Modular Analysis via Abstract Reduction Semantics

Sam Tobin-Hochstadt and David Van Horn*

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

Abstract. Modular static analysis requires treating some portion of the program opaquely. To enable such analysis, we introduce a notion of *abstract* reduction semantics. Opaque components are approximated by their specifications, which in turn are treated as abstract values during reduction. We demonstrate the technique by applying it to two kinds of specifications for higher-order languages: types and first-class contracts, showing that each soundly approximates opaque components. Finally, we derive *modular* static analyzers from these semantics, soundly predicting evaluation, contract violations, and blame assignment.

1 Modules, Specifications, and Analysis

A modular analysis, in contrast to a whole-program analysis, approximates the behavior of some portion of a program without examining the remainder. Modularity is especially important for multi-language programs, where key modules are often written in low-level languages such as C, rendering them unsuitable for high-level analyses, and for distributed systems where modules may be loaded remotely, making them unavailable at analysis time. Even for single language programs, choosing how much to analyze often depends on the situation; powerful flow analyses produce precise results at the cost of significant computational resources. It is often prohibitive to analyze the whole program at once; further, different portions of the program may have different analysis needs. All of these concerns point to the necessity of flexible and modular static analyzers.

Despite its importance, informative modular analysis of higher-order languages is considered a formidable research problem. Most [23,22,1] simply treat any unanalyzed portion as a black hole with arbitrary possible behavior. To do better, others have employed type and contract abstractions to reason about opaque components [19,18,21]. Meunier et al., for example, uses higher-order behavorial contracts at module boundaries to reason modularly. Meunier's analysis is formulated as a constraint semantics with all contract checks at the top-level. The static semantics engineering effort of Meunier et al. is daunting and unfortunately goes wrong in multiple places. Meunier's dissertation [18] corrects some of the errors of the conference paper [19], but even the corrected system is

* The authors are supported by a grant from the Mozilla Foundation and NSF Grant 0937060 to the CRA for the Computing Innovation Fellows Project.

unsound in its treatment of opaque modules (see Sections 4 and 5). Moreover, the system has been abandoned; the formalism is simply too involved to verify or scale to richer language features.

We conjecture that the fundamental weakness of Meunier's work is that its abstract semantics are too far from the actual dynamic semantics of programs. The disparity introduces engineering hurdles that require effort to overcome, and is often where both soundness and precision are compromised.

Modularity need not be such a formdidable design task *if we employ static semantics that are closely coupled with the dynamic semantics of a language.* We enable this close correspondence between static and dynamic semantics by designing *abstract* reduction semantics that characterize the behavior of programs with opaque modules. We then apply existing techniques for deriving semantics-based program analyzers [24], transforming reduction semantics into their sound, computable approximation. This simple composition of techniques yields sound, modular, semantics-based analyzers.

Our approach scales from basic specifications in the form of simple types to first-class contracts on higher-order functions, where specifications are computed at run-time. Even in this setting, the correspondence between semantics and analysis is direct.

Contributions:

- We introduce the notion of specifications as values in reduction semantics, using both types and contracts as specifications.
- We present computable, sound, modular static analyzers for languages with types and contracts, derived directly from the semantics of the languages.
- We prove modularity results for our analyses; that is, the analysis correctly approximates the analyzed portion of the program regardless of what the remainder of the program may be.
- As part of our abstract semantics for a language with contracts, we give the first reduction semantics which maintains the same quantity of information about contract errors as existing implementations such as Racket [13].

Organization: In Section 2, we review the systematic derivation of static analyzers from reduction semantics, showing the technique for a simple language with first-order modules. In Section 3, we equip our language with types, present an abstract reduction semantics where types are values, and show how to derive a modular analysis from the abstract semantics. In Section 4, we repeat the derivation for a language with second-class contracts, showing an abstract reduction semantics as well as a modular analysis; we then extend the language to accommodate first-class contracts. In Section 5, we discuss related work, and conclude in Section 6.

Prerequisites and notation: We assume a basic familiarity with reduction semantics and abstract machines and refer the reader to *Semantics Engineering with PLT Redex* [10] for background, notation, and terminology.

2 Review: Whole-Program Analysis

In this section, we review the *abstracting abstract machines* technique for designing whole-program analyses [24]. The technique is a semantics-based approach to program analysis that enables a systematic derivation of whole-program abstract[1] interpreters from abstract[2] machines, which in turn can be derived systematically from various forms of semantics such as definitional interpreters, reduction semantics, and denotational semantics. We review this technique because this paper contributes a complimentary technique for abstract interpretation at the reduction semantics level. Composing these techniques, as we will see, enables semantics-based *modular* analyses to be derived that are easily shown to be sound and computable.

The approach of abstracting abstract machines is to first refactor an abstract machine, such as the CEK machine [9], to store-allocate bindings and continuations, and then *bound* the store to some finite size. This strategy ensures a finite state-space, making analysis decidable.

Since it is also well known how to derive abstract machines from reduction semantics [10,8], we see this approach as sitting at the end of a derivational pipeline going from high-level language specifications to low-level implementations and then to abstract interpretations. The designer focuses solely on language semantics and derives sound, computable, modular abstractions systematically.

We illustrate the technique by starting with the reduction semantics for a prototypical higher-order call-by-value language with modules. From the reduction semantics, we derive a CEK-style machine, which we then refactor and abstract to obtained an abstract interpreter parameterized by a choice of approximation.

2.1 From Reduction Semantics to Abstract Machines

The syntax of our language is:

$$
\begin{array}{lll}
P & ::= & \boldsymbol{M}E \\
M, N & ::= & (\mathsf{module}\ f\ T\ V) \\
E, F & ::= & x \mid f \mid V \mid (E\ E)^f \mid (\mathsf{if0}\ E\ E\ E) \mid (\mathsf{wrong}^f\ n) \\
V & ::= & n \mid (\lambda x.E) \\
T & ::= & T \to T \mid \mathsf{int}
\end{array}
$$

A program is a sequence of modules, which we write using vector notation $\boldsymbol{M}$, followed by a top-level expression E. A module consists of a single recursive definition ($\mathsf{module} f T V$), which defines f to be the value V with type T. Module-bound variables are in scope in all modules and the top-level expression. Without loss of generality, we assume all module- and λ-bound variables are distinct and often refer to modules by the name they bind. A type is either the integer type int, or a function type $T \to T$.

[1] *Abstract* in the sense of being a sound, computable approximation.
[2] *Abstract* in the sense of being an idealized low-level evaluator.

An expression is either a value, which includes integers n and functions $(\lambda x.E)$; a variable, which is either module-bound (f, g, h) or λ-bound (x, y, z); an application $(E\ E)$; a conditional test $(\mathsf{if0}\ E\ E\ E)$; or a *wrong* expression $(\mathsf{wrong}^f\ n)$, which is not written by the programmer directly, but may arise during evaluation to indicate error states.[3] In the case of our prototypical language, errors arise only from applying non-function values (i.e., numbers).

Application expressions are annotated with the module in which they occur: $(E\,E)^f$; we use † in place of f to indicate the application appears within the top-level expression. When a program goes wrong, i.e., when a number is applied, the module (or the top-level) that contains the erroneous application expression $(n\ V)^f$ is reported. Anticipating our later development, we say that module f broke its contract with the programming language and report the module at fault and the misused value. General contracts and blame assignment are the subject of Section 4.

Computation is characterized by the $\mathbf{v}$ notion of reduction:

$$\begin{array}{rcl}((\lambda x.E)\ V)^f & \mathbf{v} & [V/x]E \\ (n\ V)^f & \mathbf{v} & (\mathsf{wrong}^f\ n) \\ (\mathsf{if0}\ 0\ E_1\ E_2) & \mathbf{v} & E_1 \\ (\mathsf{if0}\ V\ E_1\ E_2) & \mathbf{v} & E_2 \text{ if } V \neq 0,\end{array}$$

where $[V/x]E$ denotes the capture-avoiding substitution of V for x in E. The first case is the standard β_v rule for function application. The second case handles non-function application, which goes wrong—this case is included to make explicit what is ruled out by the type system. The third and fourth cases handle conditionals in the usual way.

We can now define single-step evaluation for programs by decomposing expressions into an evaluation context,

$$\mathcal{E} ::= [\,] \mid (\mathcal{E}\ E) \mid (V\ \mathcal{E}) \mid (\mathsf{if0}\ \mathcal{E}\ E\ E),$$

and a redex; contracting the redex according to the reduction relation; and plugging the contractum into the context.

To handle references to module-bound variables, we define a module environment that describes the module context $\boldsymbol{M}$ of a top-level expression:

$$\Delta_{\boldsymbol{M}} = \{(f, V) \mid (\mathsf{module}\ f\ T\ V) \in \boldsymbol{M}\}.$$

The module environment for a program can double as a notion of reduction and we now define an evaluation function for our language by the union of the $\mathbf{v}$ and $\Delta_{\boldsymbol{M}}$ notions of reductions. Evaluation is defined as the set of reachable machine states, giving us an intensional view of machines:

$$\mathit{eval}_{\mathbf{v}}(\boldsymbol{M}E) = \{E' \mid E \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}\Delta_M} E'\},$$

where $\longmapsto\!\!\!\!\rightarrow_{\mathbf{v}\Delta_M}$ is the reflexive, transitive closure of the step relation $\longmapsto_{\mathbf{v}\Delta_M}$:

$$\mathcal{E}[E] \longmapsto_{\mathbf{v}\Delta_M} \mathcal{E}[E'] \text{ if } E\ \mathbf{v}\Delta_{\boldsymbol{M}}\ E'.$$

[3] Gray indicates indicates a grammatical production not available to the programmer.

$\varsigma \longmapsto^{\mathrm{cek}}_{\mathbf{v}} \varsigma'$	
$\langle V, \rho, \mathsf{fn}^f((\lambda x.E), \rho', \kappa)\rangle$	$\langle E, \rho'[x \mapsto \langle V, \rho\rangle], \kappa\rangle$
$\langle V, \rho, \mathsf{fn}^f(n, \rho', \kappa)\rangle$	$\langle(\mathsf{wrong}^f\ n), \rho, \kappa\rangle$
$\langle 0, \rho, \mathsf{if}(E_0, E_1, \rho', \kappa)\rangle$	$\langle E_0, \rho', \kappa\rangle$
$\langle V, \rho, \mathsf{if}(E_0, E_1, \rho', \kappa)\rangle$	$\langle E_1, \rho', \kappa\rangle$, if $V \neq 0$
$\langle x, \rho, \kappa\rangle$	$\langle V, \rho', \kappa\rangle$ if $\rho(x) = \langle V, \rho'\rangle$
$\langle f, \rho, \kappa\rangle$	$\langle V, \emptyset, \kappa\rangle$, if $\Delta_{\boldsymbol{M}}(f) = V$
$\langle(E_0\ E_1)^f, \rho, \kappa\rangle$	$\langle E_0, \rho, \mathsf{ar}^f(E_1, \rho, \kappa)\rangle$
$\langle V, \rho, \mathsf{ar}^f(E, \rho', \kappa)\rangle$	$\langle E, \rho', \mathsf{fn}^f(V, \rho, \kappa)\rangle$
$\langle(\mathsf{if0}\ E_1\ E_2\ E_3), \rho, \kappa\rangle$	$\langle E_1, \rho, \mathsf{if}(E_2, E_3, \rho, \kappa)\rangle$

Fig. 1. The CEK machine.

This evaluation function effectively specifies the meaning of programs in our language, but it does not immediately admit an efficient implementation since it must partition the whole program into an evaluation context and redex *on each step*. However, more realistic abstract machines, such as the CEK machine [10], can be derived systematically from the above reduction semantics [5].

The CEK machine is environment-based; it uses environments and closures to model substitution. It represents evaluation contexts as *continuations*, an inductive data structure that models contexts in an inside-out manner. The key ideas of machines such as the CEK machine are that 1) the whole program need not be traversed to find the next redex; the machine integrates the process of plugging a contractum into a context and finding the next redex, and 2) subterms need not be traversed to implement substitution; the machine substitutes values for variables lazily by using an environment.

States of the CEK machine consist of a control string E, an environment ρ that closes the control string, and a continuation κ; states are identified up to consistent renaming of bound variables. Environments are finite maps from variables to closures. Environment extension is written $\rho[x \mapsto \langle V, \rho'\rangle]$. Evaluation contexts $\mathcal{E}$ are represented (inside-out) by continuations as follows:

- mt stands for $[\,]$;
- $\mathsf{ar}^f(E', \rho, \kappa)$ stands for $\mathcal{E}[([\,]\ E)^f]$ with E' closed by ρ for E, and κ for $\mathcal{E}$;
- $\mathsf{fn}^f(V', \rho, \kappa)$ stands for $\mathcal{E}[(V\ [\,])^f]$ with V' closed by ρ for V and κ for $\mathcal{E}$.

The transition system for the CEK machine is given in Figure 1. The first four cases implement the $\mathbf{v}$ notion of reduction, using environments to lazily represent substitution. The next case resolves variable references, effectively forcing the delayed substitution represented by the environment. The next case implements the $\Delta_{\boldsymbol{M}}$ notion of reduction for module references. The last three cases implement transitions that search for the next redex.

We can define an evaluation function in terms of the machine:

$$\mathit{eval}^{\mathrm{cek}}_{\mathbf{v}}(\boldsymbol{M}E) = \{\mathcal{U}(\varsigma) \mid \langle E, \emptyset, \mathsf{mt}\rangle \longmapsto\!\!\!\!\rightarrow^{\mathrm{cek}}_{\mathbf{v}\Delta_{\boldsymbol{M}}} \varsigma\},$$

where $\mathcal{U}$ unloads a machine state to a closed term (defined in Appendix D).

Theorem 1 (Felleisen [9]). $eval^{\text{cek}}_{\mathbf{v}} = eval_{\mathbf{v}}$.

2.2 From Abstract Machines to Abstract Abstract Machines

From the CEK machine, we can further derive machines that finitely approximate the behavior of the CEK machine (and therefore the evaluation function). We extend the machine with a store component and refactor to:

1. store-allocate bindings, obtaining the CESK machine, and
2. store-allocate continuations, obtaining the CESK* machine.

The CESK* machine easily abstracts by bounding the store to some finite size. To maintain soundness store updates are interpreted as joins and stores map addresses to *sets* of values.

$\hat{\varsigma} \longmapsto^{\text{cesk}^\star}_{\mathbf{v}} \hat{\varsigma}'$ where $\kappa \in \hat{\sigma}(a), b = \mathsf{alloc}(\hat{\varsigma}, \kappa)$	
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{fn}^f((\lambda x.E), \rho', a')$	$\langle E, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto \langle V, \rho\rangle], a'\rangle$
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{fn}^f(n, \rho', a')$	$\langle(\mathsf{wrong}^f\ n), \rho, \hat{\sigma}, a'\rangle$
$\langle 0, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{if}(E_0, E_1, \rho', a')$	$\langle E_0, \rho, \hat{\sigma}, a'\rangle$
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{if}(E_0, E_1, \rho', a')$	$\langle E_1, \rho, \hat{\sigma}, a'\rangle$ if $V \neq 0$
$\langle x, \rho, \hat{\sigma}, \kappa\rangle$	$\langle V, \rho', \hat{\sigma}, \kappa\rangle$ if $\hat{\sigma}(\rho(x)) \ni \langle V, \rho'\rangle)$
$\langle f, \rho, \hat{\sigma}, \kappa\rangle$	$\langle V, \emptyset, \hat{\sigma}, \kappa\rangle$ if $\Delta_{\boldsymbol{M}}(f) = V$
$\langle(E_0\ E_1), \rho, \hat{\sigma}, a\rangle$	$\langle E_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathsf{ar}^f(E_1, \rho, a)], b\rangle$
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{ar}^f(E, \rho', a')$	$\langle E, \rho', \hat{\sigma} \sqcup [b \mapsto \mathsf{fn}^f(V, \rho, a')], b\rangle$
$\langle(\mathsf{if0}\ E_1\ E_2\ E_3), \rho, \hat{\sigma}, a\rangle$	$\langle E_1, \rho, \hat{\sigma} \sqcup [b \mapsto \mathsf{if}(E_2, E_3, \rho, a)], b\rangle$

Fig. 2. The CESK* machine.

Figure 2 defines the CESK* machine. The transition system is parametrized by a function, alloc, which doles out addresses when something needs to be stored. If alloc always returns a fresh address, the machine operates in lock-step with the CEK machine, and is therefore equivalent to the evaluation function.

The analysis function is defined as:

$$\widehat{aval}^{\text{cek}}_{\mathbf{v}}(\boldsymbol{M}E) = \bigsqcup\{\widehat{\mathcal{U}}(\hat{\varsigma}) \mid \langle E, \emptyset, [a_0 \mapsto \mathsf{mt}], a_0\rangle \longmapsto\!\!\!\!\rightarrow^{\text{cesk}^\star}_{\mathbf{v}\Delta_{\boldsymbol{M}}} \hat{\varsigma}\},$$

where $\widehat{\mathcal{U}}$ unloads a machine state to a *set of* closed terms (defined in Appendix D).

On the other hand, if we require alloc to always allocate from some *finite* set of addresses, the state-space of the machine becomes finite. In this case, we can view each machine state of the CESK* machine as representing a set of CEK machine states by a straightforward structural map. The bounded CESK* computes a sound approximation to the reachable states of the CEK machine in the sense that if a CEK machine state ς is reachable, a CESK* machine $\hat{\varsigma}$ state is reachable such that ς is in the set of states represented by $\hat{\varsigma}$.

To distinguish these two cases, we will write $eval_{\mathbf{v}}^{\mathrm{cesk}^\star}$ when alloc is unbounded and always returns fresh addresses and $\widehat{aval}_{\mathbf{v}}^{\mathrm{cesk}^\star}$ when alloc is bounded.

Theorem 2 (Van Horn and Might [24]). *$\widehat{aval}_{\mathbf{v}}^{\mathrm{cesk}^\star}$ is a sound, computable approximation of $eval_{\mathbf{v}}$.*

3 Modular Analysis using Types

The approach of Section 2 is based on *whole-program evaluators*, and so naturally it yields *whole-program analyzers*. To modularize our analysis, we take the type specifications from modules as a specification language and develop a notion of reduction for types when considered as values, soundly approximating opaque modules with their types.

First, the set of values is extended with the set of *types*, which are a form of *abstract value*—they represent the set of all values of that type:

$$V ::= n \mid (\lambda x.E) \mid \overline{T}$$

The overline $\overline{T}$ indicates abstract values, distinguishing them from other values. Second, we define the $\widehat{\mathbf{v}}$ notion of reduction, which handles the cases of applying and testing abstract values:

$$\begin{array}{rcl}
(\overline{(T' \to T)}\ V)^f & \widehat{\mathbf{v}} & \overline{T} \\
(\overline{\mathsf{int}}\ V)^f & \widehat{\mathbf{v}} & (\mathsf{wrong}^f\ \overline{\mathsf{int}}) \\
(\mathsf{if0}\ \overline{\mathsf{int}}\ E_1\ E_2) & \widehat{\mathbf{v}} & E_1 \\
(\mathsf{if0}\ \overline{T}\ E_1\ E_2) & \widehat{\mathbf{v}} & E_2
\end{array}$$

The first case handles the application of a function type to a value, which produces the abstract value T, the range of the function. The second case handles non-function application, in particular the application of int to a value (the type system rules this case out). Finally, the last two cases handle conditional testing of an abstract value. Notice that if a value is abstracted by a function type, that value cannot possibly be 0, so only the alternative branch is taken. But if a value is abstracted by the $\overline{\mathsf{int}}$ type, it is not possible to determine which branch should be taken, hence they both are. Because of this, the $\widehat{\mathbf{v}}$ notion of reduction, and therefore the abstract machines derived from it, are non-deterministic.

Next, we extend the module language with *opaque* modules, i.e., modules whose implementation is missing at analysis time. Such missing components are approximated by their specification, in this case, their advertised type:

$$M ::= (\mathsf{module}\ f\ T\ V) \mid (\mathsf{module}\ f\ T\ \bullet).$$

We extend $\Delta_{\boldsymbol{M}}$ to resolve opaque module references to their type:

$$\Delta_{\boldsymbol{M}} = \{(f, V) \mid (\mathsf{module}\ f\ T\ V) \in \boldsymbol{M}\} \cup \{(f, \overline{T}) \mid (\mathsf{module}\ f\ T\ \bullet) \in \boldsymbol{M}\}.$$

$\varsigma \longmapsto^{\mathrm{cek}}_{\widehat{\mathbf{v}}} \varsigma'$	
$\langle V, \rho, \mathsf{fn}^f(\overline{(T' \to T)}, \emptyset, \kappa)\rangle$	$\langle \overline{T}, \emptyset, \kappa\rangle$
$\langle V, \rho, \mathsf{fn}^f(\overline{\mathsf{int}}, \emptyset, \kappa)\rangle$	$\langle(\mathsf{wrong}^f\ \overline{\mathsf{int}}), \rho, \kappa\rangle$
$\langle \overline{\mathsf{int}}, \rho, \mathsf{if}(E_1, E_2, \rho', \kappa)\rangle$	$\langle E_1, \rho', \kappa\rangle$
$\langle \overline{T}, \rho, \mathsf{if}(E_1, E_2, \rho', \kappa)\rangle$	$\langle E_2, \rho', \kappa\rangle$

$\hat{\varsigma} \longmapsto^{\mathrm{cesk}^\star}_{\widehat{\mathbf{v}}} \hat{\varsigma}'$ where $\kappa \in \hat{\sigma}(a)$	
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{fn}^f(\overline{(T' \to T)}, \rho', a')$	$\langle \overline{T}, \emptyset, \hat{\sigma}, a'\rangle$
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{fn}^f(\overline{\mathsf{int}}, \rho', a')$	$\langle(\mathsf{wrong}^f\ \overline{\mathsf{int}}), \rho, \hat{\sigma}, a'\rangle$
$\langle \overline{\mathsf{int}}, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{if}(E_1, E_2, \rho', a')$	$\langle E_1, \rho, \hat{\sigma}, a'\rangle$
$\langle \overline{T}, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{if}(E_1, E_2, \rho', a')$	$\langle E_2, \rho, \hat{\sigma}, a'\rangle$

Fig. 3. Additional CEK and CESK* machine transitions for types-as-values.

Finally, the evaluation function is defined as:

$$\mathit{eval}_{\widehat{\mathbf{v}}}(\boldsymbol{M}E) = \{E' \mid E \longmapsto\!\!\!\!\rightarrow_{\mathbf{v}\widehat{\mathbf{v}}\Delta_M} E'\}.$$

This reduction semantics and the evaluation function[4] it induces are an abstract interpretation soundly approximating a program for all well-typed instantiations of the opaque modules with particular implementations. However, the approximation is clearly not computable in general since the language is an *extension* of a Turing-complete language. Dynamic widening [7] could be employed by selectively widening a term to its type, however for the purpose of automatic program analysis we introduce an *additional abstraction* that ensures computability.

To achieve modularity, soundness *and* computability, we simply pump the *abstract* reduction semantics through the derivation pipeline outlined in Section 2. This recipe yields a CEK-style machine extended with the types-as-values notion of reduction. Just as the reduction semantics is an extension of $\mathbf{v}$, the resulting machine includes all of the CEK machine transitions in Figure 1. Because the syntax of evaluation contexts is unchanged, no additional search transitions are needed. Consequently, there are only four new machine transitions: one for each case of the $\widehat{\mathbf{v}}$ notion of reduction, shown in Figure 3 (top). Store-allocating bindings and continuations in these additional transitions then derives a CESK* machine with types-as-values, shown in Figure 3 (bottom).

The resulting evaluator, $\widehat{\mathit{aval}}^{\mathrm{cesk}^\star}_{\widehat{\mathbf{v}}}$, corresponds to the original evaluator. To prove this correspondence, we first define a type indexed relation on expressions $E \sqsubseteq_T E'$, which states that E' is an approximation of E, which both have type T. The approximation judgment is defined by a straightforward structural induction, with the exception that any expression can be approximated by its type and that opaque modules approximate their concrete counterparts:

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash E \sqsubseteq_T \overline{T}} \qquad \frac{\vdash E : T}{(\mathsf{module}\ f\ T\ E) \sqsubseteq_T (\mathsf{module}\ f\ T\ \bullet)}$$

[4] Although the $\widehat{\mathbf{v}}$ notion of reduction is non-deterministic, $\mathit{eval}_{\widehat{\mathbf{v}}}$ is a function since it includes all reachable states.

The remaining cases are deferred to Appendix B. Equipped with this definition, we can state the desired soundness theorem.

In this and subsequent theorems, we say that an abstract evaluator $\widehat{aval}$ is *sound* with respect to *eval* if every state in the concrete semantics is related by the appropriate $\sqsubseteq$ to some state in the abstract semantics. We will say that $\widehat{aval}$ is a *computable approximation* if its possible state space is finite; this is a stronger condition than required, but satisfied for the approximations we consider.

Theorem 3. *$\widehat{aval}^{\mathrm{cesk}^\star}_{\widehat{\mathbf{v}}}$ is a sound, computable approximation of $eval_{\mathbf{v}}$ for all possible instantiations of opaque modules.*

Proof. By Lemmas 1 and a subject reduction argument. □

Lemma 1. *If $E \longmapsto_{\mathbf{v}\widehat{\mathbf{v}}\Delta_M} E'$ and $\boldsymbol{M}E \sqsubseteq_T \boldsymbol{N}F$, then either $\boldsymbol{M}E' \sqsubseteq_T \boldsymbol{N}F$, or $\boldsymbol{M}E' \sqsubseteq_T \boldsymbol{N}F'$ and $F \longmapsto_{\mathbf{v}\widehat{\mathbf{v}}\Delta_N} F'$.*

Proof. If the terms decompose into related contexts and related redexes, the result follows by Lemma 2. Otherwise, the redex in E must be related to an abstract value in F, in which case the result follows by preservation of types. □

Lemma 2. *If $E\ \mathbf{v}\widehat{\mathbf{v}}\Delta_{\boldsymbol{M}}\ E'$ and $F\ \mathbf{v}\widehat{\mathbf{v}}\Delta_{\boldsymbol{N}}\ F'$ and $\boldsymbol{M}E \sqsubseteq_T \boldsymbol{N}F$ for some T, then $\boldsymbol{M}E' \sqsubseteq_T \boldsymbol{N}F'$.* □

4 Behavioral Contracts as Values

As a second example of specifications as values, we consider behavioral contracts.

Programs consist of multiple interacting components such as modules, and these components interact based on some mutually agreed upon contract that specifies the obligations and guarantees of values that are consumed and produced by each component. A contract system monitors the boundaries between components and checks that all contracts are satisfied, assigning blame to the appropriate component whenever a contract is violated.

In a first-order setting, where functions are not values, properly assessing blame at run-time is straightforward. However, matters are complicated when higher-order values such as functions or objects are included in the language. Findler and Felleisen [11] have established a semantic framework for properly assessing blame at run-time in a higher-order language, which forms the theoretical basis of the Racket contract system [13].

To illustrate, an example is given in Figure 4. The program consists of a module and top-level expression. We assume `even?` as primitive. Module `double` implements twice-iterated application on even numbers, consuming and producing functions on even numbers. The top-level expression makes use of the `double` function, but incorrectly—`double` is applied to a function that consumes even numbers, but produces an odd number. Contract checking and blame assignment in a higher-order program such as this is complicated by the fact that it is not decidable whether the argument of `double` is a function that consumes and produces even numbers. Higher-order contracts must instead be pushed down into

```
(module double
  (((pred even?) → (pred even?)) → ((pred even?) → (pred even?)))
  (λf.λx.f(fx)))
((double (λn.7)) 4)
```

```
the top-level broke the contract
(((pred even?) → (pred even?)) → ((pred even?) → (pred even?)))
on double; expected <even?>, given: 7
```

Fig. 4. A program with a higher-order contract failure.

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 it is the first-order `even?` predicate that witnesses the violation when `f` is applied to `x`.

In the remainder of this section, we derive a modular program analysis that soundly predicts contract violations and blame assignment as well as value flow, using contracts as abstract values to represent opaque components.

4.1 Reduction Semantics for Contracts

In this section, we give a reduction semantics for a prototypical higher-order, untyped, call-by-value language with modules and contracts. From the reduction semantics, we derive a CEK-style abstract machine, which we then refactor and abstract to obtained a parametrized abstract interpreter. In the following section, we develop a notion of reduction for *contracts-as-values* that leads to a modular static analysis of higher-order behavioral software contracts.

Our language is similar that of Section 2.1 but we we use behavioral contracts in place of types. The user-level syntax is changed as follows:

$$M ::= (\mathsf{module}\ f\ C\ V) \qquad C ::= \mathsf{int} \mid (C \to C) \mid (\mathsf{pred}\ (\lambda x.E)).$$

The int contract asserts a value is an integer; any asserts nothing; $(C \to C')$ asserts a value is a function accepting input satisfying C and producing values satisfying C'; finally, $(\mathsf{pred}\ (\lambda x.E))$ asserts the predicate encoded by the given function. We write any as a shorthand for $(\mathsf{pred}\,(\lambda x.0))$, the contract that always succeeds.

During reduction, we use an elaborated syntax with explicit contract checks, written $(C \Leftarrow^{f,g}_{V,f'} E)$, which checks that E produces a value satisfying C with f and g representing the two parties to the contract and f' as the module from which the original contract is taken (always either f or g). Blessed contracts, written $(C \dashrightarrow C')$, represent a function contract that has been partially verified, namely that the value is a function, but it remains to check the input satisfies C and output satisfies C' (in a higher-order setting, these checks *must* be delayed). Blame expressions, written $(\mathsf{blame}^f_g\ V\ C\ V')$, represent the information reported in the contract violation seen in Figure 4: it includes the blamed party f, the party whose contract was broken g, the (possibly) higher-order value that broke the contract V, the first-order contract check which fails C, and the

$$
\begin{array}{lll}
P & ::= & \boldsymbol{M}\, E \\
M, N & ::= & (\mathsf{module}\ f\ C\ V) \\
L & ::= & (\lambda x.E) \\
W & ::= & L \mid ((C \dashrightarrow C) \Leftarrow^{f,f}_{V,f} W) \\
V & ::= & n \mid W \\
B & ::= & (\mathsf{blame}^{f}_{f}\ V\ C\ V) \\
E, F & ::= & V \mid x \mid f^{f} \mid (E\ E)^{f} \mid (\mathsf{if0}\ E\ E\ E) \mid (C \Leftarrow^{f,f}_{V,f} E) \mid B \\
C & ::= & \mathsf{int} \mid (C \rightarrow C) \mid (C \dashrightarrow C) \mid (\mathsf{pred}\ L)
\end{array}
$$

Fig. 5. Syntax for programs with contracts.

$$
\begin{array}{rcl}
(((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V',h} W)\ V) & \mathbf{c} & (C_2 \Leftarrow^{f,g}_{V',h} (W\ (C_1 \Leftarrow^{g,f}_{V,h} V))) \\
(\mathsf{int} \Leftarrow^{f,g}_{V,f'} n) & \mathbf{c} & n \\
(\mathsf{int} \Leftarrow^{f,g}_{V,f'} W) & \mathbf{c} & (\mathsf{blame}^{f}_{f'}\ V\ \mathsf{int}\ W) \\
((C_1 \rightarrow C_2) \Leftarrow^{f,g}_{V,f'} W) & \mathbf{c} & ((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V,f'} W) \\
((C_1 \rightarrow C_2) \Leftarrow^{f,g}_{V,f'} n) & \mathbf{c} & (\mathsf{blame}^{f}_{f'}\ V\ (C_1 \rightarrow C_2)\ n) \\
((\mathsf{pred}\ L) \Leftarrow^{f,g}_{V',f'} V) & \mathbf{c} & (\mathsf{if0}\ (L\ V)\ V\ (\mathsf{blame}^{f}_{f'}\ V'\ (\mathsf{pred}\ L)\ V))
\end{array}
$$

Fig. 6. Notions of reduction for contract checking.

first-order value that witnesses the violation V'. Previous reduction semantics for contracts [11,19,14] have reported only the party that broke the contract, whereas our semantics presents the same information available in production systems. In a static analysis context, this is essential, since reporting only the failing module hides imprecision in the analysis.

The extensions to the original grammar of modular programs are presented in Figure 5. To support the evaluation of contracts, we require additional run-time syntax, presented with a gray background. We require predicate contracts to be syntactic values so as to avoid questions about when they are evaluated. In this more general context, we can consider our earlier uses of $(\mathsf{wrong}^{f}\ V)$ as syntactic sugar for $(\mathsf{blame}^{f}_{\Lambda}\ V\ \lambda\ V)$, with Λ representing the language implementation, and λ as the implicit contract between the programmer and the language.

Values now include "wrapped" functions W, which are either functions or wrapped functions with a blessed arrow contract check around it. In addition to application expressions, we annotate module references with the module in which they appear; this is important for correctly wrapping contract checks when resolving module references.

Figure 6 defines the notions of reduction for checking contracts. The first case handles the application of a checked function to a value. Notice that the blessed contract check is re-written to lower level checks on the input and output of the application; also notice that the blame annotations are swapped on the checking of the input. This contravariance in domain checks corresponds to the Findler and Felleisen treatment of blame in higher-order contracts [11]. The second and third case handle first-order checks of base values that succeed. The fourth handles first-order checks that fail. The fifth and sixth case handle checking for a function, producing a blessed contract in the case of success and blame in the case of failure. Finally, the last case handles predicate checking by reducing to

a conditional testing of the predicate, branching to the checked value when it holds and blame when it does not.

The grammar of evaluation contexts is given by:

$$\mathcal{E} ::= [\,] \mid (\mathcal{E}\ E) \mid (V\ \mathcal{E}) \mid (\mathsf{if0}\ \mathcal{E}\ E\ E) \mid (C \Leftarrow^{f,f}_{V,f} \mathcal{E}).$$

To handle references to module-bound variables, we define a module environment, like that in Section 2.1, 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" [11]:

$$\begin{aligned}\Delta_{\boldsymbol{M}} = &\{(f^f, V) \mid (\mathsf{module}\ f\ C\ V) \in \boldsymbol{M}\} \\ &\cup \{(f^g, (C \Leftarrow^{f,g}_{V,f} V)) \mid (\mathsf{module}\ f\ C\ V) \in \boldsymbol{M}, f \neq g\}.\end{aligned}$$

An evaluation function for our language is defined as follows:

$$eval_{\mathbf{c}}(\boldsymbol{M}E) = \{E' \mid E \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{vc}\Delta_{\boldsymbol{M}}} E'\}.$$

The grammar of evaluation contexts is just the same as in Section 2.1 with the additional contract checking context. Hence deriving a CEK-like machine is straightforward: we have one additional continuation form and a machine transition that searches for a redex under a contract check and one additional transition for each case of the **c** notion of reduction.

The resulting machine to be easily proved correct with respect to the reduction semantics and applying the machine refactorings and abstraction described in Section 2.2 should yield a parametrized whole-program analysis for higher-order behavioral software contracts.

Now we turn our attention to modularity, which we achieve by a contracts-as-values notion of reduction to reason about opaque components using their behavioral specification.

4.2 A Contracts-as-Values Notion of Reduction

In this section, we develop a modular analysis based on contracts by first describing how contracts behave as values. To do so, we extend the language of values with contracts and define a non-deterministic reduction semantics. To avoid confusion with contracts as first-class values, the subject of Section 4.4, we maintain the convention of using an overline for abstract values.

This reduction semantics with contracts as values is again a non-computable abstract interpretation. Computability is regained in the exactly the same manner as before by deriving an abstract machine with a bounded store.

The grammar extensions (written +=) for the language with contracts are:

$$W \mathrel{+}= \overline{(C \to C)} \mid \overline{(\mathsf{pred}\ L)} \qquad\qquad V \mathrel{+}= \overline{C}$$

$$
\begin{array}{rcll}
(\overline{(\mathsf{pred}\ L)}\ V)^f & \widehat{\mathbf{c}} & \overline{\mathsf{any}} & \\
(\overline{(\mathsf{pred}\ L)}\ V)^f & \widehat{\mathbf{c}} & (\mathsf{blame}^{f_?}_{g_?}\ V_?\ C_?\ V'_?) & (*) \\
(\overline{(C_1 \to C_2)}\ V)^f & \widehat{\mathbf{c}} & \overline{C_2} & \\
(\overline{(C_1 \to C_2)}\ V)^f & \widehat{\mathbf{c}} & (\mathsf{blame}^{f_?}_{g_?}\ V_?\ C_?\ V'_?) & (*) \\
(\overline{\mathsf{int}}\ V)^f & \widehat{\mathbf{c}} & (\mathsf{wrong}^f\ \overline{\mathsf{int}}) & \\
(\mathsf{if0}\ \overline{C}\ E_1\ E_2) & \widehat{\mathbf{c}} & E_2 & \\
(\mathsf{if0}\ \overline{\mathsf{int}}\ E_1\ E_2) & \widehat{\mathbf{c}} & E_1 & \\
(\mathsf{if0}\ \overline{(\mathsf{pred}\ L)}\ E_1\ E_2) & \widehat{\mathbf{c}} & E_1 & \\
(\mathsf{int} \Leftarrow^{f,g}_{V,f'} \overline{\mathsf{int}}) & \widehat{\mathbf{c}} & \overline{\mathsf{int}} & \\
(\mathsf{int} \Leftarrow^{f,g}_{V,f'} \overline{(\mathsf{pred}\ L)}) & \widehat{\mathbf{c}} & \overline{\mathsf{int}} & \\
(\mathsf{int} \Leftarrow^{f,g}_{V,f'} \overline{(\mathsf{pred}\ L)}) & \widehat{\mathbf{c}} & (\mathsf{blame}^f_{f'}\ V\ \mathsf{int}\ \overline{(\mathsf{pred}\ L)}) & \\
((C_1 \to C_2) \Leftarrow^{f,g}_{V,f'} \overline{(\mathsf{pred}\ L)}) & \widehat{\mathbf{c}} & \overline{(C_1 \to C_2)} & \\
((C_1 \to C_2) \Leftarrow^{f,g}_{V,f'} \overline{(\mathsf{pred}\ L)}) & \widehat{\mathbf{c}} & (\mathsf{blame}^f_{f'}\ V\ (C_1 \to C_2)\ \overline{(\mathsf{pred}\ L)}) & \\
((C_1 \to C_2) \Leftarrow^{f,g}_{V,f'} \overline{\mathsf{int}}) & \widehat{\mathbf{c}} & (\mathsf{blame}^f_{f'}\ V\ (C_1 \to C_2)\ \overline{\mathsf{int}}) &
\end{array}
$$

Fig. 7. Reduction with contracts as abstract values

```
(module f (any → (any → any)) (λx.x))
(module g ((pred (λx.x)) → int) (λx.0))
(module h any (λz.((f g) 8)))
(h 0)
```

Fig. 8. Example of indirect blame.

Again, overlines $\overline{C}$ distinguish abstract values from other values and contracts.

Analogous to the way in which types could be considered values by notions of reductions, we likewise can develop notions of reduction for contracts as abstract values. The contract-as-values notion of reduction $\widehat{\mathbf{c}}$ is given in Figure 7. In this figure, $f_?$, $V_?$, and $C_?$ are interpreted as ranging over all modules, values, and contracts that are not opaque.

The reduction rules marked with $(*)$ require explanation. They indicate that when an unknown function, approximated by a contract, is applied, it may have arbitrary behavior at runtime, including blaming any module in the program. Consider the example in Figure 8. Here, if `f` and `g` are opaque, the semantics reduces (`f g`) to $\overline{(\mathsf{any} \to \mathsf{any})}$. Applying this value to `8` reduces to the abstract value $\overline{\mathsf{any}}$, but it also performs computation; in this example, that computation blames `h`, but it might blame any module in the program. To represent this possibility, we add the two rules marked with $(*)$.

Additionally, note that when an abstract value approximated by a predicate contract is *applied* to a value $(\overline{(\mathsf{pred}\ L)}\ V)$ the result may be any value; this course approximation is necessary since it is possible to write a trivial predicate that simulate `any`. In contrast, *checking* a predicate contract against a value $((\mathsf{pred}\ L) \Leftarrow^{f,g}_{V',f'} V)$ is still handled precisely by the last rule in Figure 6.

$$g \sqsubseteq f^? \qquad V \sqsubseteq V^? \qquad C \sqsubseteq C^? \qquad V \sqsubseteq \overline{(\mathsf{pred}\ L)} \qquad (C \Leftarrow^{f,g}_{V,f'} E) \sqsubseteq \overline{C} \qquad n \sqsubseteq \overline{\mathsf{int}}$$

$$((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V,f'} E) \sqsubseteq \overline{(C_1 \to C_2)} \qquad \frac{(\mathsf{module}\ f\ C\ \bullet) \in \boldsymbol{M}}{(\mathsf{blame}^f_g\ V\ C'\ V') \sqsubseteq E}$$

Fig. 9. The $\sqsubseteq$ relation for contracts (excerpt).

The $\Delta_{\boldsymbol{M}}$ relation continues to implement the notion of module reference, but now it is extended to produce abstract contract values for opaque modules.

$$\begin{aligned}\Delta_{\boldsymbol{M}} = &\{(f^g, \overline{C}) \mid (\mathsf{module}\ f\ C\ \bullet) \in \boldsymbol{M}\} \\ &\cup \{(f^f, V) \mid (\mathsf{module}\ f\ C\ V) \in \boldsymbol{M}\} \\ &\cup \{(f^g, (C \Leftarrow^{f,g}_{V,f} V)) \mid (\mathsf{module}\ f\ C\ V) \in \boldsymbol{M}, f \neq g\}\end{aligned}$$

An evaluation function is defined as follows:

$$\mathit{eval}_{\widehat{\mathbf{c}}}(\boldsymbol{M}E) = \{E' \mid E \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{vc\widehat{c}}\Delta_{\boldsymbol{M}}} E'\}.$$

The following lemmas, which are analogous to those of Section 3, establish the soundness of the abstract notion of reduction with respect to the $\sqsubseteq$ relation defined in Figure 9 (again structural rules are deferred to Appendix C).

Lemma 3. *If* $E \longmapsto_{\mathbf{vc\widehat{c}}\Delta_{\boldsymbol{M}}} E'$ *and* $\boldsymbol{M}E \sqsubseteq \boldsymbol{N}F$*, then either* $\boldsymbol{M}E' \sqsubseteq \boldsymbol{N}F$*, or* $\boldsymbol{M}E' \sqsubseteq \boldsymbol{N}F'$ *and* $F \longmapsto_{\mathbf{vc\widehat{c}}\Delta_{\boldsymbol{N}}} F'$*.*

Proof. If the terms decompose into related contexts and related redexes, the result follows by Lemma 4. Otherwise, the redex in E must be related to an abstract value in F, in which case the result follows by Lemma 5.

Lemma 4. *If* $E\ \mathbf{v}\widehat{\mathbf{v}}\Delta_{\boldsymbol{M}}\ E'$*,* $F\ \mathbf{v}\widehat{\mathbf{v}}\Delta_{\boldsymbol{N}}\ F'$*, and* $\boldsymbol{M}E \sqsubseteq \boldsymbol{N}F$*, then* $\boldsymbol{M}E' \sqsubseteq \boldsymbol{N}F'$*.*

Lemma 5. *If* $E \longmapsto_{\mathbf{vc\widehat{c}}\Delta_{\boldsymbol{M}}} E'$ *and* $\boldsymbol{M}E \sqsubseteq \boldsymbol{N}\overline{C}$ *then* $\boldsymbol{M}E' \sqsubseteq \boldsymbol{N}\overline{C}$*.*

4.3 Deriving An Abstract Interpretation for Blame Analysis

In this section, we derive an abstract machine for the annotated language and prove it faithful to the reduction semantics. Continuations are represented as:

$$\kappa ::= \mathsf{mt} \mid \mathsf{ar}^f(E, \rho, \kappa) \mid \mathsf{fn}^f(W, \rho, \kappa) \mid \mathsf{ck}^{f,f}_{V,f}(C, \kappa) \mid \mathsf{if}(E, E, \rho, \kappa)$$

Equipped with this definition, we extend the CEK machine with rules for contract evaluation presented in Figure 10 as well as rules for handling abstract values. This extension presents no complications and so for reasons of space we defer the definition to Appendix A. Core expression evaluation and module evaluation is unchanged, inheriting the new definition of $\Delta_{\boldsymbol{M}}$ as appropriate.

Given this machine, we can now define a corresponding $\mathit{eval}^{\mathrm{cek}}_{\widehat{\mathbf{c}}}$ function:

$$\mathit{eval}^{\mathrm{cek}}_{\widehat{\mathbf{c}}}(\boldsymbol{M}E) = \{\mathcal{U}(\varsigma) \mid \langle E, \emptyset, \mathsf{mt}\rangle \longmapsto\!\!\!\!\!\rightarrow^{\mathrm{cek}}_{\mathbf{vc\widehat{c}}\Delta_{\boldsymbol{M}}} \varsigma\}.$$

$\varsigma \longmapsto^{\text{cek}}_{\mathbf{c}} \varsigma'$	
$\langle (C \Leftarrow^{f,g}_{V,f'} E), \rho, \kappa \rangle$	$\langle E, \rho, \mathsf{ck}^{f,g}_{V,f'}(C, \kappa) \rangle$
$\langle n, \rho, \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, \kappa) \rangle$	$\langle n, \rho, \kappa \rangle$
$\langle W, \rho, \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, \kappa) \rangle$	$\langle (\mathsf{blame}^{f}_{f'}\ V\ \mathsf{int}\ W), \rho, \kappa \rangle$
$\langle W, \rho, \mathsf{ck}^{f,g}_{V,f'}((C_1 \rightarrow C_2), \kappa) \rangle$	$\langle ((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V,f'} W), \rho, \kappa \rangle$
$\langle n, \rho, \mathsf{ck}^{f,g}_{V,f'}((C_1 \rightarrow C_2), \kappa) \rangle$	$\langle (\mathsf{blame}^{f}_{f'}\ V\ (C_1 \rightarrow C_2)\ n), \rho, \kappa \rangle$
$\langle V, \rho, \mathsf{ck}^{f,g}_{V',h}((\mathsf{pred}\ L), \kappa) \rangle$	$\langle (L\ V), \rho, \mathsf{if}(V, (\mathsf{blame}^{f}_{h}\ V'\ (\mathsf{pred}\ L)\ V), \rho, \kappa) \rangle$
$\langle V, \rho, \mathsf{fn}(((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V',h} W), \rho', \kappa) \rangle$	$\langle (C_1 \Leftarrow^{g,f}_{V,h} V), \rho, \mathsf{fn}(W, \rho', \mathsf{ck}^{f,g}_{V',h}(C_2, \kappa)) \rangle$

Fig. 10. Additional CEK machine transitions for contracts.

Given the above CEK machine, we follow the recipe presented in Section 2 to systematically transform the rules for contract evaluation into new rules for a CESK* machine. By then bounding the store, we obtain a sound and computable abstraction in the form of an $\widehat{aval}^{\text{cesk}^\star}_{\widehat{\mathbf{c}}}$. The derivation presents no complications and so, again, for reasons of space, the machine transitions and the $\widehat{aval}^{\text{cesk}^\star}_{\widehat{\mathbf{c}}}$ function are omitted and given in Appendix A.

Theorem 4. *$\widehat{aval}^{\text{cesk}^\star}_{\widehat{\mathbf{c}}}$ is a sound, computable approximation of $eval_{\mathbf{c}}$ for all possible instantiations of opaque modules.*

The theorem relies on the soundness of the abstract notion of reduction, established in Lemmas 3 and 4.

4.4 First-class Contracts

The above development treats contracts purely syntactically. However, since contracts are checked at run-time, they can also be *computed* at run-time. In fact, existing higher-order contract systems [13] make contracts first-class values which are dynamically computed before checking. In this section, we show the simple extensions required to express first-class contracts in our model, as well as the changes this implies for the abstractions we have developed.

Syntax: Allowing contracts to be computed means that expressions may appear and be evaluated inside contracts. Allowing useful programming over contracts requires that contracts become part of the rest of the expression language. This requires extensions as follows:

$$E \mathrel{+}= (E \rightarrow E) \qquad V \mathrel{+}= C \qquad \mathcal{E} \mathrel{+}= (\mathcal{E} \Leftarrow^{f,f}_{V,f} E) \mid (\mathcal{E} \rightarrow E) \mid (V \rightarrow \mathcal{E}),$$

In particular, we add a new definition of contract values. Evaluation occurs during the construction of function contracts, and contracts are evaluated when they appear in checks.

Semantics: We modify the definition of $\Delta_{\boldsymbol{M}}$ to:

$$\begin{aligned}\Delta_{\boldsymbol{M}} = &\{(f^g, (C \Leftarrow^{f,g}_{\bullet,f} \bullet)) \mid (\mathsf{module}\ f\ C\ \bullet) \in \boldsymbol{M}\} \\ &\cup \{(f^f, V) \mid (\mathsf{module}\ f\ C\ V) \in \boldsymbol{M}\} \\ &\cup \{(f^g, (C \Leftarrow^{f,g}_{V,f} V)) \mid (\mathsf{module}\ f\ C\ V) \in \boldsymbol{M}, f \neq g\}\end{aligned}$$

where $\bullet$ is a placeholder value. Opaque module references produce contract check expressions, which then evaluate the contract.

The rules for checking values against contracts remain the same. When a contract is checked against a placeholder, the contract transitions from a concrete value (representing just the contract) to an abstract value (representing all values that may satisfy the contract). If a non-contract value is used as a contract, the program errors since the implicit contract with the programming language has been violated. If an abstract value is used as a contract, there are two possibilities—it may evaluate to an unknown contract, in which case we approximate the result as $\overline{\mathsf{any}}$, since we do not know what the contract ensures. Otherwise, it may evaluate to a non-contract value, in which case it blames the opaque module. However, since errors which blame opaque modules may be soundly ignored by our semantics, we do not need to consider these cases.

$$\begin{array}{rll}(C \Leftarrow^{f,g}_{V,f'} \bullet) & \widehat{\mathbf{c}}^1 & \overline{C} \\ (V \Leftarrow^{f,g}_{V'',f'} V') & \widehat{\mathbf{c}}^1 & (\mathsf{wrong}^f\ V) \text{ if } V \text{ is not a contract} \\ (\overline{(\mathsf{pred}\ L)} \Leftarrow^{f,g}_{V,f'} \bullet) & \widehat{\mathbf{c}}^1 & \overline{\mathsf{any}} \\ (\overline{(\mathsf{pred}\ L)} \Leftarrow^{f,g}_{V',f'} V) & \widehat{\mathbf{c}}^1 & V \\ (\overline{(\mathsf{pred}\ L)} \Leftarrow^{f,g}_{V',f'} V) & \widehat{\mathbf{c}}^1 & (\mathsf{blame}^{f/g}_{g/f}\ V^?\ C^?\ V^?)\end{array}$$

The notation f/g indicates that in the last rule, either f or g may be blamed, with the other as the opposite party.

An evaluation function for the language is defined as follows:

$$\mathit{eval}_{\widehat{\mathbf{c}}^1}(\boldsymbol{M}E) = \{E' \mid E \longmapsto\!\!\!\!\!\rightarrow_{\mathbf{vc\widehat{c}\widehat{c}}^1 \Delta_{\boldsymbol{M}}} E'\}.$$

From this, we can derive CEK and abstract CESK* machines from our abstract reduction semantics (we again defer these straightforward definitions to Appendix A for reasons of space) to produce a modular static analyzer.

Soundness: Having extended the language of expressions to include first-class contracts, we now revisit the soundness theorem. This theorem relies on an extension of the $\sqsubseteq$ relation from Section 4.2 with additional structural rules (see Appendix C.2) to handle relations between contracts in expression as well as relations between abstract values. The key additional rule is that $C \sqsubseteq C'$ implies $\overline{C} \sqsubseteq \overline{C'}$.
This gives the following extended soundness theorem:

Theorem 5. *$\widehat{\mathit{aval}}^{\mathrm{cesk}^\star}_{\widehat{\mathbf{c}}^1}$ is a sound, computable approximation of $\mathit{eval}_{\mathbf{c}^1}$ for all possible instantiations of opaque modules.*

5 Related Work

Modular analysis of programs is a broad topic; in this section, we discuss related work on higher-order programming languages.

Modular Analyzers: Shivers [23], Serrano [22], 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 the abstract value `any` in our setting. Such an abstraction can be expressed in our setting by always using the `any` contract on unknown modules, and therefore allowing more descriptive contracts can be seen as a refinement of the abstraction on missing program components—the more we can say about the contract of an component, the more precisely we can determine the behavior of the program, possible failures, and whose to blame.

The most closely related work to our specification-as-values notion of reduction is Reppy's work that presents an analysis that uses types as abstract values in a similar fashion to our presentation [21]. Reppy develops a 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.

Cousot has demonstrated types are abstract interpretations in a denotational setting [6], which bears a conceptual relation to the modular analysis using types in Section 3, however our treatment is more syntactic: types are abstract interpretations of syntactic values of that type, which are used only for opaque modules; and our computable approximation method is based on a finite approximation of a transition system, common in the literature on abstract interpretation [7].

Set-Based Analysis from Contracts: The second-class contract system of Section 4 is closely related to and inspired by the modular set-based analysis from contracts work of Meunier et al. [19,18]. Meunier takes a set-based approach in which programs are annotated with labels and *flow inequalities* are generated that relate flow sets through the program text. When solved, the analysis maps source labels to sets of abstract values to which that expression may evaluate.

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 does not have analogues of the $*$-marked rules in Figure 7. To see this, consider the example program from Figure 8. This program reduces, in either system, to an error blaming module `h`. However, Meunier et al.'s analysis does not predict an error blaming `h`. Both systems approximate the result of (`f g`) with (the analogue of) the abstract value $\overline{(\mathsf{any} \to \mathsf{any})}$; however, since their system is lacking the $*$-marked rules, the

application of this abstract value to `8` is predicted not to produce blame. This unsoundness has been verified in the prototype described by Meunier et al.

Meunier's set-based analysis is modular in the sense that the "analysis produces the same predictions for a given point in the program regardless of whether it analyzes the whole program or just the surrounding module." Modularity is achieved by a whole-program transformation which lifts each module out of its context as well as lifting each contract check.

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*.

The addition of the $*$-marked rules remedy the unsoundness of Meunier et al.'s system, but our work also extends theirs in several ways. First, we derive a semantics-based modular analysis, avoiding needless imprecision by simply following the semantics for conditionals. Second, our system allows for any subset of modules to be analyzed together, rather than analyzing each independently. Third, our system scales to first-class contracts. Finally, we present our analysis of contracts within a general framework for modular analysis, which applies to specifications other than contracts. Their work discusses the use of theorem prover to determine whether a value has already passed a stronger contract check; we leave this to future work.

Compositional Analyzers: Another sense of the words *modular* and *compositional* is often used in the literature [20]; many authors use these (interchangeably) to mean that program components can be analyzed in isolation and whole programs can be analyzed by combining these component-wise analysis results. Banerjee and Jensen [3,2] present a type based analysis, which is modular in the sense that there is a way to summarize all of the constraints generated from a particular expression, allowing incrementalization of the analysis. It is not modular in the sense of our analysis. Flanagan [12] takes a similar approach in an untyped setting (using the terminology "componential"). Lee et al. does something similar with component-wise approach to 0CFA [17]. But in our sense of modularity, these analyses are not modular—before analysis results can be reported, the whole program must be available. Consequently, these approaches are not adequate for reasoning about opaque components.

Analyzing Contracts: Xu et al. [25] present a system for statically verifying whether a program may fail a contract check. Their strategy is to statically insert blame expressions, in effect expanding contract checks in place, optimizing the program, and reporting any remaining contract checks as potential failures. While not a program analysis, and thus in a significantly different setting than the present work, it does have some similarities, in particular using the contract of one function in the analysis of another. 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. Finally, their system greatly restricts the language of contracts to support their static checking strategy.

Many other researchers have studied verifying first-order properties of programs expressed as contracts [4]. We could use such analyses to 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 [15]. 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 [16], 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.

6 Conclusion

Deriving static analyzers from the existing semantics of a programming language has numerous benefits, among them simplicity and correctness by construction. However, deriving a modular analysis in this fashion requires a notion of of modular reduction. We therefore introduce *abstract* reduction semantics, in which some modules in the program are treated opaquely, with their specifications treated as values which approximate the true behavior of the opaque module.

Our technique is applicable to multiple forms of specifications, from simple types to complex first-class contracts. Further, our techniques provide simple proofs of both soundness and modularity. In contrast, previous approaches required complex proofs of soundness, in which errors remained nonetheless.

We hope to extend our work to both richer type systems and richer contract systems. While our contract language is richer than that considered in previous analysis work, it does not yet encompass the full scope of existing contract libraries such as that provided by Racket. We plant to investigate extensions to dependent contracts as well as additional contract combinators such as conjunction and disjunction, which may support improved analysis precision.

Acknowledgments: We thank Phillipe Meunier for discussions of his prior work and providing code for the prototype implementation of his system.

References

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

2. Banerjee, A.: A modular, polyvariant and type-based closure analysis. In: Berman, A.M. (ed.) ICFP '97. pp. 1–10. ACM Press (1997)
3. Banerjee, A., Jensen, T.: Modular control-flow analysis with rank 2 intersection types. Mathematical. Structures in Comp. Sci. 13(1), 87–124 (2003)
4. Barnett, M., Fähndrich, M., Leino, K.R.M., Müller, P., Schulte, W., Venter, H.: Specification and verification: The Spec# experience. Comm. of the ACM (2010)
5. Biernacka, M., Danvy, O.: A concrete framework for environment machines. ACM Trans. Comput. Logic 9(1), 1–30 (2007)
6. Cousot, P.: Types as abstract interpretations. In: POPL '97. pp. 316–331. ACM (1997)
7. Cousot, P., Cousot, R.: Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL '77. pp. 238–252. ACM (1977)
8. Danvy, O.: An Analytical Approach to Program as Data Objects. DSc thesis, Department of Computer Science, Aarhus University (2006)
9. Felleisen, M.: The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages. Ph.D. thesis, Indiana University (1987)
10. Felleisen, M., Findler, R.B., Flatt, M.: Semantics Engineering with PLT Redex. MIT Press (2009)
11. Findler, R.B., Felleisen, M.: Contracts for higher-order functions. In: ICFP '02. pp. 48–59. ACM (2002)
12. Flanagan, C.: Effective Static Debugging via Componential Set-Based Analysis. Ph.D. thesis, Rice University (1997)
13. Flatt, M., PLT: Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc. (2010)
14. Greenberg, M., Pierce, B.C., Weirich, S.: Contracts made manifest. In: POPL '10. pp. 353–364. ACM (2010)
15. Johan, R., Akademi, A., Wright, J.V.: Refinement Calculus: A Systematic Introduction. Springer-Verlag New York, Inc. (1998)
16. Kuan, G., MacQueen, D., Findler, R.B.: A rewriting semantics for type inference. In: Nicola, R.D. (ed.) ESOP '07. vol. 4421 (2007)
17. Lee, O., Yi, K., Paek, Y.: A proof method for the correctness of modularized 0CFA. Information Processing Letters 81, 179–185 (2002)
18. Meunier, P.: Modular Set-Based Analysis from Contracts. Ph.D. thesis, Northeastern University (2006)
19. Meunier, P., Findler, R.B., Felleisen, M.: Modular set-based analysis from contracts. In: POPL '06. pp. 218–231. ACM (2006)
20. Midtgaard, J.: Control-flow analysis of functional programs. Tech. Rep. BRICS RS-07-18, University of Aarhus (2007)
21. Reppy, J.: Type-sensitive control-flow analysis. In: ML '06. pp. 74–83. ACM (2006)
22. Serrano, M.: Control flow analysis: a functional languages compilation paradigm. In: SAC '95. pp. 118–122. ACM (1995)
23. Shivers, O.: Control-flow analysis of higher-order languages. Ph.D. thesis, Carnegie Mellon University (1991)
24. Van Horn, D., Might, M.: Abstracting abstract machines. In: ICFP '10. pp. 51–62. ACM (2010)
25. Xu, D.N., Peyton Jones, S., Claessen, S.: Static contract checking for Haskell. In: POPL '09. pp. 41–52. ACM (2009)

A Contract Machines

In this section we give a complete definition of the CEK machine for the language with modules and first-class contracts.

A.1 CEK Machine for First-class Contracts

$\varsigma \longmapsto^{\mathrm{cek}}_{\mathbf{vc\hat{c}\hat{c}}^1 \Delta_M} \varsigma'$	
LAMBDA	
$\langle V, \rho, \mathsf{fn}^f((\lambda x.E), \rho', \kappa)\rangle$	$\langle E, \rho'[x \mapsto \langle V, \rho\rangle], \kappa\rangle$
$\langle V, \rho, \mathsf{fn}^f(n, \rho', \kappa)\rangle$	$\langle(\mathsf{wrong}^f\ n), \rho, \kappa\rangle$
$\langle 0, \rho, \mathsf{if}(E_0, E_1, \rho', \kappa)\rangle$	$\langle E_0, \rho', \kappa\rangle$
$\langle V, \rho, \mathsf{if}(E_0, E_1, \rho', \kappa)\rangle$	$\langle E_1, \rho', \kappa\rangle$, if $V \neq 0$
$\langle x, \rho, \kappa\rangle$	$\langle V, \rho', \kappa\rangle$ if $\rho(x) = \langle V, \rho'\rangle$
$\langle f, \rho, \kappa\rangle$	$\langle V, \emptyset, \kappa\rangle$, if $\Delta_M(f) = V$
$\langle(E_0\ E_1)^f, \rho, \kappa\rangle$	$\langle E_0, \rho, \mathsf{ar}^f(E_1, \rho, \kappa)\rangle$
$\langle V, \rho, \mathsf{ar}^f(E, \rho', \kappa)\rangle$	$\langle E, \rho', \mathsf{fn}^f(V, \rho, \kappa)\rangle$
$\langle(\mathsf{if0}\ E_1\ E_2\ E_3), \rho, \kappa\rangle$	$\langle E_1, \rho, \mathsf{if}(E_2, E_3, \rho, \kappa)\rangle$
CONTRACTS	
$\langle(C \Leftarrow^{f,g}_{V,f'} E), \rho, \kappa\rangle$	$\langle E, \rho, \mathsf{ck}^{f,g}_{V,f'}(C, \kappa)\rangle$
$\langle n, \rho, \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, \kappa)\rangle$	$\langle n, \rho, \kappa\rangle$
$\langle W, \rho, \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, \kappa)\rangle$	$\langle(\mathsf{blame}^f_{f'}\ V\ \mathsf{int}\ W), \rho, \kappa\rangle$
$\langle W, \rho, \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), \kappa)\rangle$	$\langle((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V,f'} W), \rho, \kappa\rangle$
$\langle n, \rho, \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), \kappa)\rangle$	$\langle(\mathsf{blame}^f_{f'}\ V\ (C_1 \to C_2)\ n), \rho, \kappa\rangle$
$\langle V, \rho, \mathsf{ck}^{f,g}_{V',h}((\mathsf{pred}\ L), \kappa)\rangle$	$\langle(L\ V), \rho, \mathsf{if}(V, (\mathsf{blame}^f_h\ V'\ (\mathsf{pred}\ L)\ V), \rho, \kappa)\rangle$
$\langle V, \rho, \mathsf{fn}(((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V',h} W), \rho', \kappa)\rangle$	$\langle(C_1 \Leftarrow^{g,f}_{V,h} V), \rho, \mathsf{fn}(W, \rho', \mathsf{ck}^{f,g}_{V',h}(C_2, \kappa))\rangle$

(Rules for abstract values and first-class contracts are on the next page.)

We add the following continuation forms to handle the evaluation contexts introduced by first-class contracts:

$$\mathcal{E} \mathrel{+}= (\mathcal{E} \Leftarrow^{f,f}_{V,f} E) \mid (\mathcal{E} \to E) \mid (V \to \mathcal{E})$$
$$\kappa \mathrel{+}= \mathsf{kc}^{f,f}_{V,f}(E, \kappa) \mid \mathsf{arl}(E, \kappa) \mid \mathsf{arr}(V, \kappa)$$

$\varsigma \longmapsto^{\mathrm{cek}}_{\mathbf{vc\bar{c}\hat{c}}^1 \Delta_M} \varsigma'$	
Contracts as Abstract Values	
$\langle V, \rho, \mathsf{fn}^f(\overline{(\mathsf{pred}\ L)}, \rho', \kappa)\rangle$	$\langle \overline{\mathsf{any}}, \emptyset, \kappa\rangle$
$\langle V, \rho, \mathsf{fn}^f(\overline{(\mathsf{pred}\ L)}, \rho', \kappa)\rangle$	$\langle (\mathsf{blame}^{f_?}_{g_?}\ V_?\ C_?\ V'_?), \emptyset, \kappa\rangle$
$\langle V, \rho, \mathsf{fn}^f(\overline{(C_1 \to C_2)}, \rho', \kappa)\rangle$	$\langle \overline{C_2}, \emptyset, \kappa\rangle$
$\langle V, \rho, \mathsf{fn}^f(\overline{(C_1 \to C_2)}, \rho', \kappa)\rangle$	$\langle (\mathsf{blame}^{f_?}_{g_?}\ V_?\ C_?\ V'_?), \emptyset, \kappa\rangle$
$\langle V, \rho, \mathsf{fn}^f(\overline{\mathsf{int}}, \rho', \kappa)\rangle$	$\langle (\mathsf{wrong}^f\ \overline{\mathsf{int}}), \emptyset, \kappa\rangle$
$\langle \overline{C}, \rho, \mathsf{if}(E_1, E_2, \rho', \kappa)\rangle$	$\langle E_2, \rho', \kappa\rangle$
$\langle \overline{\mathsf{int}}, \rho, \mathsf{if}(E_1, E_2, \rho', \kappa)\rangle$	$\langle E_1, \rho', \kappa\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \mathsf{if}(E_1, E_2, \rho', \kappa)\rangle$	$\langle E_1, \rho', \kappa\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, \kappa)\rangle$	$\langle \overline{\mathsf{int}}, \rho, \kappa\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, \kappa)\rangle$	$\langle (\mathsf{blame}^f_{f'}\ V\ \mathsf{int}\ \overline{\mathsf{any}}), \emptyset, \kappa\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), \kappa)\rangle$	$\langle \overline{(C_1 \to C_2)}, \emptyset, \kappa\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), \kappa)\rangle$	$\langle (\mathsf{blame}^f_{f'}\ V\ (C_1 \to C_2)\ \overline{(\mathsf{pred}\ L)}), \emptyset, \kappa\rangle$
$\langle \overline{\mathsf{int}}, \rho, \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), \kappa)\rangle$	$\langle (\mathsf{blame}^f_{f'}\ V\ (C_1 \to C_2)\ \overline{\mathsf{int}}), \emptyset, \kappa\rangle$
First-class Contracts	
$\langle (E_1 \to E_2), \rho, \kappa\rangle$	$\langle E_1, \rho, \mathsf{arl}(E_2, \kappa)\rangle$
$\langle V, \rho, \mathsf{arl}(E, \kappa)\rangle$	$\langle E, \rho, \mathsf{arr}(V, \kappa)\rangle$
$\langle V, \rho, \mathsf{arr}(V', \kappa)\rangle$	$\langle (V' \to V), \rho, \kappa\rangle$
$\langle (E_1 \Leftarrow^{f,g}_{V,f'} E_2), \rho, \kappa\rangle$	$\langle E_1, \rho, \mathsf{kc}^{f,g}_{V,f'}(E_2, \kappa)\rangle$
$\langle V, \rho, \mathsf{kc}^{f,g}_{V',f'}(E, \kappa)\rangle$	$\langle E, \rho, \mathsf{ck}^{f,g}_{V',f'}(V, \kappa)\rangle$
$\langle \bullet, \rho, \mathsf{ck}^{f,g}_{V,f'}(C, \kappa)\rangle$	$\langle \overline{C}, \emptyset, \kappa\rangle$
$\langle V', \rho, \mathsf{ck}^{f,g}_{V'',f'}(V, \kappa)\rangle$	$\langle (\mathsf{wrong}^f\ V), \emptyset, \kappa\rangle$, if V is not a contract
$\langle \bullet, \rho, \mathsf{ck}^{f,g}_{V,f'}(\overline{(\mathsf{pred}\ L)}, \kappa)\rangle$	$\langle \overline{\mathsf{any}}, \emptyset, \kappa\rangle$
$\langle V, \rho, \mathsf{ck}^{f,g}_{V',f'}(\overline{(\mathsf{pred}\ L)}, \kappa)\rangle$	$\langle V, \rho, \kappa\rangle$
$\langle V, \rho, \mathsf{ck}^{f,g}_{V',f'}(\overline{(\mathsf{pred}\ L)}, \kappa)\rangle$	$\langle (\mathsf{blame}^{f/g}_{g/f}\ V^?\ C^?\ V^?), \emptyset, \kappa\rangle$

A.2 CESK* Machine for First-class Contracts

$\varsigma \longmapsto^{\mathrm{cesk}^\star}_{\mathbf{vc\widehat{c}c}^1 \Delta_M} \varsigma'$ where $\kappa \in \hat{\sigma}(a), b = \mathsf{alloc}(\hat{\varsigma}, \kappa)$	
LAMBDA	
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{fn}^f((\lambda x.E), \rho', a')$	$\langle E, \rho'[x \mapsto b], \hat{\sigma} \sqcup [b \mapsto \langle V, \rho\rangle], a'\rangle$
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{fn}^f(n, \rho', a')$	$\langle(\mathsf{wrong}^f\ n), \rho, \hat{\sigma}, a'\rangle$
$\langle 0, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{if}(E_0, E_1, \rho', a')$	$\langle E_0, \rho, \hat{\sigma}, a'\rangle$
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{if}(E_0, E_1, \rho', a')$	$\langle E_1, \rho, \hat{\sigma}, a'\rangle$ if $V \neq 0$
$\langle x, \rho, \hat{\sigma}, \kappa\rangle$	$\langle V, \rho', \hat{\sigma}, \kappa\rangle$ if $\hat{\sigma}(\rho(x)) \ni \langle V, \rho'\rangle)$
$\langle f, \rho, \hat{\sigma}, \kappa\rangle$	$\langle V, \emptyset, \hat{\sigma}, \kappa\rangle$ if $\Delta_M(f) = V$
$\langle(E_0\ E_1), \rho, \hat{\sigma}, a\rangle$	$\langle E_0, \rho, \hat{\sigma} \sqcup [b \mapsto \mathsf{ar}^f(E_1, \rho, a)], b\rangle$
$\langle V, \rho, \hat{\sigma}, a\rangle$ if $\kappa = \mathsf{ar}^f(E, \rho', a')$	$\langle E, \rho', \hat{\sigma} \sqcup [b \mapsto \mathsf{fn}^f(V, \rho, a')], b\rangle$
$\langle(\mathsf{if0}\ E_1\ E_2\ E_3), \rho, \hat{\sigma}, a\rangle$	$\langle E_1, \rho, \hat{\sigma} \sqcup [b \mapsto \mathsf{if}(E_2, E_3, \rho, a)], b\rangle$
CONTRACTS	
$\langle(C \Leftarrow^{f,g}_{V,f'} E), \rho, a, \sigma\rangle$	$\langle E, \rho, \sigma[b \mapsto \mathsf{ck}^{f,g}_{V,f'}(C, a)], b\rangle$
$\langle n, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, a')$	$\langle n, \rho, \sigma, a'\rangle$
$\langle W, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, a')$	$\langle(\mathsf{blame}^f_{f'}\ V\ \mathsf{int}\ W), \rho, \sigma, a'\rangle$
$\langle W, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), a')$	$\langle((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V,f'} W), \rho, \sigma, a'\rangle$
$\langle n, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), a')$	$\langle(\mathsf{blame}^f_{f'}\ V\ (C_1 \to C_2)\ n), \rho, \sigma, a'\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V',h}((\mathsf{pred}\ L), a')$	$\langle(L\ V), \rho, \sigma[b \mapsto \mathsf{if}(V, (\mathsf{blame}^f_h\ V'\ (\mathsf{pred}\ L)\ V), \rho, a')], b\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{fn}(((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V',h} W), \rho', a')$	$\langle(C_1 \Leftarrow^{g,f}_{V,h} V), \rho, \sigma[b \mapsto \mathsf{fn}(W, \rho', \mathsf{ck}^{f,g}_{V',h}(C_2, a'))], b\rangle$

$\varsigma \longmapsto^{\mathrm{cesk}^\star_{\mathbf{vc\widehat{c}c}^1\Delta_M}} \varsigma'$	
Contracts as Abstract Values	
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{fn}^f(\overline{(\mathsf{pred}\ L)}, \rho', a')$	$\langle \overline{\mathsf{any}}, \emptyset, \sigma, \kappa\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{fn}^f(\overline{(\mathsf{pred}\ L)}, \rho', a')$	$\langle(\mathsf{blame}^{f_?}_{g_?}\ V_?\ C_?\ V'_?), \emptyset, \sigma, \kappa\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{fn}^f(\overline{(C_1 \to C_2)}, \rho', a')$	$\langle \overline{C_2}, \emptyset, \sigma, \kappa\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{fn}^f(\overline{(C_1 \to C_2)}, \rho', a')$	$\langle(\mathsf{blame}^{f_?}_{g_?}\ V_?\ C_?\ V'_?), \emptyset, \sigma, \kappa\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{fn}^f(\overline{\mathsf{int}}, \rho', a')$	$\langle(\mathsf{wrong}^f\ \overline{\mathsf{int}}), \emptyset, \sigma, \kappa\rangle$
$\langle \overline{C}, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{if}(E_1, E_2, \rho', a')$	$\langle E_2, \rho', \sigma, a'\rangle$
$\langle \overline{\mathsf{int}}, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{if}(E_1, E_2, \rho', a')$	$\langle E_1, \rho', \sigma, a'\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{if}(E_1, E_2, \rho', a')$	$\langle E_1, \rho', \sigma, a'\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, \kappa)$	$\langle \overline{\mathsf{int}}, \rho, \sigma, a'\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}(\mathsf{int}, \kappa)$	$\langle(\mathsf{blame}^f_{f'}\ V\ \mathsf{int}\ \overline{\mathsf{any}}), \emptyset, \sigma, a'\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), a')$	$\langle \overline{(C_1 \to C_2)}, \emptyset, \sigma, a'\rangle$
$\langle \overline{(\mathsf{pred}\ L)}, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), a')$	$\langle(\mathsf{blame}^f_{f'}\ V\ (C_1 \to C_2)\ \overline{(\mathsf{pred}\ L)}), \emptyset, \sigma, a'\rangle$
$\langle \overline{\mathsf{int}}, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}((C_1 \to C_2), a')$	$\langle(\mathsf{blame}^f_{f'}\ V\ (C_1 \to C_2)\ \overline{\mathsf{int}}), \emptyset, \sigma, a'\rangle$
First-class Contracts	
$\langle(E_1 \to E_2), \rho, \sigma, a\rangle$	$\langle E_1, \rho, \sigma \sqcup [b \mapsto \mathsf{arl}(E_2, a)], b\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{arl}(E, a')$	$\langle E, \rho, \sigma \sqcup [b \mapsto \mathsf{arr}(V, a')], b\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{arr}(V', a')$	$\langle(V' \to V), \rho, \sigma, a'\rangle$
$\langle(E_1 \Leftarrow^{f,g}_{V,f'} E_2), \rho, \sigma, a'\rangle$	$\langle E_1, \rho, \sigma \sqcup [b \mapsto \mathsf{kc}^{f,g}_{V,f'}(E_2, a')], b\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{kc}^{f,g}_{V',f'}(E, a')$	$\langle E, \rho, \sigma \sqcup [b \mapsto \mathsf{ck}^{f,g}_{V',f'}(V, a')], b\rangle$
$\langle \bullet, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}(C, a')$	$\langle \overline{C}, \emptyset, \sigma, a'\rangle$
$\langle V', \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V'',f'}(V, a')$	$\langle(\mathsf{wrong}^f\ V), \emptyset, \sigma, a'\rangle$, if V is not a contract
$\langle \bullet, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V,f'}(\overline{(\mathsf{pred}\ L)}, a')$	$\langle \overline{\mathsf{any}}, \emptyset, \sigma, a'\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V',f'}(\overline{(\mathsf{pred}\ L)}, a')$	$\langle V, \rho, \sigma, a'\rangle$
$\langle V, \rho, \sigma, a\rangle$, if $\kappa = \mathsf{ck}^{f,g}_{V',f'}(\overline{(\mathsf{pred}\ L)}, a')$	$\langle(\mathsf{blame}^{f/g}_{g/f}\ V^?\ C^?\ V^?), \emptyset, \sigma, a'\rangle$

B Type and approximation judgments

B.1 Typing

The typing judgment for programs is defined as follows:

$$\frac{\forall(\mathsf{module}\ f\ T\ V) \in \boldsymbol{M}.\Gamma_{\boldsymbol{M}} \vdash V : T \quad \Gamma_{\boldsymbol{M}} \vdash E : T'}{\boldsymbol{M}E : T'},$$

where

$$\Gamma_{\boldsymbol{M}} = \{(f,T) \mid (\mathsf{module}\ f\ T\ E) \in \boldsymbol{M}\} \cup \{(f,T) \mid (\mathsf{module}\ f\ T\ \bullet) \in \boldsymbol{M}\}.$$

The typing judgment for expressions is just simple typing defined as usual with the exception of the last case, which assigns a type to a type value:

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \qquad \frac{\Gamma(f) = T}{\Gamma \vdash f : T} \qquad \Gamma \vdash n : \mathsf{int}$$

$$\frac{\Gamma \vdash E_1 : T \quad \Gamma \vdash E_2 : T' \quad \Gamma \vdash E_3 : T'}{\Gamma \vdash (\mathsf{if0}\ E_1\ E_2\ E_3) : T'} \qquad \frac{\Gamma; x : T \vdash E : T'}{\Gamma \vdash (\lambda x.E) : T \to T'}$$

$$\frac{\Gamma \vdash E : T' \to T \quad \Gamma \vdash E' : T'}{\Gamma \vdash (E\ E') : T} \qquad \Gamma \vdash \overline{T} : T$$

B.2 Approximation of Typed Programs

The approximation judgment is defined by transitive closure of the rules below.

Abstract Values

$$\frac{\Gamma \vdash E : T}{\Gamma \vdash E \sqsubseteq_T \overline{\overline{T}}}$$

Structural Rules

$$\frac{M_0 \sqsubseteq N_0 \quad \ldots \quad M_n \sqsubseteq N_n \quad \Gamma_{M_0 \ldots M_n} \vdash E \sqsubseteq_T E'}{M_0 \ldots M_n E \sqsubseteq_T N_0 \ldots N_n E'}$$

$$(\mathsf{module}\ f\ T\ V) \sqsubseteq (\mathsf{module}\ f\ T\ \bullet) \qquad M \sqsubseteq M \qquad \frac{\Gamma \vdash E : T}{E \sqsubseteq_T E}$$

$$\frac{\Gamma \vdash E_0 \sqsubseteq_{T' \to T} E_0' \quad \Gamma \vdash E_1 \sqsubseteq_{T'} E_1'}{\Gamma \vdash (E_0\ E_1) \sqsubseteq_T (E_0'\ E_1')} \qquad \frac{\Gamma; x : T \vdash E \sqsubseteq_{T'} E'}{\Gamma \vdash (\lambda x.E) \sqsubseteq_{T \to T'} (\lambda x.E')}$$

$$\frac{\Gamma \vdash E_0 \sqsubseteq_T E_0' \quad \Gamma \vdash E_1 \sqsubseteq_{T'} E_1' \quad \Gamma \vdash E_2 \sqsubseteq_{T'} E_2'}{\Gamma \vdash (\mathsf{if0}\ E_0\ E_1\ E_2) \sqsubseteq_{T'} (\mathsf{if0}\ E_0'\ E_1'\ E_2')}$$

C Contract approximation judgments

Again, we take the transitive closure of the rules.

C.1 Second-class contracts

Abstract Values

$$n \sqsubseteq \overline{\mathsf{int}} \qquad V \sqsubseteq \overline{\mathsf{any}} \qquad V \sqsubseteq \overline{(\mathsf{pred}\ L)} \qquad (C \Leftarrow^{f,g}_{V,f'} E) \sqsubseteq \overline{C}$$

$$((C_1 \dashrightarrow C_2) \Leftarrow^{f,g}_{V,f'} E) \sqsubseteq \overline{(C_1 \rightarrow C_2)}$$

Placeholders

$$g \sqsubseteq f^? \qquad V \sqsubseteq V^? \qquad C \sqsubseteq C^?$$

Modules

$$\frac{(\mathsf{module}\ f\ C\ \bullet) \in \boldsymbol{M}}{(\mathsf{blame}^f_g\ V\ C'\ V') \sqsubseteq E} \qquad (\mathsf{module}\ f\ T\ V) \sqsubseteq (\mathsf{module}\ f\ T\ \bullet)$$

Structural Rules

$$E \sqsubseteq E \qquad \frac{E \sqsubseteq E'}{(\lambda x.E) \sqsubseteq (\lambda x.E')} \qquad \frac{E_0 \sqsubseteq E_0' \quad E_1 \sqsubseteq E_1'}{(E_0\ E_1) \sqsubseteq (E_0'\ E_1')}$$

$$\frac{E_0 \sqsubseteq E_0' \quad E_1 \sqsubseteq E_1' \quad E_2 \sqsubseteq E_2'}{(\mathsf{if0}\ E_0\ E_1\ E_2) \sqsubseteq (\mathsf{if0}\ E_0'\ E_1'\ E_2')} \qquad \frac{E \sqsubseteq E' \quad V \sqsubseteq V'}{(C \Leftarrow^{f,g}_{V,f'} E) \sqsubseteq (C \Leftarrow^{f,g}_{V',f'} E')}$$

$$\frac{f \sqsubseteq f' \quad g \sqsubseteq g' \quad C \sqsubseteq C' \quad V_1 \sqsubseteq {V_1}' \quad V_2 \sqsubseteq {V_2}'}{(\mathsf{blame}^f_g\ V_1\ C\ V_2) \sqsubseteq (\mathsf{blame}^{f'}_{g'}\ V_1'\ C'\ V_2')}$$

$$\frac{M_0 \sqsubseteq N_0 \quad \ldots \quad M_n \sqsubseteq N_n \quad \Gamma_{M_0 \ldots M_n} \vdash E \sqsubseteq E'}{M_0 \ldots M_n E \sqsubseteq N_0 \ldots N_n E'} \qquad M \sqsubseteq M$$

C.2 First-class contracts

The below rules are used *in addition* to the rules above for second-class contracts.

$$\frac{C \sqsubseteq C'}{\overline{C} \sqsubseteq \overline{C'}} \qquad \frac{L \sqsubseteq L'}{(\mathsf{pred}\ L) \sqsubseteq (\mathsf{pred}\ L')} \qquad \frac{{E_1} \sqsubseteq {E_1}' \quad {E_2} \sqsubseteq {E_2}'}{(E_1 \rightarrow E_2) \sqsubseteq (E_1' \rightarrow E_2')}$$

$$\frac{E \sqsubseteq E' \quad V \sqsubseteq V' \quad C \sqsubseteq C'}{(C \Leftarrow^{f,g}_{V,f'} E) \sqsubseteq (C' \Leftarrow^{f,g}_{V',f'} E')}$$

D Unload Functions

$$\mathcal{U}(\langle E, \rho, \kappa\rangle) = \mathcal{E}[\mathsf{close}(E, \rho)]$$

where $\mathcal{E} = \mathsf{unwind}(\kappa)$.

$$\mathsf{close}(E, \{\boldsymbol{x} \mapsto \langle \boldsymbol{V}, \boldsymbol{\rho}\rangle\}) = [\mathsf{close}(\boldsymbol{V}, \boldsymbol{\rho})/\boldsymbol{x}]E$$

$$\begin{array}{lll}
\mathsf{unwind}(\mathsf{mt}) & = [\,] & \\
\mathsf{unwind}(\mathsf{ar}^f(E, \rho, \kappa)) & = \mathcal{E}[([]\ \mathsf{close}(E, \rho))^f] & \text{if } \mathcal{E} = \mathsf{unwind}(\kappa) \\
\mathsf{unwind}(\mathsf{fn}^f(V, \rho, \kappa)) & = \mathcal{E}[(\mathsf{close}(V, \rho)\ [])^f] & \text{if } \mathcal{E} = \mathsf{unwind}(\kappa) \\
\mathsf{unwind}(\mathsf{if}(E_1, E_2, \rho, \kappa)) & = \mathcal{E}[(\mathsf{if0}\ []\ \mathsf{close}(E_1, \rho)\ \mathsf{close}(E_2, \rho))] & \text{if } \mathcal{E} = \mathsf{unwind}(\kappa) \\
\mathsf{unwind}(\mathsf{ck}^{f,g}_{V',h}(V, \kappa)) & = \mathcal{E}[(V \Leftarrow^{f,g}_{V',h} [])] & \text{if } \mathcal{E} = \mathsf{unwind}(\kappa) \\
\mathsf{unwind}(\mathsf{kc}^{f,g}_{V',h}(E, \kappa)) & = \mathcal{E}[([] \Leftarrow^{f,g}_{V',h} E)] & \text{if } \mathcal{E} = \mathsf{unwind}(\kappa) \\
\mathsf{unwind}(\mathsf{arl}(E, \kappa)) & = \mathcal{E}[([] \to E)] & \text{if } \mathcal{E} = \mathsf{unwind}(\kappa) \\
\mathsf{unwind}(\mathsf{arr}(V, \kappa)) & = \mathcal{E}[(V \to [])] & \text{if } \mathcal{E} = \mathsf{unwind}(\kappa)
\end{array}$$

$$\widehat{\mathcal{U}}(\langle E, \rho, \sigma, a\rangle) = \{\mathcal{E}[E'] \mid \kappa \in \sigma(a),\ E' \in \widehat{\mathsf{close}}_\sigma(E, \rho),\ \mathcal{E} \in \widehat{\mathsf{unwind}}_\sigma(\kappa)\}$$

$$\widehat{\mathsf{close}}_\sigma(E, \{\boldsymbol{x} \mapsto \boldsymbol{a}\}) = \{[\widehat{\mathsf{close}}_\sigma(\boldsymbol{V}, \boldsymbol{\rho})/\boldsymbol{x}]E \mid \langle \boldsymbol{V}, \boldsymbol{\rho}\rangle \in \sigma(\boldsymbol{a})\}$$

$$\begin{array}{lll}
\widehat{\mathsf{unwind}}_\sigma(\mathsf{mt}) & \ni [\,] & \\
\widehat{\mathsf{unwind}}_\sigma(\mathsf{ar}^f(E, \rho, \kappa)) & \ni \mathcal{E}[([]\ \widehat{\mathsf{close}}_\sigma(E, \rho))^f] & \text{if } \mathcal{E} \in \widehat{\mathsf{unwind}}_\sigma(\kappa) \\
\widehat{\mathsf{unwind}}_\sigma(\mathsf{fn}^f(V, \rho, \kappa)) & \ni \mathcal{E}[(\widehat{\mathsf{close}}_\sigma(V, \rho)\ [])^f] & \text{if } \mathcal{E} \in \widehat{\mathsf{unwind}}_\sigma(\kappa) \\
\widehat{\mathsf{unwind}}_\sigma(\mathsf{if}(E_1, E_2, \rho, \kappa)) & \ni \mathcal{E}[(\mathsf{if0}\ []\ \widehat{\mathsf{close}}_\sigma(E_1, \rho)\ \widehat{\mathsf{close}}_\sigma(E_2, \rho))] & \text{if } \mathcal{E} \in \widehat{\mathsf{unwind}}_\sigma(\kappa) \\
\widehat{\mathsf{unwind}}_\sigma(\mathsf{ck}^{f,g}_{V',h}(V, \kappa)) & \ni \mathcal{E}[(V \Leftarrow^{f,g}_{V',h} [])] & \text{if } \mathcal{E} \in \widehat{\mathsf{unwind}}_\sigma(\kappa) \\
\widehat{\mathsf{unwind}}_\sigma(\mathsf{kc}^{f,g}_{V',h}(E, \kappa)) & \ni \mathcal{E}[([] \Leftarrow^{f,g}_{V',h} E)] & \text{if } \mathcal{E} \in \widehat{\mathsf{unwind}}_\sigma(\kappa) \\
\widehat{\mathsf{unwind}}_\sigma(\mathsf{arl}(E, \kappa)) & \ni \mathcal{E}[([] \to E)] & \text{if } \mathcal{E} \in \widehat{\mathsf{unwind}}_\sigma(\kappa) \\
\widehat{\mathsf{unwind}}_\sigma(\mathsf{arr}(V, \kappa)) & \ni \mathcal{E}[(V \to [])] & \text{if } \mathcal{E} \in \widehat{\mathsf{unwind}}_\sigma(\kappa)
\end{array}$$