \texttt{\#t}, \kappa \in \sigma(k) \\
\langle V, \rho, \mathsf{fn}^\ell(\bullet/\mathcal{C}, \varrho, k), \sigma\rangle & \longmapsto & \langle V, \rho, \mathsf{fn}(\texttt{DEMONIC}, \emptyset, k), \sigma\rangle & \text{if } \delta(\texttt{proc?}, \bullet/\mathcal{C}) \ni \texttt{\#t}
\end{array}$$

Typically, machines such as the CESK are constructed on the basis of a deterministic reduction semantics and are therefore deterministic machines. When bounded to perform program analysis they become non-deterministic. In our setting however, we have based the machine on a non-deterministic reduction semantics, so even if the machine uses a fresh allocation strategy, the machine just as non-deterministic as the semantics. Bounding the machine's store imposes an additional level of non-determinism that ensures computability while retaining soundness.

For the purposes of our implementation, we instantiate alloc for a 0CFA-like approximation. To allocate bindings, we use the variable names as addresses. To allocate continuations, we use the continuation frame (the continuation but without the pointer to the next frame) as the address. We also incorporate abstract garbage collection, which improves precision and performance.

7.3 Soundness and decidability

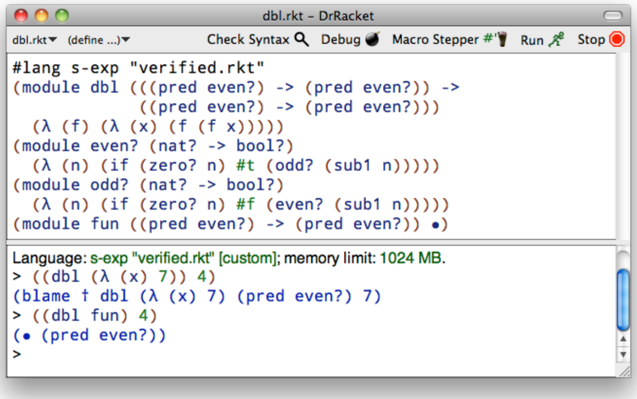
Using an unbounded allocation strategy, it is straightforward to show the machine is a faithful evaluator for both the abstract reduction semantics. The proof follows the outline of a standard proof [11, page 102], adapted *mutatis mutandis*. Having established the correspondence, relating the computable approximation to the reduction semantics is now only a matter of relating the approximating and non-approximating machine. This proof is likewise a straightforward adaptation of that given by Van Horn and Might [34]. The key idea is to define a partial order on the abstract machine's state-space and show that the finite machine maintains a step-wise approximation of the unbounded machine, which involves straightforward reasoning by case analysis on the machine transition relation.

Decidability follows from the finiteness of the machine's state-space. Threading closures and continuations through a bounded store bounds both the space of function values and the control stack. Primitive operations and base values are the remaining source of infinite domains in the machine. We apply a simple widening operation on the results of δ that ensures the set of abstract base values is finite.

Discussion It is worth noting that in this section we have applied a semantics-based technique for deriving *whole-program* analysis. However, by applying the recipe to a semantics that reasons about modular programs, a *modular* analysis was obtained. Although modular program analysis has been the topic of significant research, this observation suggests that modularity and approximation may be considered orthogonal.

8. Implementation

We have implemented our system in a prototype tool for verifying contracts, as seen in figure 1. We make use of the Redex tool [11] to directly translate our reduction semantics as well as our abstract machine definition into executable form. This produces a tool



```
#lang s-exp "verified.rkt"
(module dbl (((pred even?) -> (pred even?)) ->
             ((pred even?) -> (pred even?)))
  (λ (f) (λ (x) (f (f x)))))
(module even? (nat? -> bool?)
  (λ (n) (if (zero? n) #t (odd? (sub1 n)))))
(module odd? (nat? -> bool?)
  (λ (n) (if (zero? n) #f (even? (sub1 n)))))
(module fun ((pred even?) -> (pred even?)) •)
```

```
Language: s-exp "verified.rkt" [custom]; memory limit: 1024 MB.
> ((dbl (λ (x) 7)) 4)
(blame † dbl (λ (x) 7) (pred even?) 7)
> ((dbl fun) 4)
(• (pred even?))
>
```

Figure 1. Interactive program verification environment

which, when given a program written in the concrete syntax seen in figure 1, produces the set of possible results predicted by the semantics.

As seen above, we can take the example directly from the introduction, define the relevant modules, and explore the behavior of different choices for the main expression. In other words, we have an interactive environment for exploring the behavior of our verification system. The labeling and annotation required in our model is automatically inserted.

Our tool is built on top of Racket [15] and DrRacket [13], which provide an extensible programming language and development environment. We are able to define our semantics as a new language [33], so that files written in our new language can be compiled and executed with the Racket toolchain, using the semantics of this paper.

We also take advantage of the built-in visualization support in Redex so that users of our tool can select to visualize the graph of the state-space explored by the reduction system. Users can select either the uncomputable reduction semantics of section 5, the computable semantics of section 7, or a visualization of the behavior of either one.

Implementation extensions To make our prototype more useful, we have implemented several extensions to the system described in foregoing discussion to bring our language closer to existing Racket programs, our goal for verification.

- First, we support multiple argument functions. This complicates the definitions of 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 the FC metafunction more sophisticated, so that it generates less complex functions, improving running time and simplifying visualization.
- Fourth, we implement abstract garbage collection [27] in the computable model, reducing the size of the state-space explored in practice. In particular, abstract GC enables naive allocation strategies to perform with high precision since store merges are avoided by reclaiming unreachable space.
- Fifth, we add simpler rules to the model to handle such cases as non-recursive functions and non-dependent contracts. This brings the model closer to programmers expectation of the semantics of the language, and simplifies visualizations.
- Finally, we provide richer blame information when programs go wrong, as can be seen in the screen shot of figure 1. In addition to the $\texttt{blame}^{\ell}_{\ell'}$ result, which indicates the blamed party between ℓ and ℓ', our semantics is extended to track the first-order component of the contract that witnessed the failure, and the first-order value that directly caused the failure, along with the higher-order value that was ultimately responsible for causing the contract failure. This is the full complement of information available in Racket's production contract library, which reports the failure of our dbl example as:

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

Our prototype is available as open source software at

http://github.com/samth/var/

9. Related work

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

Symbolic evaluation and abstract interpretation Symbolic execution [19] 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.

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.

Abstract interpretation provides a general theory of semantic approximation [6] that relates concrete semantics to abstract semantics interpreting programs over a domain of abstract values. Our approach is very much an instance of abstract interpretation. The reachable state semantics of $\mathbf{c}$ is our concrete semantics. The $\widehat{\mathbf{c}}$ is 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 [8]. Our work applies this idea to behavioral contracts and higher-order programs.

Verification of contracts The most closely related work to ours is the modular set-based analysis based on contracts by Meunier et al. [24, 25]. 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 only used in the proof. Despite this, the system is unsound, since it lacks an analogue of DEMONIC. 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 that always pass is also greater—Meunier's system includes only the first of our rules for $V \vdash C$.

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. [35] 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 [12], run the GHC optimizer, and examine the result to see if any contract checks are left in the residual program. If there are, the system reports them as potentially failing.

In addition to the manifest limitations of relying on what the compiler can optimize away to implement contract verification, Xu et al. also consider a more limited set of contracts, not including disjunction, conjunction, or data structures, and explicitly prohibit contracts from containing code that might possibly go wrong. In contrast, we directly reason about the semantics of our language, handle a rich contract language, and allow arbitrary code in predicates. 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. Further, their system does not attempt to track the possible parties to blame errors.

Blume and McAllester [5] 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 DEMONIC 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.

Many researchers have studied verifying first-order properties of programs expressed as contracts [4, 8]. Such analyses could improve precision for first-order predicate checks in our system.

Combining expressions with specifications Giving semantics to programs combined with specifications has a long history in the setting of program refinements [18]. 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 verification, and (c) the use of higher-order contracts as specifications.

Type inference and checking can be recast as a reduction semantics [22], and doing so bears a conceptual resemblance to our types-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 types and 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 [29] 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 and thus could be derived from the semantics of the language rather than requiring custom design.

Modular program analysis Shivers [32], Serrano [31], 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 $\bullet$ 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 [14] 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 [2, 3] and Lee et al. [23] take similar approaches to type-based and 0CFA-style analyses, respectively.

Other approaches to higher-order verification Kobayashi et al. [20, 21] have recently proposed approaches to verification of temporal properties of higher-order programs based on model checking. This work differs from ours in three 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. Despite these differences, we believe that the combination of reduction semantics and model checking approaches to higher-order program verification is a fruitful area of future work.

Rondon et al. [30] 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 $V \vdash C$ relation would increase the precision and effectiveness of our system.

10. Conclusion

We have presented a spectrum of abstractions 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 using 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 is a powerful technique which turns the semantics of a programming language with executable specifications in a modular verifier for those specifications.

Acknowledgments: We are grateful to Phillipe Meunier for discussions of his prior work and providing code for the prototype implementation of his system; to Casey Klein for help with Redex; and to Christos Dimoulas for discussions and advice.

References

[1] J. M. Ashley and R. K. Dybvig. A practical and flexible flow analysis for higher-order languages. *ACM TOPLAS*, 20(4), 1998.

[2] A. Banerjee. A modular, polyvariant and type-based closure analysis. In A. M. Berman, editor, *ICFP '97*. ACM, 1997.

[3] A. Banerjee and T. Jensen. Modular control-flow analysis with rank 2 intersection types. *Mathematical. Structures in Comp. Sci.*, 13(1), 2003.

[4] M. Barnett, M. Fähndrich, K. R. M. Leino, P. Müller, W. Schulte, and H. Venter. Specification and verification: The Spec# experience. *Comm. of the ACM*, 2010.

[5] M. Blume and D. McAllester. Sound and complete models of contracts. *Journal of Functional Programming*, 16, 2006.

[6] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL '77*. ACM, 1977.

[7] C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. POPL '11. ACM, 2011.

[8] M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. In *FoVeOOS'10*. Springer-Verlag, 2011.

[9] M. Fähndrich, M. Barnett, and F. Logozzo. Embedded contract languages. In *SAC'10*. ACM, 2010.

[10] M. Felleisen and D. P. Friedman. A calculus for assignments in higher-order languages. In *POPL'87*. ACM, 1987.

[11] M. Felleisen, R. B. Findler, and M. Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[12] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP '02*. ACM, 2002.

[13] R. B. Findler, J. Clements, C. Flanagan, M. Flatt, S. Krishnamurthi, P. Steckler, and M. Felleisen. Drscheme: a programming environment for Scheme. *JFP*, 12(02), 2002.

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

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

[16] M. Greenberg. personal communication.

[17] R. Hinze, J. Jeuring, and A. Löh. Typed contracts for functional programming. In M. Hagiya and P. Wadler, editors, *FLOPS'10*, volume 3945 of *LNCS*, chapter 15. Springer, 2006.

[18] R. Johan, A. Akademi, and J. V. Wright. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag New York, Inc., 1998.

[19] J. C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7), 1976.

[20] N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. POPL '09. ACM, 2009.

[21] N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. In *PLDI'11*. ACM, 2011.

[22] G. Kuan, D. MacQueen, and R. B. Findler. A rewriting semantics for type inference. In R. D. Nicola, editor, *ESOP '07*, volume 4421, 2007.

[23] O. Lee, K. Yi, and Y. Paek. A proof method for the correctness of modularized 0CFA. *IPL*, 81, 2002.

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

[25] P. Meunier, R. B. Findler, and M. Felleisen. Modular set-based analysis from contracts. In *POPL '06*. ACM, 2006.

[26] B. Meyer. *Eiffel : The Language*. Prentice Hall PTR, 1991.

[27] M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP'06*, 2006.

[28] S. S. Muchnick and N. D. Jones, editors. *Program Flow Analysis: Theory and Applications*. Prentice Hall, 1981.

[29] J. Reppy. Type-sensitive control-flow analysis. In *ML '06*. ACM, 2006.

[30] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. PLDI '08. ACM, 2008.

[31] M. Serrano. Control flow analysis: a functional languages compilation paradigm. In *SAC '95*. ACM, 1995.

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

[33] S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. PLDI '11. ACM, 2011.

[34] D. Van Horn and M. Might. Abstracting abstract machines. In *ICFP'10*. ACM, 2010.

[35] D. N. Xu, S. Peyton Jones, and S. Claessen. Static contract checking for Haskell. In *POPL '09*. ACM, 2009.

A. Auxiliary definitions

A.1 Operations on concrete values

We assume V, U are concrete here.

$$
\begin{aligned}
& (\texttt{add1}, n, n+1) \in \widetilde{\delta} \\
& (\texttt{car}, (U,V), U) \in \widetilde{\delta} \\
& (\texttt{cdr}, (U,V), V) \in \widetilde{\delta} \\
& (\texttt{+}, n, m, n+m) \in \widetilde{\delta} \\
& (\texttt{=}, n, n, \texttt{\#t}) \in \widetilde{\delta} \\
n \neq m \implies & (\texttt{=}, n, m, \texttt{\#f}) \in \widetilde{\delta} \\
& (\mathsf{cons}, U, V, (U,V)) \in \widetilde{\delta} \\
& (\texttt{nat?}, n, \texttt{\#t}) \in \widetilde{\delta} \\
V \notin \mathcal{N} \implies & (\texttt{nat?}, V, \texttt{\#f}) \in \widetilde{\delta} \\
V \in \{\texttt{\#t}, \texttt{\#f}\} \implies & (\texttt{bool?}, V, \texttt{\#t}) \in \widetilde{\delta} \\
V \notin \{\texttt{\#t}, \texttt{\#f}\} \implies & (\texttt{bool?}, V, \texttt{\#f}) \in \widetilde{\delta} \\
& (\texttt{empty?}, \texttt{empty}, \texttt{\#t}) \in \widetilde{\delta} \\
V \neq \texttt{empty} \implies & (\texttt{empty?}, V, \texttt{\#f}) \in \widetilde{\delta} \\
& (\texttt{cons?}, (U,V), \texttt{\#t}) \in \widetilde{\delta} \\
V' \neq (U,V) \implies & (\texttt{cons?}, V', \texttt{\#f}) \in \widetilde{\delta} \\
& (\texttt{proc?}, (\lambda_x y.E), \texttt{\#t}) \in \widetilde{\delta} \\
V \neq (\lambda_x y.E) \implies & (\texttt{proc?}, V, \texttt{\#f}) \in \widetilde{\delta} \\
& (\texttt{false?}, \texttt{\#f}, \texttt{\#t}) \in \widetilde{\delta} \\
V \neq \texttt{\#f} \implies & (\texttt{false?}, V, \texttt{\#f}) \in \widetilde{\delta}
\end{aligned}
$$

(We have omitted the rules producing blame for arity mismatch and undefined cases.)

A.2 Operations on abstract values

We assume V, U are abstract here.

$$
\begin{aligned}
V \vdash o? \implies & (o?, V, \texttt{\#t}) \in \widehat{\delta} \\
V \nvdash o? \implies & (o?, V, \texttt{\#f}) \in \widehat{\delta} \\
V \vdash^{\sim} o? \implies & (o?, V, \bullet/\lfloor\texttt{bool?}\rfloor) \in \widehat{\delta} \\
V \vdash \texttt{nat?} \implies & (\texttt{add1}, V, \bullet/\lfloor\texttt{nat?}\rfloor) \in \widehat{\delta} \\
V \nvdash \texttt{nat?} \implies & (\texttt{add1}^{\ell}, V, \texttt{blame}^{\ell}_{\texttt{add1}}) \in \widehat{\delta} \\
V \vdash^{\sim} \texttt{nat?} \implies & (\texttt{add1}, V, \bullet/\lfloor\texttt{nat?}\rfloor) \in \widehat{\delta} \\
& \wedge (\texttt{add1}^{\ell}, V, \texttt{blame}^{\ell}_{\texttt{add1}}) \in \widehat{\delta} \\
V \vdash \texttt{cons?} \implies & (\texttt{car}, V, \pi_1(V)) \in \widehat{\delta} \\
V \nvdash \texttt{cons?} \implies & (\texttt{car}^{\ell}, V, \texttt{blame}^{\ell}_{\texttt{car}}) \in \widehat{\delta} \\
V \vdash^{\sim} \texttt{cons?} \implies & (\texttt{car}, V, \pi_1(V)) \in \widehat{\delta} \\
& \wedge (\texttt{car}^{\ell}, V, \texttt{blame}^{\ell}_{\texttt{car}}) \in \widehat{\delta} \\
V \vdash \texttt{cons?} \implies & (\texttt{cdr}, V, \pi_2(V)) \in \widehat{\delta} \\
V \nvdash \texttt{cons?} \implies & (\texttt{cdr}^{\ell}, V, \texttt{blame}^{\ell}_{\texttt{cdr}}) \in \widehat{\delta} \\
V \vdash^{\sim} \texttt{cons?} \implies & (\texttt{cdr}, V, \pi_2(V)) \in \widehat{\delta} \\
& \wedge (\texttt{cdr}^{\ell}, V, \texttt{blame}^{\ell}_{\texttt{cdr}}) \in \widehat{\delta} \\
V \vdash \texttt{nat?} \wedge U \vdash \texttt{nat?} \implies & (\texttt{+}, V, U, \bullet/\lfloor\texttt{nat?}\rfloor) \in \widehat{\delta} \\
V \nvdash \texttt{nat?} \implies & (\texttt{+}, V, U, \texttt{blame}^{\ell}_{\texttt{+}}) \in \widehat{\delta} \\
U \nvdash \texttt{nat?} \implies & (\texttt{+}, V, U, \texttt{blame}^{\ell}_{\texttt{+}}) \in \widehat{\delta} \\
V \vdash^{\sim} \texttt{nat?} \wedge U \vdash^{\sim} \texttt{nat?} \implies & (\texttt{+}, V, U, \bullet/\lfloor\texttt{nat?}\rfloor) \in \widehat{\delta} \\
V \vdash^{\sim} \texttt{nat?} \implies & (\texttt{+}, V, U, \texttt{blame}^{\ell}_{\texttt{+}}) \in \widehat{\delta} \\
U \vdash^{\sim} \texttt{nat?} \implies & (\texttt{+}, V, U, \texttt{blame}^{\ell}_{\texttt{+}}) \in \widehat{\delta} \\
V \vdash \texttt{nat?} \wedge U \vdash \texttt{nat?} \implies & (\texttt{=}, V, U, \bullet/\lfloor\texttt{bool?}\rfloor) \in \widehat{\delta} \\
V \nvdash \texttt{nat?} \implies & (\texttt{=}, V, U, \texttt{blame}^{\ell}_{\texttt{=}}) \in \widehat{\delta} \\
U \nvdash \texttt{nat?} \implies & (\texttt{=}, V, U, \texttt{blame}^{\ell}_{\texttt{=}}) \in \widehat{\delta} \\
V \vdash^{\sim} \texttt{nat?} \wedge U \vdash^{\sim} \texttt{nat?} \implies & (\texttt{=}, V, U, \bullet/\lfloor\texttt{bool?}\rfloor) \in \widehat{\delta} \\
V \vdash^{\sim} \texttt{nat?} \implies & (\texttt{=}, V, U, \texttt{blame}^{\ell}_{\texttt{=}}) \in \widehat{\delta} \\
V_1 \vdash^{\sim} \texttt{nat?} \implies & (\texttt{=}, V, U, \texttt{blame}^{\ell}_{\texttt{=}}) \in \widehat{\delta} \\
& (\mathsf{cons}, V, U, (V,U)) \in \widehat{\delta}
\end{aligned}
$$

Contract proves or refutes base predicate $C \vdash o?$ and $C \nvdash o?$

$$
\lfloor\texttt{false?}\rfloor \vdash \texttt{bool?} \qquad \langle C{,}D\rangle \vdash \texttt{cons?} \qquad C \to x.D \vdash \texttt{proc?}
$$

$$
\lfloor o?\rfloor \vdash o? \qquad \frac{C \vdash o? \quad D \vdash o?}{C \vee D \vdash o?} \qquad \frac{C \vdash o? \text{ or } D \vdash o?}{C \wedge D \vdash o?}
$$

$$
\frac{o? \neq \texttt{proc?}}{C \to x.D \nvdash o?} \qquad \frac{o? \neq \texttt{cons?}}{\langle C{,}D\rangle \nvdash o?} \qquad \frac{C \nvdash o? \quad D \nvdash o?}{C \vee D \nvdash o?}
$$

$$
\frac{C \nvdash o? \text{ or } D \nvdash o?}{C \wedge D \nvdash o?} \qquad \frac{[\mu x.C/x]C \nvdash o?}{\mu x.C \nvdash o?}
$$

$$
\frac{o? \neq o?' \quad \{o?, o?'\} \neq \{\texttt{false?}, \texttt{bool?}\}}{\lfloor o?'\rfloor \nvdash o?}
$$

Basic reductions $\langle E,\rho,\kappa,\sigma\rangle \longmapsto \langle F,\varrho,\iota,\varsigma\rangle$

$$\begin{array}{llll}
\langle (E\ F)^\ell,\rho,\kappa,\sigma\rangle & \longmapsto & \langle E,\rho,\mathsf{ar}^\ell(F,\rho,k),\sigma[k\mapsto\kappa]\rangle & \\
\langle (\mathtt{if}\ E\ F_1\ F_2),\rho,\kappa,\sigma\rangle & \longmapsto & \langle E,\rho,\mathsf{if}(F_1,E_2,\rho,k),\sigma[k\mapsto\kappa]\rangle & \\
\langle (o\ E)^\ell,\rho,\kappa,\sigma\rangle & \longmapsto & \langle E,\rho,\mathsf{op}^\ell(o,k),\sigma[k\mapsto\kappa]\rangle & \\
\langle (o\ E\ F)^\ell,\rho,\kappa,\sigma\rangle & \longmapsto & \langle E,\rho,\mathsf{opl}^\ell(o,F,\rho,k),\sigma[k\mapsto\kappa]\rangle & \\
\langle x,\rho,\kappa,\sigma\rangle & \longmapsto & \langle V,\varrho,\kappa,\sigma\rangle & \text{if } (V,\varrho)\in\sigma(\rho(x)) \\
\langle V,\rho,\mathsf{ar}^\ell(E,\varrho,k),\sigma\rangle & \longmapsto & \langle E,\varrho,\mathsf{fn}^\ell(V,\rho,k),\sigma[k\mapsto\kappa]\rangle & \\
\langle V,\rho,\mathsf{fn}^\ell((\lambda_y x.E),\varrho,k),\sigma\rangle & \longmapsto & \langle E,\varrho[x\mapsto a,y\mapsto b],\kappa,\sigma[a\mapsto(V,\rho),b\mapsto((\lambda_y x.E),\varrho)]\rangle & \\
\langle V,\rho,\mathsf{fn}^\ell(U,\varrho,k),\sigma\rangle & \longmapsto & \langle \mathtt{blame}^\ell_\Lambda,\emptyset,\mathsf{mt},\emptyset\rangle & \text{if } \delta(\mathtt{proc?},U)\ni\mathtt{\#f} \\
\langle V,\rho,\mathsf{if}(E,F,\varrho,k),\sigma\rangle & \longmapsto & \langle E,\varrho,\kappa,\sigma\rangle & \text{if } \delta(\mathtt{false?},V)\ni\mathtt{\#f} \\
\langle V,\rho,\mathsf{if}(E,F,\varrho,k),\sigma\rangle & \longmapsto & \langle F,\varrho,\kappa,\sigma\rangle & \text{if } \delta(\mathtt{false?},V)\ni\mathtt{\#t} \\
\langle V,\rho,\mathsf{op}^\ell(o,a),\sigma\rangle & \longmapsto & \langle A,\emptyset,\kappa,\sigma\rangle & \text{if } \delta(o^\ell,V)\ni A \\
\langle ((U,\varrho),(V,\rho)),\emptyset,\mathsf{op}(\mathtt{car},a),\sigma\rangle & \longmapsto & \langle U,\varrho,\kappa,\sigma\rangle & \\
\langle ((U,\varrho),(V,\rho)),\emptyset,\mathsf{op}(\mathtt{cdr},a),\sigma\rangle & \longmapsto & \langle V,\rho,\kappa,\sigma\rangle & \\
\langle V,\rho,\mathsf{opl}^\ell(o,E,\varrho,k),\sigma\rangle & \longmapsto & \langle E,\varrho,\mathsf{opr}^\ell(o,V,\rho,k),\sigma\rangle & \\
\langle V,\rho,\mathsf{opr}^\ell(\mathtt{cons},U,\varrho,a),\sigma\rangle & \longmapsto & \langle ((U,\varrho),(V,\rho)),\emptyset,\kappa,\sigma\rangle & \\
\langle V,\rho,\mathsf{opr}^\ell(o,U,\varrho,a),\sigma\rangle & \longmapsto & \langle A,\emptyset,\kappa,\sigma\rangle & \\
\langle \mathtt{blame}^\ell_{\ell'},\rho,\kappa,\sigma\rangle & \longmapsto & \langle \mathtt{blame}^\ell_{\ell'},\emptyset,\mathsf{mt},\emptyset\rangle &
\end{array}$$

Module references

$$\begin{array}{llll}
\langle f^f,\rho,\kappa,\sigma\rangle & \longmapsto & \langle V,\emptyset,\kappa,\sigma\rangle & \text{if } (f^f,V)\in\widehat{\Delta}(\boldsymbol{M}) \\
\langle f^g,\rho,\kappa,\sigma\rangle & \longmapsto & \langle V,\emptyset,\mathsf{chk}^{f,g}_h(C,\emptyset,k),\sigma[k\mapsto\kappa]\rangle & \text{if } (f^f,(C\Leftarrow^{f,g}_h V))\in\widehat{\Delta}(\boldsymbol{M})
\end{array}$$

Contract checking

$$\begin{array}{llll}
\langle (C\Leftarrow^{f,g}_h E),\rho,\kappa,\sigma\rangle & \longmapsto & \langle E,\rho,\mathsf{chk}^{f,g}_h(C,\rho,k),\sigma[k\mapsto\kappa]\rangle & \\
\langle V,\rho,\mathsf{chk}^{f,g}_h(C,\varrho,k),\sigma\rangle & \longmapsto & \langle V,\rho,\mathsf{fn}(U,\varrho,k'),\sigma[k'\mapsto\mathsf{if}(V/\{C\},\mathtt{blame}^f_g,\rho,k)]\rangle & \\
 & & \quad\text{where } C \text{ is flat and } U=\textsc{fc}(C,V) & \\
\langle V,\rho,\mathsf{fn}^\ell(((C\dashrightarrow x.D)\Leftarrow^{f,g}_h a),\varrho,k),\sigma\rangle & \longmapsto & \langle V,\rho,\mathsf{chk}^{g,f}_h(C,\varrho,k'),\sigma[k'\mapsto\mathsf{fn}^\ell(U,\varrho',k''), & \\
 & & \qquad\qquad\qquad\qquad k''\mapsto\mathsf{chk}^{f,g}_h(D,\varrho[x\mapsto b],k), & \\
 & & \qquad\qquad\qquad\qquad b\mapsto(V,\rho)]\rangle & \\
 & & \text{where } (U,\varrho')\in\sigma(a) & \\
\langle V,\rho,\mathsf{chk}^{f,g}_h(C\rightarrow x.D,\varrho,k),\sigma\rangle & \longmapsto & \langle (((C\dashrightarrow x.D)\Leftarrow^{f,g}_h a),\varrho,\iota,\sigma[a\mapsto(V,\rho)]\rangle & \text{if } \delta(\mathtt{proc?},V)\ni\mathtt{\#t} \\
\langle V,\rho,\mathsf{chk}^{f,g}_h(C\rightarrow x.D,\varrho,k),\sigma\rangle & \longmapsto & \langle \mathtt{blame}^f_h,\emptyset,\mathsf{mt},\emptyset\rangle & \text{if } \delta(\mathtt{proc?},V)\ni\mathtt{\#f} \\
\langle V,\rho,\mathsf{chk}^{f,g}_h(C\wedge D,\varrho,k),\sigma\rangle & \longmapsto & \langle V,\rho,\mathsf{chk}^{f,g}_h(C,\varrho,i),\sigma[i\mapsto\mathsf{chk}^{f,g}_h(D,\varrho,k)]\rangle & \\
\langle V,\rho,\mathsf{chk}^{f,g}_h(C\vee D,\varrho,k),\sigma\rangle & \longmapsto & \langle V,\rho,\mathsf{ar}(U,\varrho,i),\sigma[i\mapsto\mathsf{chk\text{-}or}^{f,g}_h(V,\rho,C\vee D,\varrho,k)]\rangle & \\
 & & \text{where } U=\textsc{fc}(C) & \\
\langle V,\rho,\mathsf{chk\text{-}or}^{f,g}_h(U,\varrho,C\vee D,\rho',k),\sigma\rangle & \longmapsto & \langle U/\{C\},\varrho,\kappa,\sigma\rangle & \text{if } \delta(\mathtt{false?},V)\ni\mathtt{\#f} \\
\langle V,\rho,\mathsf{chk\text{-}or}^{f,g}_h(U,\varrho,C\vee D,\rho',k),\sigma\rangle & \longmapsto & \langle U,\varrho,\mathsf{chk}^{f,g}_h(D,\rho',k),\sigma\rangle & \text{if } \delta(\mathtt{false?},V)\ni\mathtt{\#t}
\end{array}$$

Abstract values

$$\begin{array}{llll}
\langle V,\rho,\mathsf{fn}^\ell(\bullet/\mathcal{C},\varrho,k),\sigma\rangle & \longmapsto & \langle E,\rho,\mathsf{begin}(U,\varrho,k),\sigma\rangle & \text{if } \delta(\mathtt{proc?},\bullet/\mathcal{C})\ni\mathtt{\#t} \\
 & & \multicolumn{2}{l}{\text{where } E=\textsc{amb}(\{\mathtt{\#t},\textsc{demonic}(\bigwedge\textsc{dom}(\mathcal{C}),V)\}) \text{ and } U=\bullet/\textsc{rng}(\mathcal{C})} \\
\langle V,\rho,\mathsf{begin}(E,\varrho,k),\sigma\rangle & \longmapsto & \langle E,\varrho,\kappa,\sigma\rangle & \\
\langle \bullet/\mathcal{C}\uplus\{C_1\vee C_2\},\rho,\kappa,\sigma\rangle & \longmapsto & \langle \bullet/\mathcal{C}\cup\{C_i\},\rho,\kappa,\sigma\rangle & \\
\langle \bullet/\mathcal{C}\uplus\{\mu x.C\},\rho,\kappa,\sigma\rangle & \longmapsto & \langle \bullet/\mathcal{C}\uplus\{[\mu x.C/x]C\},\rho,\kappa,\sigma\rangle &
\end{array}$$

Higher-order pair contract checking

$$\begin{array}{llll}
\langle V,\rho,\mathsf{chk}^{f,g}_h(\langle C,D\rangle,\varrho,k),\sigma\rangle & \longmapsto & \langle \mathtt{blame}^f_h,\emptyset,\mathsf{mt},\emptyset\rangle & \text{if } \delta(\mathtt{cons?},V)\ni\mathtt{\#f} \\
\langle V,\rho,\mathsf{chk}^{f,g}_h(\langle C,D\rangle,\varrho,k),\sigma\rangle & \longmapsto & \langle U,\rho,\mathsf{op}(\mathtt{car},i),\sigma[i\mapsto\mathsf{chk}^{f,g}_h(C,\varrho,k'),k'\mapsto\mathsf{chk\text{-}cons}^{f,g}_h(D,\varrho,U,\rho,k)]\rangle & \\
 & & \text{if } \delta(\mathtt{cons?},V)\ni\mathtt{\#t},\ \text{where } U=V/\{\lfloor\mathtt{cons?}\rfloor\} & \\
\langle V,\rho,\mathsf{chk\text{-}cons}^{f,g}_h(C,\varrho,U,\rho',k),\sigma\rangle & \longmapsto & \langle U,\rho',\mathsf{op}(\mathtt{cdr},i),\sigma[i\mapsto\mathsf{chk}^{f,g}_h(C,\varrho,k'),k'\mapsto\mathsf{opr}(\mathtt{cons},V,\rho,k)]\rangle &
\end{array}$$