Soft Contract Verification

Phúc C. Nguyễn
University of Maryland
pcn@cs.umd.edu

Sam Tobin-Hochstadt
Indiana University
samth@cs.indiana.edu

David Van Horn
University of Maryland
dvanhorn@cs.umd.edu

Abstract

Behavioral software contracts are a widely used mechanism for governing the flow of values between components. However, run-time monitoring and enforcement of contracts imposes significant overhead and delays discovery of faulty components to run-time.

To overcome these issues, we present *soft contract verification*, which aims to statically prove either complete or partial contract correctness of components, written in an untyped, higher-order language with first-class contracts. Our approach uses higher-order symbolic execution, leveraging contracts as a source of symbolic values including unknown behavioral values, and employs an updatable heap of contract invariants to reason about flow-sensitive facts. We prove the symbolic execution soundly approximates the dynamic semantics and that *verified programs can't be blamed*.

The approach is able to analyze first-class contracts, recursive data structures, unknown functions, and control-flow-sensitive refinements of values, which are all idiomatic in dynamic languages. It makes effective use of an off-the-shelf solver to decide problems without heavy encodings. The approach is competitive with a wide range of existing tools—including type systems, flow analyzers, and model checkers—on their own benchmarks.

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

Keywords Higher-order contracts; symbolic execution

1. Static verification for dynamic languages

Contracts (Meyer 1991; Findler and Felleisen 2002) have become a prominent mechanism for specifying and enforcing invariants in dynamic languages (Disney 2013; Plosch 1997; Austin et al. 2011; Strickland et al. 2012; Hickey et al. 2013). They offer the expressivity and flexibility of programming in a dynamic language, while still giving strong guarantees about the interaction of components. However, there are two downsides: (1) contract monitoring is expensive, often prohibitively so, which causes programmers to write more lax specifications, compromising correctness for efficiency; and (2) contract violations are found only at run-time, which delays discovery of faulty components with the usual negative engineering consequences.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

ACM 978-1-4503-2873-9/14/09…$15.00.
http://dx.doi.org/10.1145/2628136.2628156

Static verification of contracts would empower programmers to state stronger properties, get immediate feedback on the correctness of their software, and avoid worries about run-time enforcement cost since, once verified, contracts could be removed. All-or-nothing approaches to verification of typed functional programs has seen significant advances in the recent work on static contract checking (Xu et al. 2009; Xu 2012; Vytiniotis et al. 2013), refinement type checking (Terauchi 2010; Zhu and Jagannathan 2013; Vazou et al. 2013, 2014), and model checking (Kobayashi 2009b; Kobayashi et al. 2010, 2011). However, the highly dynamic nature of untyped languages makes verification more difficult.

Programs in dynamic languages are often written in idioms that thwart even simple verification methods such as type inference. Moreover, contracts themselves are written within the host language in the same idiomatic style. This suggests that moving beyond all-or-nothing approaches to verification is necessary.

In previous work (Tobin-Hochstadt and Van Horn 2012), we proposed an approach to *soft contract verification*, which enables piecemeal and modular verification of contracts. This approach augments a standard reduction semantics for a functional language with contracts and modules by endowing it with a notion of "unknown" values refined by sets of contracts. Verification is carried out by executing programs on abstract values.

To demonstrate the essence of the idea, consider the following contrived, but illustrative example. Let `pos?` and `neg?` be predicates for positive and negative integers. Contracts can be arbitrary predicates, so these functions are also contracts. Consider the following contracted function (written in Lisp-like notation):

```
(f : pos? → neg?)       ; contract
(define (f x) (* x -1)) ; function
```

We can verify this program by (symbolically) running it on an "unknown" input. Checking the domain contract refines the input to be an unknown satisfying the set of contracts {`pos?`}. By embedding some basic facts about `pos?`, `neg?`, and `-1` into the reduction relation for `*`, we conclude `(* {pos?} -1)` $\longmapsto$ {`neg?`}, and voilà, we've shown once and for all `f` meets its contract obligations and cannot be blamed. We could therefore soundly eliminate any contract which blames `f`, in this case `neg?`.

This approach is simple and effective for many programs, but suffers from several shortcomings, which we solve in this paper:

Solver-aided reasoning: While embedding symbolic arithmetic knowledge for specific, known contracts works for simple examples, it fails to reason about arithmetic generally. Contracts often fail to verify because equivalent formulations of contracts are not hard-coded in the semantics of primitives. Many systems address this issue by incorporating an SMT solver. However, for a higher-order language, solver integration is often achieved by reasoning in a theory of uninterpreted functions or semantic embeddings (Knowles and Flanagan 2010; Rondon et al. 2008; Vytiniotis et al. 2013).

In this paper, we observe that higher-order contracts can be effectively verified using only a simple first-order solver. The key in-

sight is that contracts delay higher-order checks and failures always occur with a first order witness. By relying on a (symbolic) semantic approach to carry out higher-order contract monitoring, we can use an SMT solver to reason about integers without the need for sophisticated encodings. (Examples in §2.3.)

Flow sensitive reasoning: Just as our semantic approach decomposes higher-order contracts into first-order properties, first-order contracts naturally decompose into conditionals. Our prior approach fails to reason effectively about conditionals, requiring contract checks to be built-in to the semantics. As a result, even simple programs with conditionals fail to verify:

```
(g : int? → neg?)
(define (g x) (if (pos? x) (f x) (f 8)))
```

This is because the true-branch call to `f` is `(f {int?})` by substitution, although we know from the guard that `x` satisfies `pos?`.

In this paper, we observe that flow-sensitivity can be achieved by replacing substitution with *heap-allocated* abstract values. These heap addresses are then refined as they flow through predicates and primitive operations, with no need for special handling of contracts (§2.2). As a result, the system is not only effective for contract verification, but can also handle safety verification for programs with no contracts at all.

First-class contracts: Pragmatic contract systems enable first-class contracts so new combinators can be written as functions that consume and produce contracts. But to the best of our knowledge, no verification system currently supports first class contracts (or refinements), and in most approaches it appears fundamentally difficult to incorporate such a notion.

Because we handle contracts (and all other features) by *execution*, first-class contracts pose no significant technical challenge and our system reasons about them effectively (§2.5).

Converging for non-tail recursion: Of course, simply executing programs has a fundamental drawback—it will fail to terminate in many cases, and when the inputs are unknown, execution will almost always diverge. Our prior work used a simple loop detection algorithm that handled only tail-recursive functions. As a result, even simple programs operating over inductive data timed out.

In this paper, we accelerate the convergence of programs by identifying and approximating regular accumulation of evaluation contexts, causing common recursive programs to converge on unknown values, while providing precise predictions (§2.4). As with the rest of our approach, this happens during execution and is therefore robust to complex, higher-order control flow.

Combining these techniques yields a system competitive with a diverse range of existing powerful static checkers, achieving many of their strengths in concert, while balancing the benefits of static contract verification with the flexibility of dynamic enforcement.

We have built a prototype soft verification engine, which we dub SCV, based on these ideas and used it to evaluate the approach (§4). Our evaluation demonstrates that the approach can verify properties typically reserved for approaches that rely on an underlying type system, while simultaneously accommodating the dynamism and idioms of untyped programming languages. We take examples from work on soft typing (Cartwright and Fagan 1991; Wright and Cartwright 1997), type systems for untyped languages (Tobin-Hochstadt and Felleisen 2010), static contract checking (Xu et al. 2009; Xu 2012), refinement type checking (Terauchi 2010), and model checking of higher-order functional languages (Kobayashi 2009b; Kobayashi et al. 2010, 2011).

SCV can prove all contract checks redundant for almost all of the examples taken from this broad array of existing program analysis and type checking work, and can handle many of the tricky higher-order verification problems demonstrated by other systems. In other words, our approach is competitive with type systems, model checkers, and soft typing systems on each of their chosen benchmarks—in contrast, work on higher-order model checking does not handle benchmarks aimed at soft typing or occurrence typing, and vice versa. In the cases where SCV does not prove the complete absence of contract errors, the vast majority of possible dynamic errors are ruled out, justifying numerous potential optimizations. Over this corpus of programs, 99% of the contract and run-time type checks are proved safe, and could be eliminated.

We also evaluate the verification of three small interactive video games which use first-class and dependent contracts pervasively. The results show the subsequent elimination of contract monitoring has a dramatic effect: from a factor speed up of 7 in one case, to three orders of magnitude in the others. In essence, these results show the games are infeasible without contract verification.

2. Worked examples

We now present the main ideas of our approach through a series of examples taken from work on other verification techniques, starting from the simplest and working up to a complex object encoding.

2.1 Higher-order symbolic reasoning

Consider the following simple function that transforms functions on even integers into functions on odd integers. It has been ascribed this specification as a contract, which can be monitored at run-time.

```
(e2o : (even? → even?) → (odd? → odd?))
(define (e2o f)
  (λ (n) (- (f (+ n 1)) 1)))
```

A contract monitors the flow of values between components. In this case, the contract monitors the interaction between the context and the `e2o` function. It is easy to confirm that `e2o` is correct with respect to the contract; `e2o` holds up its end of the agreement, and therefore cannot be blamed for any run-time failures that may arise. The informal reasoning goes like this: First assume `f` is an `even? → even?` function. When applied, we must ensure the argument is even (otherwise `e2o` is at fault), but may assume the result is even (otherwise the context is at fault). Next assume `n` is odd (otherwise the context is at fault) and ensure the result is odd (otherwise `e2o` is at fault). Since `(+ n 1)` is even when `n` is odd, `f` is applied to an even argument, producing an even result. Subtracting one therefore gives an odd result, as desired.

This kind of reasoning mimics the step-by-step computation of `e2o`, but rather than considering some particular inputs, it considers these inputs symbolically to verify all possible executions of `e2o`. We systematize this kind of reasoning by augmenting a standard reduction semantics for contracts with symbolic values that are refined by sets of contracts. At first approximation, the semantics includes reductions such as:

$$(+\ \{\texttt{odd?}\}\ 1) \longmapsto \{\texttt{even?}\}, \text{ and}$$
$$(\{\texttt{even?} \to \texttt{even?}\}\ \{\texttt{even?}\}) \longmapsto \{\texttt{even?}\}.$$

This kind of symbolic reasoning mimics a programmer's informal intuitions which employ contracts to refine unknown values and to verify components meet their specifications. If a component cannot be blamed in the symbolic semantics, we can safely conclude it cannot be blamed in general.

2.2 Flow sensitive reasoning

Programmers using untyped languages often use a mixture of type-based and flow-based reasoning to design programs. The analysis naturally takes advantage of type tests idiomatic in dynamic languages even when the tests are buried in complex expressions. The

following function taken from work on occurrence typing (Tobin-Hochstadt and Felleisen 2010) can be proven safe using our symbolic semantics:

```
(f : (or/c int? str?) cons? → int?)
(define (f x p)
  (cond
    [(and (int? x) (int? (car p))) (+ x (car p))]
    [(int? (car p)) (+ (str-len x) (car p))]
    [else 0]))
```

Here, `int?`, `str?`, and `cons?` are type predicates for integers, strings, and pairs, respectively. The contract `(or/c int? str?)` uses the `or/c` contract combinator to construct a contract specifying a value is either an integer or a string.

A programmer would convince themselves this program was safe by using the control dominating predicates to refine the types of `x` and `(car p)` in each branch of the conditional.[1] Our symbolic semantics accommodates exactly this kind of reasoning in order to verify this example. However, there is a technical challenge here. A straightforward substitution-based semantics would not reflect the flow-sensitive facts. Focusing just on the first clause, a substitution model would give:

```
(cond
  [(and (int? {(or/c int? str?)}) (int? (car {cons?})))
   (+ {(or/c int? str?)} (car {cons?}))] …)
```

At this point, it's too late to communicate the refinement of these sets implied by the test evaluating to true, so the semantics would report the contract on `+` potentially being violated because the first argument may be a string, and the second argument may be anything. We overcome this challenge by modelling symbolic values as heap-allocated sets of contracts. When predicates and data structure accessors are applied to heap addresses, we refine the corresponding sets to reflect what must be true. So the program is modelled as:

```
(cond
  [(and (int? L1) (int? (car L2)))
   (+ L1 (car L2))] …)
```
where $L_1 \mapsto \{$`(or/c int? string?)`$\}$, $L_2 \mapsto \{$`cons?`$\}$

In the course of evaluating the test, we get to `(int? L1)`, the semantics conceptually forks the evaluator and refines the heap:

$$\texttt{(int? } L_1\texttt{)} \longmapsto \texttt{true}, \text{ where } L_1 \mapsto \{\texttt{int?}\}$$
$$\longmapsto \texttt{false}, \text{ where } L_1 \mapsto \{\texttt{string?}\}$$

Similar refinements to L_2 are communicated through the heap for `(int? (car L2))`, thereby making `(+ L1 (car L2))` safe. This simple idea is effective in achieving flow-based refinements. It naturally handles deeply nested and inter-procedural conditionals.

2.3 Incorporating an SMT solver

The techniques described so far are highly effective for reasoning about functions and many kinds of recursive data structures. However, effective reasoning about many kinds of base values, such as integers, requires sophisticated domain-specific knowledge. Rather than build such a tool ourselves, we defer to existing high-quality solvers for these domains. Unlike many solver-aided verification tools, however, we use the solver *only* for queries on base values, rather than attempting to encode a rich, higher-order language into one that is accepted by the solver.

[1] The call to `str-len` is safe because `(and (int? x) (int? (car p)))` being false and `(int? (car p))` being true implies that `(int? x)` is false, which in turns implies `x` is a string as enforced by `f`'s contract.

To demonstrate our approach, we take an example (`intro3`) from work on model checking higher-order programs (Kobayashi et al. 2010).

```
(>/c : int? → any → bool?)
(define (>/c lo) (λ (x) (and (int? x) (> x lo))))

(define (f x g) (g (+ x 1)))

(h : [x : int?] → [y : (>/c x)] → (>/c y))
(define (h x) ...) ; unknown definition

(main : int? → (>/c 0))
(define (main n) (if (≥ n 0) (f n (h n)) 1))
```

In this program, we define a contract combinator (`>/c`) that creates a check for an integer from a lower bound; a helper function `f`, which comes without a contract; and an *unknown* function `h` that given an integer `x`, returns a function mapping some number `y` that is greater than `x` to an answer greater than `y`—here `h`'s specification is given, but not its implementation. (Note `h`'s contract is dependent.) We verify `main`'s correctness, which means it definitely returns a positive integer and does not violate `h`'s contract.

According to its contract, `main` is passed an integer `n`. If `n` is negative, `main` returns `1`, satisfying the contract. Otherwise the function applies `f` to `n` and `(h n)`. Function `h`, by its contract, returns another function that requires a number greater than `n`. Examining `f`'s definition, we see `h` (now bound to `g`) is eventually applied to `(+ n 1)`. Let n_1 be the result of `(+ n 1)`. And by `h`'s contract, we know the answer is another integer greater than `(+ n 1)`. Let us name this answer n_2. In order to verify that `main` satisfies contract `(>/c 0)`, we need to verify that n_2 is a positive integer.

Once `f` returns, the heap contains several addresses with contracts:

$$\begin{array}{lcl} \texttt{n} & \mapsto & \{\texttt{int?}, \texttt{(≥/c 0)}\} \\ n_1 & \mapsto & \{\texttt{int?}, \texttt{(=/c (+ n 1))}\} \\ n_2 & \mapsto & \{\texttt{int?}, \texttt{(>/c } n_1\texttt{)}\} \end{array}$$

We then translate this information to a query for an external solver:

```
n, n1, n2: INT;
ASSERT n ≥ 0;
ASSERT n1 = n + 1;
ASSERT n2 > n1;
QUERY n2 > 0;
```

Solvers such as CVC4 (Barrett et al. 2011) and Z3 (De Moura and Bjørner 2008) easily verify this implication, proving `main`'s correctness.

Refinements such as `(≥/c 0)` are generated by *primitive* applications `(≥ x 0)`, and queries are generated from translation of the heap, not arbitrary expressions. This has a few consequences. First, by the time we have value `v` satisfying predicate `p` on the heap, we know that `p` terminates successfully on `v`. Issues such as errors (from `p` itself) or divergence are handled elsewhere in other evaluation branches. Second, we only need to translate a small set of simple, well understood contracts—not arbitrary expressions. Evaluation naturally breaks down complex expressions, and properties are discovered even when they are buried in complex, higher-order functions. Given a translation for `(>/c 0)`, the analysis automatically takes advantage of the solver even when the predicate contains `>` in a complex way, such as `(λ (x) (or (> x 0) e)` where e is an arbitrary expression. Predicates that lack translations to SMT only reduce precision, never soundness.

2.4 Converging for non-tail recursion

The techniques sketched above provide high precision in the examples considered, but simply executing programs on abstract values is unlikely to terminate in the presence of recursion. When an

abstract value stands for an infinite set of concrete values, execution may proceed infinitely, building up an ever-growing evaluation context. To tackle this problem, we *summarize* this context to coalesce repeated structures and enable termination on many recursive programs. Although guaranteed termination is not our goal, the empirical results (§4) demonstrate that the method is effective in practice.

The following example program is taken from work on model checking of higher-order functional programs (Kobayashi et al. 2010), and demonstrates checking non-trivial safety properties on recursive functions. Note that no loop invariants need be provided by the user.

```
(main : (and/c int? ≥0?) → (and/c int? ≥0?))
(define (main n)
  (let ([l (make-list n)])
    (if (> n 0) (car (reverse l empty)) 0)))

(define (reverse l ac)
  (if (empty? l) ac
      (reverse (cdr l) (cons (car l) ac))))

(define (make-list n)
  (if (= n 0) empty
      (cons n (make-list (- n 1)))))
```

Again, we aim to verify both the specified contract for `main` as well as the preconditions for primitive operations such as `car`. Most significantly, we need to verify that `(reverse l empty)` produces a non-empty list (so that `car` succeeds) and that its first element is a positive integer. The local functions `reverse` and `make-list` do not come with a contract.

This problem is more challenging than the original OCaml version of the same program, due to the lack of types. This program represents a common idiom in dynamic languages: not all values are contracted, and there is no type system on which to piggy-back verification. In addition, programmers often rely on inter-procedural reasoning to justify their code's correctness, as here with `reverse`.

We verify `main` by applying it to an abstract (unknown) value n_1. The contract ensures that within the body, n_1 is a non-negative integer.

The integer n_1 is first passed to `make-list`. The comparison `(= n1 0)` non-deterministically returns `true` and `false`, updating the information known about n_1 to be either `0` or `(>/c 0)` in each corresponding case. In the first case, `make-list` returns `empty`. In the second case, `make-list` proceeds to the recursive application `(make-list n2)`, where n_2 is the abstract non-negative integer obtained from evaluating `(- n1 1)`. However, `(make-list n2)` is identical to the original call `(make-list n1)` up to renaming, since both n_1 and n_2 are non-negative. Therefore, we pause here and use a summary of `make-list`'s result instead of continuing in an infinite loop.

Since we already know that `empty` is one possible result of `(make-list n1)`, we use it as the result of `(make-list n2)`. The application `(make-list n1)` therefore produces the pair $\langle n_1, \texttt{empty}\rangle$, which is another answer for the original application. We could continue this process and plug this new result into the pending application `(make-list n2)`. But by observing that the application produces a list of one positive integer when the recursive call produces `empty`, we approximate the new result $\langle n_1, \texttt{empty}\rangle$ to a non-empty list of positive integers, and then use this approximate answer as the result of the pending application `(make-list n2)`. This then induces another result for `(make-list n1)`, a list of two or more positive integers, but this is subsumed by the previous answer of non-empty integer list. We have now discovered *all* possible return values of `make-list` when applied to a non-negative integer: it maps `0` to `empty`, and positive integers to a non-empty list of positive integers.

Although our explanation made use of the order, the soundness of analyzing `make-list` does not depend on the order of exploring non-deterministic branches. Each recursive application with repeated arguments generates a waiting context, and each function return generates a new case to resume. There is an implicit worklist algorithm in the modified semantics (§3.8).

When `make-list` returns to `main`, we have two separate cases: either n_1 is 0 and `l` is `empty`, or n_1 is positive and `l` is non-empty. In the first case, `(> n1 0)` is false and `main` returns `0`, satisfying the contract. Otherwise, `main` proceeds to reversing the list before taking its first element.

Using the same mechanism as with `make-list`, the analysis infers that `reverse` returns a non-empty list when either of its arguments (`l` or `acc`) is non-empty. In addition, `reverse` only receives arguments of proper lists, so all partial operations on `l` such as `car` and `cdr` are safe when `l` is not `empty`, without needing an explicit check. The function eventually returns a non-empty list of integers to `main`, justifying `main`'s call to the partial function `car`, producing a positive integer. Thus, `main` never has a run-time error in any context.

While this analysis makes use of the implementation of `make-list` and `reverse`, that does not imply that it is whole-program. Instead, it is *modular* in its use of unknown values abstracting arbitrary behavior. For example, `make-list` could instead be an abstract value represented by a contract that always produces lists of integers. The analysis would still succeed in proving all contracts safe except the use of `car` in `main`—this shows the flexibility available in choosing between precision and modularity. In addition, the analysis does not have to be perfectly precise to be useful. If it successfully verifies most contracts in a module, that already greatly improves confidence about the module's correctness and justifies the elimination of numerous expensive dynamic checks.

2.5 Putting it all together

The following example illustrates all aspects of our system. For this, we choose a simple encoding of classes as functions that produce objects, where objects are again functions that respond to messages named by symbols. We then verify the correctness of a *mixin*: a function from classes to classes. The `vec/c` contract enforces the interface of a 2D-vector class whose objects accept messages `'x`, `'y`, and `'add` for extracting components and vector addition.

```
(define vec/c
  ([msg : (one-of/c 'x 'y 'add)]
   → (match msg
       [(or 'x 'y) real?]
       ['add (vec/c → vec/c)])))
```

This definition demonstrates several powerful contract system features which we are able to handle:

- contracts are first-class values, as in the definition of `vec/c`,
- contracts may include arbitrary predicates, such as `real?`,
- contracts may be recursive, as in the contract for `'add`,
- function contracts may express *dependent* relationships between the domain and range—the contract of the result of method selection for `vec/c` depends on the method chosen.

Suppose we want to define a mixin that takes any class that satisfies the `vec/c` interface and produces another class with added vector operations such as `'len` for computing the vector's length. The `extend` function defines this mixin, and `ext-vec/c` specifies the new interface. We verify that `extend` violates no contracts and returns a class that respects specifications from `ext-vec/c`.

```
(extend : (real? real? → vec/c)
          → (real? real? → ext-vec/c))
(define (extend mk-vec)
  (λ (x y)
    (let ([vec (mk-vec x y)])
      (λ (m)
        (match m
          ['len
           (let ([x (vec 'x)] [y (vec 'y)])
             (sqrt (+ (* x x) (* y y))))]
          [_ (vec m)])))))

(define ext-vec/c
  ([msg : (one-of/c 'x 'y 'add 'len)]
   → (match msg
       [(or 'x 'y) real?]
       ['add (vec/c → vec/c)]
       ['len (and/c real? (≥/c 0))])))
```

To verify `extend`, we provide an arbitrary value, which is guaranteed by its contract to be a class matching `vec/c`. The mixin returns a new class whose objects understand messages `'x`, `'y`, `'add`, and `'len`. This new class defines method `'len` and relies on the underlying class to respond to `'x`, `'y`, and `'add`. Because the old class is constrained by contract `vec/c`, the new class will not violate its contract when responding to messages `'x`, `'y`, and `'add`.

For the `'len` message, the object in the new vector class extracts its components as abstract numbers `x` and `y`, according to interface `vec/c`. It then computes their squares and leaves the following information on the heap:

$$\begin{array}{lcl} \texttt{x}^2 & \mapsto & \{\texttt{real?}, \texttt{(=/c (* x x))}\} \\ \texttt{y}^2 & \mapsto & \{\texttt{real?}, \texttt{(=/c (* y y))}\} \\ \texttt{s} & \mapsto & \{\texttt{real?}, \texttt{(=/c (+ x}^2\ \texttt{y}^2\texttt{))}\} \end{array}$$

Solvers such as Z3 (De Moura and Bjørner 2008) can handle simple non-linear arithmetic and verify that the sum `s` is non-negative, thus the `sqrt` operation is safe. Execution proceeds to take the square root—now called `l`—and refines the heap with the following mapping:

$$\texttt{l} \quad \mapsto \quad \{\texttt{real?}, \texttt{(=/c (sqrt s))}\}$$

When the method returns, its result is checked by contract `ext-vec/c` to be a non-negative number. We again rely on the solver to prove that this is the case.

Therefore, `extend` is guaranteed to produce a new class that is correct with respect to interface `vec-ext/c`, justifying the elimination of expensive run-time checks. In a Racket program computing the length of 100000 random vectors, eliminating these contracts results in a 100-fold speed-up. While such dramatic results are unlikely in full programs, measurements of existing Racket programs suggests that 50% speedups are possible (Strickland et al. 2012).

3. A Symbolic Language with Contracts

In this section, we give a reduction system describing the core of our approach. Symbolic λ_{C} is a model of a language with first-class contracts and *symbolic values*. We first present the semantics, including handling of primitives and unknown functions. We then describe how the handling of primitive values integrates with external solvers. Finally, we show an abstraction of our symbolic system to accelerate convergence. For each abstraction, we relate concrete and symbolic programs and prove a soundness theorem.

At a high level, the key idea of our semantics is that abstract values behave non-deterministically in all possible ways that concrete values might behave. Furthmore, abstract values can be bounded by specifications in the form of contracts that limit these behaviors. As a result, an operational semantics for abstract values explores all the ways that the concrete program under consideration might be used.

Given this operational semantics, we can then examine the results of evaluation to see if any results are errors blaming the components we wish to verify. If they do not, then our soundness theorem implies that there are no ways for the component to be blamed, regardless of what other parts of the program do. Thus, we have verified the component against its contract in all contexts. We make this notion precise in section 3.6.

Programs	p, q	::=	$\overrightarrow{m}\ e$
Modules	m	::=	$\texttt{(module}\ f\ u_c\ u\texttt{)}$
Expressions	e, c, d	::=	$v \mid x^\ell \mid e\ e^\ell \mid o\ \overrightarrow{e}^\ell \mid \texttt{if}\ e\ e\ e \mid c \texttt{→} \lambda x.d$
		$\mid$	$\texttt{mon}^{\ell,\ell}_\ell(c, e) \mid \texttt{blame}^\ell_\ell \mid \texttt{assume}(v, v)$
Pre-values	u	::=	$\lambda x.e \mid b \mid \langle v,v\rangle \mid v \texttt{→} \lambda x.c \mid \bullet$
Values	v	::=	$a \mid u\,/\,\overrightarrow{v}$
Base values	b	::=	$\texttt{true} \mid \texttt{false} \mid n \mid \texttt{empty}$
Operations	o	::=	$o_? \mid \texttt{cons} \mid \texttt{car} \mid \texttt{cdr} \mid \texttt{add1} \mid \texttt{+} \mid \texttt{=}$
		$\mid$	$\ldots$
Predicates	$o_?$	::=	$\texttt{num?} \mid \texttt{false?} \mid \texttt{cons?} \mid \texttt{empty?}$
		$\mid$	$\texttt{proc?} \mid \texttt{dep?}$
Variables	x, f, ℓ	$\in$	*identifier*
Addresses	a	$\in$	*address*

Figure 1. Syntax of Symbolic λ_{C}

3.1 Syntax of Symbolic λ_{C}

Our initial language models the functional core of many modern dynamic languages, extended with behavioral, first-class contracts, as well as symbolic values. The abstract syntax is shown in figure 1. Syntax needed only for symbolic execution is highlighted in gray; we discuss it after the syntax of concrete programs.

A program p is a sequence of module definitions followed by a top-level expression which may reference the modules. Each module m has a name f and exports a single value u with behavior enforced by contract u_c. (Generalizing to multiple-export modules is straightforward.)

Expressions include standard forms such as values, variable and module references, applications, and conditionals, as well as those for constructing and monitoring contracts. Contracts are first-class values and can be produced by arbitrary expressions. For clarity, when an expression plays the role of a contract, we use the metavariable c and d, rather than e. A *dependent* function contract ($c \texttt{→} \lambda x.d$) monitors a function's argument with c and its result with the contract produced by applying $\lambda x.d$ to the argument.

A contract violation at run-time causes *blame*, an error with information about who violated the contract. We write $\texttt{blame}^\ell_{\ell''}$ to mean module ℓ is blamed for violating the contract from ℓ''. The form ($\texttt{mon}^{\ell,\ell'}_{\ell''}(c, e)$) monitors expression e with contract c, with ℓ being the positive party, ℓ' the negative party, and ℓ'' the source of the contract. The system blames the positive party if e produces a value violating c, and the negative one if e is misused by the *context* of the contract check. To make context information available at run-time, we annotate references and applications with labels indicating the module they appear in, or † for the top-level expression. For example, $x^\dagger$ denotes a reference to the name x from the top level, and $(\texttt{add1}\ x)^\ell$ denotes an addition inside module ℓ. When a module ℓ causes a primitive error, such as applying 5, we also write $\texttt{blame}^\ell_\Lambda$, indicating that it violates a contract with the language. We omit labels when they are irrelevant or can be inferred.

Pre-values u—extended to values below—include abstractions, base values, pairs of values, and dependent contracts with domain components evaluated. Base values include numbers, booleans, and the empty list. Primitive operations over values are standard, including predicates $o_?$ for dynamic testing of data types.

To reason about absent components, we equip λ_C with *unknown*, or *symbolic values*, which abstract over multiple concrete values exhibiting a range of behavior. An unknown value $\bullet$ stands for any value in the language. For soundness, execution must account for *all* possible concretizations of abstract values, and reduction becomes non-deterministic. As the program proceeds through tests and contract checks, more assumptions can be made about abstract values. To remember these assumptions, we take the *pre-values* and refine each with a set of contracts it is known to satisfy, written $u/\overrightarrow{v}$.

Finally, to track refinements of unknown values, we use heap addresses a as symbolic values and track them in a heap, which is a finite map from addresses to refined pre-values:

$$\text{Heaps} \quad \sigma ::= \overrightarrow{\langle a, u/\overrightarrow{v} \rangle}.$$

The heap σ maps addresses allocated for unknown values to refinements expressed as contracts; these refinements are updated during reduction and represent upper bounds on what they might be at runtime. Intuitively, any possible concrete execution can be obtained by substituting addresses with concrete values within bounds specified by the heap. We omit refinements when they are empty or irrelevant.

3.2 Semantics of Symbolic λ_C

We now turn to the reduction semantics for Symbolic λ_C, which combines standard rules for untyped languages with behavior for unknown values. Reduction is defined as a relation on states, parameterized by a module context:

$$\overrightarrow{m} \vdash \varsigma \longmapsto \varsigma'$$

States consist either of an expression paired with a heap, or blame:

$$\text{States} \quad \varsigma ::= (e, \sigma) \mid \texttt{blame}^{\ell}_{\ell}.$$

We present the rules inline; a full version of all rules is given in the appendix of the the accompanying technical report (Nguyễn et al. 2014). In the inline presentation of rules, we systematically omit labels in contracts, these are presented in the full rules. We omit the module context whenever it is irrelevant.

3.2.1 Basic rules

Applications of primitives are interpreted by a δ relation, which maps operations, arguments and heaps to results and new heaps.

$$\textit{Apply-Primitive} \quad \frac{\delta(\sigma, o\,\overrightarrow{v},) \ni \varsigma}{(o\ \overrightarrow{v}), \sigma \longmapsto \varsigma}$$

The use of a δ relation in reduction semantics is standard, but typically it is a function and is independent of the heap. We make δ dependent on the heap in order to use and update the current set of invariants; we make it a relation, since it may behave non-deterministically on unknown values. For example, in interpreting `(> • 5)`, the δ relation will produce two results: one `true`, with an updated heap to reflect the unknown value is `(>/c 5)`; the other `false`, with a heap reflecting the opposite. The δ relation is thus the hub of the verification system and a point of interaction with the SMT solver. It is described in more detail in section 3.3.

Applications of λ-abstractions follow standard β-reduction; applications of non-functions result in blame.

$$\textit{Apply-Function} \quad \frac{}{((\lambda x.e)\ v), \sigma \longmapsto [v/x]e, \sigma} \qquad \textit{Apply-Non-Function} \quad \frac{\delta(\sigma, \texttt{proc?}, v) \ni (\texttt{false}, \sigma')}{(v\ v'), \sigma \longmapsto \texttt{blame}, \sigma'}$$

Notice that the δ relation is employed to determine whether the value in operator position is a function using the `proc?` primitive. (Non-functions include concrete numbers and booleans as well as abstract values known to exclude functions; application of abstract values that may be functions is described in section 3.2.3.)

Conditionals follow a common treatment in untyped languages in which values other than `false` are considered true.

$$\textit{If-True} \quad \frac{\delta(\sigma, \texttt{false?}, v) \ni (\texttt{false}, \sigma')}{\text{if } v\ e_1\ e_2, \sigma \longmapsto e_1, \sigma'} \qquad \textit{If-False} \quad \frac{\delta(\sigma, \texttt{false?}, v) \ni (\texttt{true}, \sigma')}{\text{if } v\ e_1\ e_2, \sigma \longmapsto e_2, \sigma'}$$

Just as in the case of *Apply-Non-Function*, the interpretation of conditionals uses the δ relation to determine whether `false?` holds, which takes into account all of the knowledge accumulated in σ and in either branch that is taken, updates the current knowledge to reflect whether `false?` of v holds. This is the mechanism by which control-flow based refinements are enabled.

The two rules for module references reflect the approach in which contracts are treated as *boundaries* between components (Dimoulas et al. 2011): a module self-reference incurs no contract check, while cross-module references are protected by the specified contract.

$$\textit{Module-Self-Reference} \quad \frac{(\texttt{module}\ f\ u_c\ u) \in \overrightarrow{m}}{\overrightarrow{m} \vdash f^f, \sigma \longmapsto u, \sigma} \qquad \textit{Module-External-Reference} \quad \frac{(\texttt{module}\ f\ u_c\ u) \in \overrightarrow{m} \qquad f \neq \ell}{\overrightarrow{m} \vdash f^{\ell}, \sigma \longmapsto \texttt{mon}(u_c, u), \sigma}$$

Finally, any state that is stuck with blame inside an evaluation context transitions to a final blame state that discards the surrounding context and heap.

$$\textit{Halt-Blame} \quad \frac{}{\mathcal{E}[\texttt{blame}], \sigma \longmapsto \texttt{blame}}$$

Evaluation contexts as defined as follows:

$$\begin{array}{rcl} \mathcal{E} ::= & & [\,] \mid \mathcal{E}\ e \mid v\ \mathcal{E} \mid o\ \overrightarrow{v}\ \mathcal{E}\ \overrightarrow{e} \mid \text{if } \mathcal{E}\ e\ e \\ & \mid & \texttt{mon}(\mathcal{E}, e) \mid \texttt{mon}(v, \mathcal{E}) \mid \mathcal{E} \texttt{ @ } \lambda x.e \end{array}$$

3.2.2 Contract monitoring

Contract monitoring follows existing operational semantics for contracts (Findler and Felleisen 2002), with extensions to handle and refine symbolic values.

There are several cases for checking a value against a contract. If the contract is not a function contract, we say it is *flat*, denoting a first-order property to be checked immediately. We thus expand the checking expression to a conditional.

$$\textit{Monitor-Flat-Contract} \quad \frac{\delta(\sigma, \texttt{dep?}, v_c) \ni (\texttt{false}, \sigma') \qquad \sigma' \vdash v : v_c\,\textbf{?}}{\texttt{mon}(v_c, v), \sigma \longmapsto \text{if } (v_c\ v)\ \texttt{assume}(v, v_c)\ \texttt{blame}, \sigma'}$$

Since contracts are first-class, they can also be abstract values; we rely on δ to determine whether a value is a flat contract by using (the negation of) the predicate for dependent contracts, `dep?`, instead of examining the syntax. This rule is standard except for the use of $\texttt{assume}(v, v_c)$ and the $(\cdot \vdash \cdot : \cdot\,\textbf{?})$ judgment. The $\texttt{assume}(v, v_c)$ form, which would normally just be v, dynamically refines value v and the heap to indicate that v satisfies v_c; `assume` is discussed further in section 3.2.3. The judgment $\sigma' \vdash v : v_c\,\textbf{?}$, which would normally just be omitted, indicates that the contract v_c cannot

be statically judged to either pass or fail for v, which is why the predicate must be applied. This judgment and its closely related counterparts ($\cdot \vdash \cdot : \cdot \checkmark$) and ($\cdot \vdash \cdot : \cdot \times$), which statically prove a value must or must not satisfy a given contract respectively, are discussed in section 3.4. If a flat contract can be statically proved or refuted, monitoring can be short-circuited.

$$\textit{Monitor-Proved}\quad \frac{\delta(\sigma, \texttt{dep?}, v_c) \ni (\texttt{false}, \sigma') \qquad \sigma' \vdash v : v_c \checkmark}{\texttt{mon}(v_c, v), \sigma \longmapsto v, \sigma'}$$

$$\textit{Monitor-Refuted}\quad \frac{\delta(\sigma, \texttt{dep?}, v_c) \ni (\texttt{false}, \sigma') \qquad \sigma' \vdash v : v_c \times}{\texttt{mon}(v_c, v), \sigma \longmapsto \texttt{blame}, \sigma'}$$

Monitoring a function contract against a function is interpreted the standard η-expansion of contracts.

$$\textit{Monitor-Function-Contract}\quad \frac{\delta(\sigma, \texttt{proc?}, v) \ni (\texttt{true}, \sigma')}{\texttt{mon}(v_c \to \lambda x.d, v), \sigma \longmapsto \lambda x.\texttt{mon}(d, (v\ \texttt{mon}(v_c, x))), \sigma'}$$

Monitoring a function contract against a non-function results in an error.

$$\textit{Monitor-Non-Function}\quad \frac{\delta(\sigma, \texttt{dep?}, v_c) \ni (\texttt{true}, \sigma_1) \qquad \delta(\sigma_1, \texttt{proc?}, v) \ni (\texttt{false}, \sigma_2)}{\texttt{mon}(v_c, v), \sigma \longmapsto \texttt{blame}, \sigma_2}$$

When a dependent contract is represented by a address in the heap, we look up the address and use the result.

$$\textit{Monitor-Unknown-Function-Contract}\quad \frac{\delta(\sigma, \texttt{dep?}, a) \ni (\texttt{true}, \sigma_1) \qquad \delta(\sigma_1, \texttt{proc?}, v) \ni (\texttt{true}, \sigma_2) \qquad \sigma_2(a) = v_c \to \lambda x.d}{\texttt{mon}(a, v), \sigma \longmapsto \lambda x.\texttt{mon}(d, v\ \texttt{mon}(v_c, x)), \sigma_2}$$

3.2.3 Handling unknown values

The final set of reduction rules concern unknown values and refinements.

$$\textit{Refine-Concrete}\quad \frac{u \neq \bullet}{u, \sigma \longmapsto u/\emptyset, \sigma} \qquad \textit{Refine-Unknown}\quad \frac{a \notin \mathit{dom}(\sigma)}{\bullet, \sigma \longmapsto a, \sigma[a \mapsto \bullet/\emptyset]}$$

These two rules show reduction of pre-values, which initially have no refinement. If the pre-value is unknown, we additionally create a fresh address and add it to the heap.

The assume form uses the refine metafunction to update the heap of refinements to take into account the new information; see figure 2 for the definition of refine.

$$\textit{Assume}\quad \frac{(\sigma', v') = \mathit{refine}(\sigma, v, v_c)}{\texttt{assume}(v, v_c), \sigma \longmapsto v', \sigma'}$$

Refinement is straightforward propagation of known contracts, including expanding values known to be pairs via cons? into pair values, and values known to be function contracts (via dep?) into function contract values.

Finally, we must handle application of unknown values. The first rule simply produces a new unknown value and heap address for the result of a call. If the unknown function came with a contract, this new unknown value will be refined by the contract via reduction.

$$\textit{Apply-Unknown}\quad \frac{\delta(\sigma, \texttt{proc?}, a) \ni (\texttt{true}, \sigma')}{a\ v, \sigma \longmapsto a_a, \sigma'[a_a \mapsto \bullet]} \qquad \textit{Havoc}\quad \frac{\delta(\sigma, \texttt{proc?}, a) \ni (\texttt{true}, \sigma')}{a\ v, \sigma \longmapsto \texttt{havoc}\ v, \sigma'}$$

The second reduction rule for applying an unknown function, labeled *Havoc*, handles the possible dynamic behavior of the unknown function. A value passed to the unknown function may itself be a function with behavior, whose implementation we hope to verify. This function may further be invoked by the unknown function on unknown arguments. To simulate this, we assume *arbitrary* behavior from this unknown function and put the argument in a so-called demonic context implemented by the havoc operation, defined in a module added to every program; the definition is given below.

$$\begin{aligned}
&\mathit{refine}(\sigma, a, v) &&= (\sigma'[a \mapsto v'], a) \\
&&&\quad \text{where } (\sigma', v') = \mathit{refine}(\sigma, \sigma(a), v) \\
&\mathit{refine}(\sigma, \bullet/\overrightarrow{v}, \texttt{cons?}) &&= (\sigma[a_1 \mapsto \bullet][a_2 \mapsto \bullet], \langle a_1, a_2\rangle/\overrightarrow{v}) \\
&&&\quad \text{where } a_1, a_2 \notin \mathit{dom}(\sigma) \\
&\mathit{refine}(\sigma, \bullet/\overrightarrow{v}, \texttt{dep?}) &&= (\sigma[a \mapsto \bullet], a \to \lambda x.\bullet/\overrightarrow{v}) \\
&&&\quad \text{where } a \notin \mathit{dom}(\sigma) \\
&\mathit{refine}(\sigma, u/\overrightarrow{v}, v_i) &&= (\sigma, u/\overrightarrow{v} \cup \{v_i\})
\end{aligned}$$

Figure 2. Refinement for Symbolic λ_C

```
(module havoc (any → λ _.false)
  (λx.amb({(havoc (x •)), (havoc (car x)), (havoc (cdr x))})))
```

$$\begin{aligned}
\mathit{amb}(\{e\}) &= e \\
\mathit{amb}(\{e, e_1, \ldots\}) &= \textsf{if } \bullet\ e\ \mathit{amb}(\{e_1, \ldots\})
\end{aligned}$$

The havoc function never produces useful results; its only purpose is to probe for all potential errors in the value provided. This context, and thus the havoc module, may be blamed for misuse of accessors and applications; we ignore these, as they represent potential failures in *omitted* portions of the program. Using havoc is key to soundness in modular higher-order static checking (Fähndrich and Logozzo 2011; Tobin-Hochstadt and Van Horn 2012); we discuss its role further in section 3.6. Intuitively, precise execution of properly contracted functions prevents havoc from destroying every analysis.

3.3 Primitive operations

Primitive operations are the primary place where unknown values in the heap are refined, in concert with successful contract checks. Figure 3 shows a representative excerpt of δ's definition; the full definition is given in the accompanying technical report.

The first three rules cover primitive predicate checks. Ambiguity never occurs for concrete values, and an abstract value may definitely prove or refute the predicate if the available information is enough for the conclusion. If the proof system cannot decide a definite result for the predicate check, δ conservatively includes *both* answers in the possible results and records assumptions chosen for each non-deterministic branch in the appropriate heap. The last three rules reveal possible refinements when applying partial functions such as add1, which fails when given non-numeric inputs. This mechanism, when combined with the SMT-aided proof system given below, is sufficient to provide the precision necessary to prove the absence of contract errors.

3.4 SMT-aided proof system

Contract checking and primitive operations rely on a proof system to statically relate values and contracts. We write $\sigma \vdash v : v_c \checkmark$ to mean value v satisfies contract v_c, where all addresses in v are defined in σ. In other words, under any possible instantiation of the unknown values in v, it would satisfy v_c when checked according to the semantics. On the other hand, $\sigma \vdash v : v_c \times$ indicates that v definitely fails v_c. Finally, $\sigma \vdash v : v_c$ **?** is a conservative answer when information from the heap and refinement set is insufficient to draw a definite conclusion. The effectiveness of our analysis depends on the precision of this provability relation—increasing the number of contracts that can be related statically to values prunes spurious paths and eliminates impossible error cases.

$$
\begin{array}{rcl}
\delta(\sigma, o_?, v) & \ni & (\texttt{true}, \sigma) \qquad \text{if } \sigma \vdash v : o_? \checkmark \\
\delta(\sigma, o_?, v) & \ni & (\texttt{false}, \sigma) \qquad \text{if } \sigma \vdash v : o_? \times \\
\delta(\sigma, o_?, a) & \supseteq & \{(\texttt{true}, \sigma_t), (\texttt{false}, \sigma_f)\} \\
\multicolumn{3}{r}{\text{if } \sigma \vdash a : o_? \textbf{?} \text{ and } (\sigma_t, \_) = \mathit{refine}(\sigma, a, o_?)} \\
\multicolumn{3}{r}{\text{and } (\sigma_f, \_) = \mathit{refine}(\sigma, a, \neg o_?)} \\
\multicolumn{3}{c}{\ldots} \\
\delta(\sigma, \texttt{add1}, n) & \ni & (n+1, \sigma) \\
\delta(\sigma, \texttt{add1}, v) & \ni & (a, \sigma'[a \mapsto \bullet/\texttt{num?}]) \\
\multicolumn{3}{l}{\text{where } \delta(\sigma, \texttt{num?}, v) \ni (\texttt{true}, \sigma'), v \neq n, \text{ and } a \notin \sigma'} \\
\delta(\sigma, \texttt{add1}, v) & \ni & (\texttt{blame}_\Lambda, \sigma') \\
\multicolumn{3}{r}{\text{where } \delta(\sigma, \texttt{num?}, v) \ni (\texttt{false}, \sigma')} \\
\multicolumn{3}{c}{\ldots}
\end{array}
$$

Figure 3. Selected primitive operations

3.4.1 Simple proof system

A simple proof system can be obtained which returns definite answers for concrete values, uses heap refinements, and handles negation of predicates and disjointness of data types.

$$
\begin{array}{ll}
\sigma \vdash n : \texttt{num?} \checkmark & \\
\sigma \vdash n : o_? \times & \text{if } o_? \in \{\texttt{cons?}, \texttt{proc?}, \text{etc.}\} \\
\sigma \vdash u/\overrightarrow{v} : v_i \checkmark & \text{if } v_i \in \overrightarrow{v} \\
\sigma \vdash u/\overrightarrow{v} : o_? \times & \text{if } \neg o_? \in \overrightarrow{v} \\
\sigma \vdash a : v \checkmark & \text{if } \sigma \vdash \sigma(a) : v \checkmark \\
\sigma \vdash a : v \times & \text{if } \sigma \vdash \sigma(a) : v \times \\
\sigma \vdash a : v \textbf{?} & \text{if } \sigma \vdash \sigma(a) : v \textbf{?} \\
\ldots & \\
\sigma \vdash v : v_c \textbf{?} & \text{(conservative default)}
\end{array}
$$

Notice that the proof system only needs to handle a small number of well-understood contracts. We rely on evaluation to naturally break down complex contracts into smaller ones and take care of subtle issues such as divergence and crashing. By the time we have $u/\overrightarrow{v}$, we can assume all contracts in $\overrightarrow{v}$ have terminated with success on u. With these simple and obvious rules, our system can already verify a significant number of interesting programs. With SMT solver integration, as described below, we can handle far more interesting constraints, including relations between numeric values, without requiring an encoding of the full language.

3.4.2 Integrating an SMT solver

We extend the simple provability relation by employing an external solver.

We first define the translation $\{\!|\cdot|\!\}_S$ from heaps and contract-value pairs into formulas in solver S:

$$
\begin{array}{rcl}
\{\!|\overrightarrow{(a, c)}|\!\}_S & = & \bigwedge \overrightarrow{\{\!|a : c|\!\}_S} \\
\{\!|a_1 : \texttt{(>/c } n\texttt{)}|\!\}_S & = & \texttt{ASSERT } a_1 \texttt{ > } n \\
\{\!|a_1 : \texttt{(>/c } a_2\texttt{)}|\!\}_S & = & \texttt{ASSERT } a_1 \texttt{ > } a_2 \\
\{\!|a : \texttt{(=/c (+ } a_1\ a_2\texttt{))}|\!\}_S & = & \texttt{ASSERT } a \texttt{ = } a_1 \texttt{ + } a_2 \\
\multicolumn{3}{c}{\ldots}
\end{array}
$$

The translation of a heap is the conjunction of all formulas generated from translatable refinements. The function is partial, and there are straightforward rules for translating specific pairs of $(a : c)$ where c are drawn from a small set of simple, well-understood contracts. This mechanism is enough for the system to verify many interesting programs because the analysis relies on evaluation to break down complex, higher-order predicates. Not having a translation for some contract c only reduces precision and does not affect soundness.

Next, the extension ($\vdash_S$) is straightforward. The old relation ($\vdash$) is refined by a solver S. Whenever the basic relation proves $\sigma \vdash v : c\textbf{?}$, we call out to the solver to try to either prove or refute the claim:

$$
\frac{\{\!|\sigma|\!\}_S \wedge \neg\{\!|v : c|\!\}_S \text{ is unsat}}{\sigma \vdash_S v : c \checkmark} \qquad \frac{\{\!|\sigma|\!\}_S \wedge \{\!|v : c|\!\}_S \text{ is unsat}}{\sigma \vdash_S v : c \times}
$$

The solver-aided relation uses refinements available on the heap to generate premises $\{\!|\sigma|\!\}_S$. Unsatisfiability of $\{\!|\sigma|\!\}_S \wedge \neg\{\!|v : c|\!\}_S$ is equivalent to validity of $\{\!|\sigma|\!\}_S \Rightarrow \{\!|v : c|\!\}_S$, hence value definitely satisfies contract c. Likewise, unsatisfiability of $\{\!|\sigma|\!\}_S \wedge \{\!|v : c|\!\}_S$ means v definitely refutes c. In any other case, we relate the value-contract pair to the conservative answer.

3.5 Program evaluation

We give a reachable-states semantics to programs: the initial program p is paired with an empty heap, and `eval` produces all states in the reflexive, transitive closure of the single-step reduction relation closed under evaluation contexts.

$$
\begin{array}{l}
\texttt{eval} \ : p \to \mathcal{P}(\varsigma) \\
\texttt{eval}(\overrightarrow{m} e) = \{\varsigma \mid \overrightarrow{m} \vdash (e'; e), \emptyset \longmapsto\!\!\!\!\!\rightarrow \varsigma\} \\
\quad \text{where } e' = \mathit{amb}(\{\texttt{true}, \overrightarrow{\texttt{havoc} f}\}), (\texttt{module } f\ v_c\ v) \in \overrightarrow{m}
\end{array}
$$

Modules with unknown definitions, which we call *opaque*, complicate the definition of `eval`, since they may contain references to concrete modules. If only the main module is considered, an opaque module might misuse a concrete value in ways not visible to the system. We therefore apply `havoc` to each concrete module before evaluating the main expression.

3.6 Soundness

A program with unknown components is an abstraction of a fully-known program. Thus, the semantics of the abstracted program should approximate the semantics of any such concrete version. In particular, any behavior the concrete program exhibits should also be exhibited by the abstract approximation of that program.

However, we must be precise as to which behaviors are relevant. Suppose we have a single concrete module that links against a single opaque module. The semantics of this program should include all of the possible behaviors, both good and bad, of the known module assuming the opaque module always lives up to its contract. We exclude from consideration behaviors that cause the unknown module to be blamed, since it is of course impossible to verify an unknown program. In other words, we try to verify the parts of the program that are known, assuming arbitrary, but correct, behavior for the parts of the program that are unknown.

For this reason, the precise semantic account of blame is crucial. The demonic `havoc` context can introduce blame of both the known and unknown modules; since we can distinguish these parties, it is easy to ignore blame of the unknown context.

In the remainder of this section, we formally define the approximation relation and show that evaluation preserves the approximation, i.e. if program q is an approximation of program p (q is like p but with potentially more unknowns), then the evaluation of q is an approximation of the evaluation of p.

Approximation: We define two approximation relations: between modules and between pairs of expressions and heaps.

We write $\varsigma \sqsubseteq \varsigma'$ to mean "ς' approximates ς," or "ς refines ς'," which intuitively means ς' stands for a set of states including ς. For example, $(1, \{\}) \sqsubseteq (a, \{a \mapsto \bullet\})$.

One complication introduced by addresses is that a *single* address in the abstract program may accidentally approximate *multiple* distinct values in the concrete one. Such accidental approximations are not in general preserved by reduction, as in the following

$$\frac{(u/\overrightarrow{v}, \sigma_1) \sqsubseteq^F (\sigma_2(a), \sigma_2) \quad F(a) = u/\overrightarrow{v}}{(u/\overrightarrow{v}, \sigma_1) \sqsubseteq^F (a, \sigma_2)} \qquad \frac{(\sigma_1(a_1), \sigma_1) \sqsubseteq^F (\sigma_2(a_2), \sigma_2) \quad F(a_2) = a_1}{(a_1, \sigma_1) \sqsubseteq^F (a_2, \sigma_2)}$$

$$\frac{(u_1/\overrightarrow{v_1}, \sigma_1) \sqsubseteq^F (u_2/\overrightarrow{v_2}), \sigma_2) \qquad (v_c, \sigma_1) \sqsubseteq^F (v_d, \sigma_2)}{(u_1/\overrightarrow{v_1} \cup \{v_c\}, \sigma_1) \sqsubseteq^F (u_2/\overrightarrow{v_2} \cup \{v_d\}, \sigma_2)}$$

$$\frac{}{(u, \sigma_1) \sqsubseteq^F (\bullet, \sigma_2)} \qquad \frac{(u_1/\overrightarrow{v_1}, \sigma_1) \sqsubseteq^F (u_2/\overrightarrow{v_2}, \sigma_2)}{(u_1/\overrightarrow{v_1} \cup v, \sigma_1) \sqsubseteq^F (u_2/\overrightarrow{v_2}, \sigma_2)}$$

$$\frac{\texttt{(module } f\ u_c\ \bullet\texttt{)} \in \overrightarrow{m} \qquad or \qquad f \in \{\dagger, \texttt{havoc}\}}{(\texttt{blame}^f_g, \sigma_1) \sqsubseteq^F_{\overrightarrow{m}} (e, \sigma_2)}$$

Figure 4. Selected Approximation Rules

example where $(e_1, \sigma_1) \sqsubseteq (e_2, \sigma_2)$:

$$\begin{array}{lcll} e_1 & = & \texttt{(if false 1 2)} & \sigma_1 = \{\} \\ e_2 & = & \texttt{(if } a\ a\ a\texttt{)} & \sigma_2 = \{a \mapsto \bullet\} \end{array}$$

The abstract program does not continue to approximate the concrete one in their next states:

$$\begin{array}{lcll} e_1 & \longmapsto & (2, \sigma_1') & \sigma_1' = \{\} \\ e_2 & \longmapsto & (a, \sigma_2') & \sigma_2' = \{a \mapsto \texttt{false}\} \end{array}$$

We therefore also define a "strong" version of the approximation relation, $\sqsubseteq^F$, where each address in the abstract program approximates exactly one value in the concrete program, and this consistency is witnessed by some function F from addresses to values. Then $e \sqsubseteq e'$ means that $\exists F. e \sqsubseteq^F e'$ Since no such function exists between e_1 and e_2 above, $e_1 \not\sqsubseteq^F e_2$ for any F, and therefore $e_1 \not\sqsubseteq e_2$.

Figure 4 shows the important cases in the definition of $\sqsubseteq^F$; we omit structurally recursive rules. All pre-values are approximated by $\bullet$, and unknown values with contracts approximate values that satisfy the same contracts. We extend the relation $\sqsubseteq^F_{\overrightarrow{m}}$ structurally to evaluation contexts $\mathcal{E}$, point-wise to sequences, and to sets of program states.

In the following example, $(e_1, \sigma_1) \sqsubseteq^F (e_2, \sigma_2)$, where $F = \{a_0 \mapsto \texttt{false}, a_1 \mapsto 1, a_2 \mapsto 2\}$:

$$\begin{array}{ll} e_1 = \texttt{(if false 1 2)} & \sigma_1 = \{\} \\ e_2 = \texttt{(if } a_1\ a_2\ a_3\texttt{)} & \sigma_2 = \{a_1 \mapsto \bullet, a_2 \mapsto \bullet, a_3 \mapsto \bullet\} \end{array}$$

Notice that F's domain is a superset of the domain of the heap σ_2. In addition, our soundness result does not consider additional errors that blame unknown modules or the `havoc` module, and therefore we parameterize the approximation relation $\sqsubseteq^F_{\overrightarrow{m}}$ with the module definitions $\overrightarrow{m}$ to select the opaque modules. We omit these parameters where they are easily inferred to ease notation.

With the definition of approximation in hand, we are now in a position to state the main soundness theorem for the system.

Theorem 1 (Soundness of Symbolic λ_C)**.**
If $p \sqsubseteq^F_{\overrightarrow{m}} q$ where $q = \overrightarrow{m}e$ and $\varsigma \in eval(p)$, then there exists some $\varsigma' \in eval(q)$ such that $\varsigma \sqsubseteq^{F'}_{\overrightarrow{m}} \varsigma'$.

We defer all proofs to the technical report for space.

3.7 Verification and the blame theorem

We can now define verification as a simple corollary of soundness. First we defined when a module is *verified* by our approach.

Definition 1 (Verified module)**.**
A module `(module` $f\ u_c\ u$`)` $\in p$ *is* verified in p *if* $u \neq \bullet$ *and* $eval(p) \not\ni \texttt{blame}^f$.

Now, by soundness, f is always safe.

Theorem 2 (Verified modules can't be blamed)**.**
If a module named f is verified in p, then for any concrete program q for which p is an abstraction, $eval(q) \not\ni \texttt{blame}^f$.

3.8 Taming the infinite state space

A naive implementation of the above semantics will diverge for many programs. Consider the following example:

```
(define (fact n)
  (if (= n 0) 1 (* n (fact (- n 1)))))
(fact •)
```

Ignoring error cases, it eventually reduces non-deterministically to all of the following:

$$\begin{array}{r} \texttt{1} \text{ if } a_n \mapsto \texttt{0} \\ \texttt{(* } a_n\ \texttt{1)} \text{ if } a_n \not\mapsto \texttt{0}, a_{n-1} \mapsto \texttt{0} \\ \texttt{(* } a_n\ \texttt{(* } a_{n-1}\ \texttt{(fact } a_{n-1}\texttt{)))} \text{ if } a_n, a_{n-1} \not\mapsto \texttt{0} \end{array}$$

where a_{n-1} is a fresh address resulting from subtracting a_n by one. The process continues with a_{n-2}, a_{n-3}, etc. This behavior from the analysis happens because it attempts to approximate *all* possible concrete substitutions to abstract values. Although `fact` terminates for all concrete naturals, there are an infinite number of those: a_n can be `0`, `1`, `2`, and so on.

To enforce termination for all programs, we can resort to well-known techniques such as finite state or pushdown abstractions (Van Horn and Might 2012). But often those are overkill at the cost of precision. Consider the following program:

```
(let* ([id (λ (x) x)] [y (id 0)] [z (id 1)])
  (< y z))
```

where a monovariant flow analysis such as 0CFA (Shivers 1988) thinks `y` and `z` can be both `0` and `1`, and pushdown analysis thinks `y` is 0 and `z` is either `0` or `1`. For a concrete, straight-line program, such imprecision seems unsatisfactory. We therefore aim for an analysis that provides exact execution for non-recursive programs and retains enough invariants to verify interesting properties of recursive ones. The analysis quickly terminates for a majority of programming patterns with decent precision, although it is not guaranteed to terminate in the general case—see section 4 for empirical results.

One technical difficulty is that the semantics of contracts prevents us from using a recursive function's contract directly as a loop invariant, because contracts are only boundary-level enforcement. It is unsound to assume returned values of internal calls can be approximated by contracts, as in `f` below:

```
(f : nat? → nat?)
(define (f n) (if (= n 0) "" (str-len (f (- n 1)))))
```

If we assume the expression `(f (- n 1))` returns a number as specified in the contract, we will conclude `f` never returns, and is blamed either for violating its own contract by returning a string, or for applying `str-len` to a number. However, `f` returns `0` when applied to `1`. To soundly and precisely approximate this semantics in the absence of types, we recover data type invariants by execution.

Summarizing function results: To accelerate convergence, we modify the application rules as follows. At each application, we decide whether execution should step to the function's body or wait for known results from other branches. When an application `(f v)` reduces to a similar application, we plug in known results instead

$$\frac{\mathcal{E} \neq \mathcal{E}_1[(\mathtt{rt}_{\langle \sigma_0, \lambda x.e, v_0\rangle}\ \mathcal{E}_k)] \text{ for any } \mathcal{E}_1, \mathcal{E}_k, \sigma_0, v_0}{\langle \Xi, M, \mathcal{E}[((\lambda x.e)\ v)], \sigma\rangle \longmapsto \langle \Xi, M, \mathcal{E}[(\mathtt{rt}_{\langle \sigma, \lambda x.e, v\rangle}\ [v/x]e)], \sigma\rangle} \qquad \frac{\overrightarrow{\langle a_o, a_n\rangle} = F \qquad \sigma' = \sigma[\overrightarrow{a_n \mapsto \sigma_0(a_0)\ \oplus\ \sigma(a_0)}]}{\langle \Xi, M, \mathcal{E}[(\mathtt{blur}_{\langle F, \sigma_0, v_0\rangle}\ v)], \sigma\rangle \longmapsto \langle \Xi, M, \mathcal{E}[v_0\ \oplus\ v], \sigma'\rangle}$$

$$\frac{\mathcal{E} = \mathcal{E}_1[(\mathtt{rt}_{\langle \sigma_0, \lambda x.e, v_0\rangle}\ \mathcal{E}_k)] \text{ for some } \mathcal{E}_1, \mathcal{E}_k, \sigma_0, v_0 \qquad \langle \sigma, v\rangle \not\sqsubseteq \langle \sigma_0, v_0\rangle \qquad v_1 = v_0\ \oplus\ v}{\langle \Xi, M, \mathcal{E}[((\lambda x.e)\ v)], \sigma\rangle \longmapsto \langle \Xi, M, \mathcal{E}[(\mathtt{rt}_{\langle \sigma, \lambda x.e, v_1\rangle}\ [v_1/x]e)], \sigma\rangle}$$

$$\frac{\begin{array}{c}\mathcal{E} = \mathcal{E}_1[(\mathtt{rt}_{\langle \sigma_0, v_f, v_0\rangle}\ \mathcal{E}_k)] \text{ for some } \mathcal{E}_1, \mathcal{E}_k, \sigma_0, v_0 \qquad \langle \sigma, v\rangle \sqsubseteq^F \langle \sigma_0, v_0\rangle \\ \Xi' = \Xi \sqcup [\langle \sigma_0, v_f, v_0\rangle \mapsto \langle F, \sigma, \mathcal{E}_1, \mathcal{E}_k\rangle] \qquad \langle v_a, \sigma_a\rangle \in M[\langle \sigma_0, v_f, v_0\rangle] \qquad \sigma' = \sigma[\overrightarrow{a_n \mapsto \sigma_a[a_o]}] \text{ where } \overrightarrow{\langle a_o, a_n\rangle} = F\end{array}}{\langle \Xi, M, \mathcal{E}[(v_f\ v)], \sigma\rangle \longmapsto \langle \Xi', M, \mathcal{E}_1[(\mathtt{rt}_{\langle \sigma_0, v_f, v_0\rangle}\ (\mathtt{blur}_{\langle F, \sigma_a, v_a\rangle}\ \mathcal{E}_k[v_a]))], \sigma'\rangle}$$

$$\frac{M' = M \sqcup [\langle \sigma_0, v_f, v_0\rangle \mapsto \langle v, \sigma\rangle]}{\langle \Xi, M, \mathcal{E}[(\mathtt{rt}_{\langle \sigma_0, v_f, v_0\rangle}\ v)], \sigma\rangle \longmapsto \langle \Xi, M', \mathcal{E}[v], \sigma\rangle}$$

$$\frac{M' = M \sqcup [\langle \sigma_0, v_f, v_0\rangle \mapsto \langle v, \sigma\rangle] \qquad \langle F, \sigma_k, \mathcal{E}_1, \mathcal{E}_k\rangle \in \Xi[\langle \sigma_0, v_f, v_0\rangle] \qquad \sigma'_k = \sigma_k[\overrightarrow{a_n \mapsto \sigma(a_o)}] \text{ where } \overrightarrow{\langle a_o, a_n\rangle} = F}{\langle \Xi, M, \mathcal{E}[(\mathtt{rt}_{\langle \sigma_0, v_f, v_0\rangle}\ v)], \sigma\rangle \longmapsto \langle \Xi, M', \mathcal{E}_1[(\mathtt{rt}_{\langle \sigma_0, v_f, v_0\rangle}\ (\mathtt{blur}_{\langle F, \sigma, v\rangle}\ \mathcal{E}_k[v]))], \sigma'_k\rangle}$$

Figure 5. Summarizing Semantics

Expressions	e	$+= (\mathtt{rt}_{\langle \sigma, v, v\rangle}\ e) \mid (\mathtt{blur}_{\langle F, \sigma, v\rangle}\ e)$
Values	v	$+= \mu x.\overrightarrow{v} \mid !x$
Evaluation contexts	$\mathcal{E}$	$+= (\mathtt{rt}_{\langle \sigma, v, v\rangle}\ \mathcal{E}) \mid (\mathtt{blur}_{\langle F, \sigma, v\rangle}\ \mathcal{E})$
Context memo tables	Ξ	$::= \overrightarrow{((\sigma, v, v), \overrightarrow{(F, \sigma, \mathcal{E}, \mathcal{E})})}$
Value memo tables	M	$::= \overrightarrow{((\sigma, v, v), \overrightarrow{(v, \sigma)})}$
Renamings	F	$::= \overrightarrow{\langle a, a\rangle}$

Figure 6. Syntax extensions for approximation

of executing `f`'s body again, avoiding the infinite loop. Correspondingly, when `(f v)` returns, we plug the new-found answer into contexts that need the result of `(f v)`. The execution continues until it has a set soundly describing the results of `(f v)`.

To track information about application results and waiting contexts, we augment the execution with two global tables M and Ξ as shown in figure 6. We borrow the choice of metavariable names from work on concrete summaries (Johnson and Van Horn 2014).

A value memo table M maps each application to known results and accompanying refinements. Intuitively, if $M(\sigma, v_f, v_x) \ni (v, \sigma')$ then in some execution branch, there is an application $(v_f\ v_x), \sigma \longmapsto\!\!\!\!\rightarrow (v, \sigma')$.

A context memo table Ξ maps each application to contexts waiting for its result. Intuitively, $\Xi(\sigma, v_f, v_x) \ni (F, \sigma', \mathcal{E}_1, \mathcal{E}_k)$ means during evaluation, some expression $\mathcal{E}_1[(\mathtt{rt}_{\langle \sigma, v_f, v_x\rangle}\ [\mathcal{E}_k[(v_f\ v_z)]])]$ with heap σ' is paused because applying $(v_f\ v_z)$ under assumptions in σ' is subsumed by applying $(v_f\ v_x)$ under assumptions in σ up to consistent address renaming specified by function F.

To keep track of function applications seen so far, we extend the language with the expression $(\mathtt{rt}_{\langle \sigma, v, v'\rangle}\ e)$, which marks e as being evaluated as the result of applying v to v', but otherwise behaves like e. The expression $(\mathtt{blur}_{\langle F, \sigma, v\rangle}\ e)$, whose detailed role is discussed below, approximates e under guidance from a "previous" value v.

Finally, we add recursive contracts $\mu x.\overrightarrow{v}$ and recursive references $!x$ for approximating inductive sets of values. For example, $\mu x.\{\mathtt{empty}, \langle \bullet/\mathtt{nat?}, !x\rangle\}$ approximates all finite lists of naturals.

A state in the approximating semantics with summarization consists of global tables Ξ, M, and a set S of explored states $\overrightarrow{\varsigma}$.

Reduction now relates tables Ξ, M, and a set of states $\overrightarrow{\varsigma}$ to new tables Ξ', M' and a new set of states $\overrightarrow{\varsigma}'$. We define a relation $\langle \Xi, M, \varsigma\rangle \longrightarrow \langle \Xi, M, \varsigma\rangle$, and then lift this relation point-wise to sets of states. Figure 5 only shows rules that use the global tables or new expression forms.

In the first rule, if an application $((\lambda x.e)\ v)$ is not previously seen, execution proceeds as usual, evaluating expression e with x bound to v, but marking this expression using `rt`.

Second, if a previous application of $((\lambda x.e)\ v_0)$ results in application of the same function to a new argument v, we approximate the new argument before continuing. Taking advantage of knowledge of the previous argument, we guess the transition from the v_0 to v and heuristically emulate an arbitrary amount of such transformation using the $\oplus$ operator. For example, if v_0 is `empty` and v is $\langle$`1`,`empty`$\rangle$, we approximate the latter to $\mu x.\{$`empty`, $\langle$`1`,$!x\rangle\}$, denoting a list of `1`'s. If a different number is later prepended to the list, it is approximated to a list of numbers. The $\oplus$ operator should work well in common cases and not hinder convergence in the general case. Failure to give a good approximation to a value results in non-termination but does not affect soundness.

Third, when an application results in a similar one with potentially refined arguments, we avoid stepping into the function body and use known results from table M instead. In addition, we refine the current heap to make better use of assumptions about the particular "base case". We also remember the current context as one waiting for the result of such application. To speed up convergence, apart from feeding a new answer v_a to the context, we wrap the entire expression inside $(\mathtt{blur}_{\langle F, \sigma, v\rangle}\ [\ ])$ to approximate the future result.

The fourth rule in figure 5 shows reduction for returning from an application. Apart from the current context, the value is also returned to any known context waiting on the same application. Besides, the value is also remembered in table M. The resumption and refinement are analogous to the previous rule.

Finally, expression $(\mathtt{blur}_{\langle F, \sigma, v_0\rangle}\ v)$ approximates value v under guidance from the previous value v_0 and also approximates values on the heap from observation of the previous case. Overall, the approximating operator $\oplus$ occurs in three places: arguments of recursive applications, result of recursive applications, and abstract

values on the heap when recursive applications return. Empirical results for our tool are presented in section 4.

Soundness of summarization: A system (Ξ, M, S) approximates a state ς if that state can be recovered from the system through approximation rules. The crucial rule, given below, states that if the system (Ξ, M, S) already approximates expression e and the application $(v_f\ v_x)$ is known to reduce to e, then (Ξ, M, S) is an approximation of $\mathcal{E}_k[e]$ where $\mathcal{E}_k$ is a waiting context for this application.

$$\frac{\begin{array}{c}(\mathtt{rt}_{\langle \_,v,\_\rangle}\ [\,]) \notin \mathcal{E}_0 \\ (v_x,\sigma) \sqsubseteq (v_z,\sigma') \qquad (v_y,\sigma) \sqsubseteq (v_z,\sigma') \\ \Xi(\sigma',v,v_z) \ni (F,\sigma',\mathcal{E}_0',\mathcal{E}_k') \qquad (\mathcal{E}_0,\sigma) \sqsubseteq (\mathcal{E}_0',\sigma') \\ (\mathcal{E}_k,\sigma) \sqsubseteq (\mathcal{E}_k',\sigma') \qquad (\mathcal{E}_0[(\mathtt{rt}_{\langle \sigma_1,v,v_y\rangle}\ e)],\sigma) \sqsubseteq (\Xi,M,S)\end{array}}{(\mathcal{E}_0[(\mathtt{rt}_{\langle \sigma_0,v,v_x\rangle}\ \mathcal{E}_k[(\mathtt{rt}_{\langle \sigma_1,v,v_y\rangle}\ e)])],\sigma) \sqsubseteq (\Xi,M,S)}$$

As a consequence, summarization properly handles repetition of waiting contexts, and gives results that approximate any number of recursive applications. We refer readers to the appendix of the accompanying technical report for the full definition of the approximation relation.

With this definition in hand, we can state the central lemma to establish the soundness of the revised semantics that uses summarization.

Lemma 1 (Soundness of summarization)**.**
If $\varsigma \sqsubseteq (\Xi, M, S)$ *and* $\varsigma \longmapsto \varsigma'$*, then* $(\Xi, M, S) \longmapsto\!\!\!\!\!\rightarrow (\Xi', M', S')$ *such that* $\varsigma' \sqsubseteq (\Xi', M', S')$.

The proof is given in the accompanying technical report. With this lemma in place, it is straightforward to replay the proof of the soundness and blame theorems.

4. Implementation and evaluation

To validate our approach, we implemented a static contract checking tool, SCV, based on the semantics presented in section 3, along with a number of implementation extensions for increased precision and performance. We then applied SCV to a wide selection of programs drawn from the literature on verification of higher-order programs, and report on the results.

The source code for SCV and all benchmarks are available along with instructions on reproducing the results we report here:

```
github.com/philnguyen/soft-contract
```

In order to quantify the importance of the techniques presented in this paper, we also created a simpler tool which omits the key contributions of this work. This slimmed down system, which we refer to as "Simple" below, (a) does not call out to a solver, but relies on remembering seen contracts, (b) never refines the contracts associated with a heap address, but splits disjunctive contracts and unrolls recursive contracts, and (c) does not use our technique for summarizing repeated context. To enable a full comparison on all benchmarks, the Simple tool supports first-class contracts. This simpler system is extremely similar to that presented by our earlier work (Tobin-Hochstadt and Van Horn 2012), but works on all of our benchmarks.

Implementation extensions: SCV supports an extended language beyond that presented in section 3 in order to handle realistic programs. First, more base values and primitive operations are supported, such as strings and symbols (and their operations), although we do not yet use a solver to reason about values other than integers. Second, data structure definitions are allowed at the top-level. Each new data definition induces a corresponding (automatic) extension to the definition of `havoc` to deal with the new class of data. Third, modules have multiple named exports, to handle the examples presented in section 2, and can include local, non-exported, definitions. Fourth, functions can accept multiple arguments and can be defined to have variable-arity, as with `+`, which accepts arbitrarily many arguments. This introduces new possibilities of errors from arity mismatches. Fifth, a much more expressive contract language is implemented with `and/c`, `or/c`, `struct/c`, μ`/c` for conjunctive, disjunctive, data type, and recursive contracts, respectively. Sixth, we provide solver back-ends for both CVC4 (Barrett et al. 2011) and Z3 (De Moura and Bjørner 2008).

Evaluating on existing benchmarks: To evaluate the applicability of SCV to a wide variety of challenging higher-order contract checking problems, we collect examples from the following sources: programs that make use of control-flow-based typing from work on **occurrence typing** (Tobin-Hochstadt and Felleisen 2010), programs from work on **soft typing**, which uses flow analysis to check the preconditions of operations (Cartwright and Fagan 1991), programs with sophisticated specifications from work on model checking **higher-order recursion schemes** (Kobayashi et al. 2011), programs from work on inference of **dependent refinement types** (Terauchi 2010), and programs with rich contracts from our prior work on **higher-order symbolic execution** (Tobin-Hochstadt and Van Horn 2012). We also evaluate SCV on three interactive student video games built for a first-year programming course: **Snake**, **Tetris**, and **Zombie**. These programs were all originally written as sample solutions, following the style expected of students in the course. Of these, Zombie is the most interesting: it was originally an object-oriented program, translated using the encoding seen in section 2.5.

We present our results in summary form in table 1, grouping each of the above sets of benchmark programs; expanded forms of the tables are provided in the accompanying technical report. The table shows total line count (excluding blank lines and comments) and the number of static occurrences of contracts and primitives requiring dynamic checks such as function applications and primitive operations. These checks can be eliminated if we can show that they never fail; this has proven to produce significant speedups in practice, even without eliminating more expensive contract checks (Tobin-Hochstadt et al. 2011).

The table reports time (in milliseconds) and the number of false positives for SCV and our reduced system omitting the key contributions of this work (labeled "Simple"); "∞" indicates a timeout after 5 minutes.

A false positive is a contract violation reported by the analysis, but by human inspection, cannot happen. The programs we consider are all known not to have contract errors, and thus all potential errors are false positives.

In cases where a tool times out, we give an upper bound on the number of false positive error reports. For example, the Simple system times out on two of the higher-order recursion scheme programs, meaning that if it were to complete, it would report *at most* 94 false positives, counting all contract checks from the two programs on which it times out, and the measured false positives on the programs where it completes.

Execution times are measured on a Core i7 2.7GHz laptop with 8GB of RAM.

Discussion: First, SCV works on a benchmarks for a range of previous static analyzers, from type systems to model checking to program analysis.

Second, most programs are analyzed in a reasonable amount of time; the longest remaining analysis time is under 30 seconds. This demonstrates that although the termination acceleration method of section 3.8 is not fully general, it is effective for many programming patterns. For example, SCV terminates with good precision on `last`

Corpus	Lines	Checks	Simple Time (ms)	Simple False Pos.	SCV Time (ms)	SCV False Pos.
Occurrence Typing	115	142	155.8	15	8.9	0
Soft Typing	134	177	424.5	9	380.3	0
Higher-order Recursion Schemes	301	467	∞	≤ 94	3,253.8	4
Dependent Refinement Types	69	116	∞	≤ 66	193.0	1
Higher-order Symbolic Execution	236	319	∞	≤ 19	4,372.7	1
Student Video Games						
Snake	202	270	9,452.5	0	3.008.8	0
Tetris	308	351	∞	-	27,408.5	0
Zombie	249	393	∞	-	11,335.9	0

Table 1. Summary benchmark results. (See the accompanying technical report for detailed results.)

from Wright and Cartwright (1997), which hides recursion behind the Y combinator.

Third, across all benchmarks, over 99% (2329/2335) of the contract checks are statically verified, enabling the elimination both of small checks for primitive operations and expensive contracts; see below for timing results. This result emphasizes the value of static contract checking: gaining confidence about correctness from expensive contracts without actually incurring their cost.

Overall, our experiments show that our approach is able to discover and use invariants implied by conditional flows of control and contract checks. Obfuscations such as multiple layers of abstractions or complex chains of aliases do not impact precision (a common shortcoming of flow analysis).

Our approach does not yet give a way to prove deep structural properties expressed as dependent contracts such as "map over a list preserves the length" or "all elements in the result of filter satisfy the predicate", resulting in the false positives seen in table 1. However, it can already be used to verify many interesting programs because often safety questions depend only on knowledge of top-level constructors. Examples of these patterns appear in programs from Kobayashi et al. (2011) for programs such as `reverse` (see also §2.4), `nil`, and `mem`.

Finally, soft contract verification is more broadly applicable than the systems from which our benchmarks are drawn, which typically are successful only on their own benchmarks. For example, type systems such as occurrence typing (Tobin-Hochstadt and Felleisen 2010) cannot verify any non-trivial contracts, and most soft typing systems do not consider contracts at all. Systems based on higher-order model-checking (Kobayashi et al. 2011), and dependent refinement types (Terauchi 2010) assume a typed language; encoding our programs using large disjoint unions produces unverifiable results.

This broad applicability is why we are not able to directly compare SCV to these other systems across all benchmarks. Instead, the Simple system serves as a benchmark for a system which does not contain our primary contributions.

Contract optimization: We also report speedup results for the three most complex programs in our evaluation, which are interactive games designed for first-year programming courses (Snake, Tetris, and Zombie). For each, we recorded a trace of input and timer events while playing the game, and then used that trace to rerun the game (omitting all graphical rendering) both with the contracts that we verified, and with the contracts manually removed. Each game was run 100 times in both modes; the total time is presented below.

Program	Contracts On (ms)	Contracts Off (ms)
snake	475,799	59
tetris	1,127,591	186
zombie	12,413	1,721

The timing results are quite striking—speedup ranges from over 5x to over 5000x. This does not indicate, of course, that speedups of these magnitudes are achievable for real programs. Instead, it shows that programmers avoid the rich contracts we are able to verify, because of their unacceptable performance overhead. Soft contract verification therefore enables programmers to write these specifications without the run-time cost.

The difference in timing between Zombie and the other two games is intriguing because Zombie uses higher-order dependent contracts extensively, along the lines of `vec/c` from section 2.5, which intuitively should be more expensive. An investigation reveals that most of the cost comes from monitoring flat contracts, especially those that apply to data structures. For example, in Snake, disabling `posn/c`, a simple contract that checks for a `posn` struct with two numeric fields, cuts the run-time by a factor of 4. This contract is repeatedly applied to every such object in the game. In contrast, higher-order contracts, as in the object encodings used in Zombie, delay contracts and avoid this repeated checking.

5. Related work

In this section, we relate our work to four related strands of research: soft-typing, static contract verification, refinement types, and model checking of recursion schemes.

Soft typing: Verifying the preconditions of primitive operations can be seen as a weak form of contract verification and soft typing is a well studied approach to this kind of verification (Cartwright and Felleisen 1996). There are two predominant approaches to soft-typing: one is based on a generalization of Hindley-Milner type inference (Cartwright and Fagan 1991; Wright and Cartwright 1997; Aiken et al. 1994), which views an untyped program as being embedded in a typed one and attempts to safely eliminate coercions (Henglein 1994). The other is founded on set-based abstract interpretation of programs (Flanagan et al. 1996; Flanagan and Felleisen 1999). Both approaches have proved effective for statically checking preconditions of primitive operations, but the approach does not scale to checking pre- and post-conditions of arbitrary contracts. For example, Soft Scheme (Cartwright and Fagan 1991) is not path-sensitive and does not reason about arithmetic, thus it is unable to verify many of the occurrence-typing or higher-order recursion scheme examples considered in the evaluation.

Contract verification: Following in the set-based analysis tradition of soft-typing, there has been work extending set-based analysis to languages with contracts (Meunier et al. 2006). This work shares the overarching goal of this paper: to develop a static contract checking approach for components written in untyped languages with contracts. However the work fails to capture the control-flow-based type reasoning essential to analyzing untyped programs and is unsound (as discussed by Tobin-Hochstadt and Van Horn (2012)).

Moreover, the set-based formulation is complex and difficult to extend to features considered here.

Our prior work (Tobin-Hochstadt and Van Horn 2012), as discussed in the introduction, also performs soft contract verification, but with far less sophistication and success. As our empirical results show, the contributions of this paper are required to tackle the arithmetic relations, flow-sensitive reasoning, and complex recursion found in our benchmarks.

An alternative approach has been applied to checking contracts in Haskell and OCaml (Xu 2012; Xu et al. 2009), which is to inline monitors into a program following a transformation by Findler and Felleisen (2002) and then simplify the program, either using the compiler, or a specialized symbolic engine equipped with an SMT solver. The approach would be applicable to untyped languages except for the final step dubbed *logicization*, a type-based transformation of program expressions into first-order logic (FOL). A related approach used for Haskell is to use a denotational semantics that can be mapped into FOL, which is then model checked (Vytiniotis et al. 2013), but this approach is highly dependent on the type structure of a program. Further, these approaches assume a different semantics for contract checking that monitors recursive calls. This allows the use of contracts as inductive hypotheses in recursive calls. In contrast, our approach can naturally take advantage of this stricter semantics of contract checking and type systems, but can also accommodate the more common and flexible checking policy. Additionally, our approach does not rely on type information, the lack of which makes these approaches inapplicable to many of our benchmarks.

Contract verification in the setting of typed, first-order contracts is much more mature. A prominent example is the work on verifying C# contracts as part of the Code Contracts project (Fähndrich and Logozzo 2011).

Refinement type checking: Refinement types are an alternative approach to statically verifying pre- and post-conditions in a higher-order functional language. There are several approaches to checking type refinements; one is to restrict the computational power of refinements so that checking is decidable at type-checking time (Freeman and Pfenning 1991); another is allow unrestricted refinements as in contracts, but to use a solver to attempt to discharge refinements (Knowles and Flanagan 2010; Rondon et al. 2008; Vazou et al. 2013). In the latter approach, when a refinement cannot be discharged, some systems opt to reject the program (Rondon et al. 2008; Vazou et al. 2013), while others such as hybrid type-checking residualize a run-time check to enforce the refinement (Knowles and Flanagan 2010), similar to the way soft-typing residualizes primitive pre-condition checks. The end result of our approach most closely resembles that of hybrid checking, although the technique applies regardless of the type discipline and approaches the problem using different tools.

DJS (Chugh et al. 2012b,a) supports expressive refinement specification and verification for stateful JavaScript programs, including sophisticated dependent specifications which SCV cannot verify. However, most dependent properties require heavy annotations. Moreover, `null` inhabits every object type. Thus the approach cannot give the same guarantees about programs such as `reverse` (§2.4) without significantly more annotation burden. Additionally, it relies on whole program annotation, type-checking, and analysis.

Model checking higher-order recursion schemes: Much of the recent work on model checking of higher-order programs relies on the decidability of model checking trees generated by higher-order recursion schemes (HORS) (Ong 2006). A HORS is essentially a program in the simply-typed λ-calculus with recursion and finitely inhabited base types that generates (potentially infinite) trees. Program verification is accomplished by compiling a program to a HORS in which the generated tree represents program event sequences (Kobayashi 2009b; Kobayashi et al. 2010). This method is sound and complete for the simply typed λ-calculus with recursion and finite base types, but the gap between this language and realistic languages is significant. Subsequently, an untyped variant of HORS has been developed (Tsukada and Kobayashi 2010), which has applications to languages with more advanced type systems, but despite the name it does not lead to a model checking procedure for the untyped λ-calculus. A subclass of untyped HORS is the class of recursively typed recursion schemes, which has applications to typed object-oriented programs (Kobayashi and Igarashi 2013). In this setting, model checking is undecidable, but relatively complete with a certain recursive intersection type system (anything typable in this system can be verified). To cope with infinite data domains such as integers, counter-example guided abstraction refinement (CEGAR) techniques have been developed (Kobayashi et al. 2011). The complexity of model checking even for the simply typed case is n-EXPTIME hard (where n is the rank of the recursion scheme), but progress on decision procedures (Kobayashi and Ong 2009; Kobayashi 2009a) has lead to verification engines that can verify a number of "small but tricky higher-order functional programs in less than a second."

In comparison, the HORS approach can verify some specifications which SCV cannot, but in a simpler (typed) setting, whereas our lightweight method applies to richer languages. Our approach handles untyped higher-order programs with sophisticated language features and infinite data domains. Higher-order program invariants may be stated as behavioral contracts, while the HORS-based systems only support assertions on first order data. Our work is also able to verify programs with unknown external functions, not just unknown integer values, which is important for modular program verification, and we are able to verify many of the small but tricky programs considered in the HORS work.

6. Conclusions and perspective

We have presented a lightweight method and prototype implemementation for static contract checking using a non-standard reduction semantics that is capable of verifying higher-order modular programs with arbitrarily omitted components. Our tool, SCV, scales to realistic language features such as recursive data structures and modular programs, and verifies programs written in the idiomatic style of dynamic languages. The analysis proves the absence of run-time errors without excessive reliance on programmer help. With zero annotation, SCV already helps programmers find unjustified usage of partial functions with high precision and could even be modified to suggest inputs that break the program. With explicit contracts, programmers can enforce rich specifications to their programs and have those optimized away without incurring the significant run-time overhead entailed by dynamic enforcement.

While in this paper, we have addressed the problem of soft contract verification, the technical tools we have introduced apply beyond this application. For example, a run of SCV can be seen as a modular program analysis—it soundly predicts which functions are called at any call site. Moreover it can be composed with whole-program analysis techniques to derive modular analyses (Van Horn and Might 2010). A small modification to blur to cause it to pick a small set of concrete values would turn our system into a concolic execution engine (Larson and Austin 2003). Adding temporal contracts (Disney et al. 2011) to our system would produce a model checker for higher-order languages. This breadth of application follows directly from the semantics-based nature of our approach.

Acknowledgments

We thank Carl Friedrich Bolz, Jeffrey S. Foster, Michael Hicks, J. Ian Johnson, Lindsey Kuper, Aseem Rastogi, and Matthew Wilson for comments. We thank the anonymous reviewers of ICFP 2014 for their detailed reviews, which helped to improve the presentation and technical content of the paper. This material is based on research sponsored by the NSF under award 1218390, the NSA under the Science of Security program, and DARPA under the programs Automated Program Analysis for Cybersecurity (FA8750-12-2-0106). The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon.

References

A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. POPL, 1994.

T. H. Austin, T. Disney, and C. Flanagan. Virtual values for language extension. OOPSLA, 2011.

C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. CAV. 2011.

R. Cartwright and M. Fagan. Soft typing. PLDI, 1991.

R. Cartwright and M. Felleisen. Program verification through soft typing. *ACM Comput. Surv.*, 1996.

R. Chugh, D. Herman, and R. Jhala. Dependent types for JavaScript. In *OOPSLA*, 2012a.

R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements: A logic for duck typing. In *POPL*, 2012b.

L. De Moura and N. Bjørner. Z3: an efficient SMT solver. TACAS, 2008.

C. Dimoulas, R. B. Findler, C. Flanagan, and M. Felleisen. Correct blame for contracts: no more scapegoating. POPL, 2011.

T. Disney. contracts.coffee, July 2013. URL `http://disnetdev.com/contracts.coffee/`.

T. Disney, C. Flanagan, and J. McCarthy. Temporal higher-order contracts. ICFP, 2011.

M. Fähndrich and F. Logozzo. Static contract checking with abstract interpretation. FoVeOOS, 2011.

R. B. Findler and M. Felleisen. Contracts for higher-order functions. ICFP, 2002.

C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Program. Lang. Syst.*, 1999.

C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. PLDI, 1996.

T. Freeman and F. Pfenning. Refinement types for ML. PLDI, 1991.

F. Henglein. Dynamic typing: syntax and proof theory. *Science of Computer Programming*, 1994.

R. Hickey, M. Fogus, and contributors. core.contracts, July 2013. URL `https://github.com/clojure/core.contracts`.

J. I. Johnson and D. Van Horn. Abstracting abstract control. *CoRR*, 2014. URL `http://arxiv.org/abs/1305.3163`.

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

N. Kobayashi. Model-checking higher-order functions. PPDP, 2009a.

N. Kobayashi. Types and higher-order recursion schemes for verification of higher-order programs. POPL, 2009b.

N. Kobayashi and A. Igarashi. Model-Checking Higher-Order programs with recursive types. ESOP, 2013.

N. Kobayashi and C. H. L. Ong. A type system equivalent to the modal Mu-Calculus model checking of Higher-Order recursion schemes. LICS, 2009.

N. Kobayashi, N. Tabuchi, and H. Unno. Higher-order multi-parameter tree transducers and recursion schemes for program verification. POPL, 2010.

N. Kobayashi, R. Sato, and H. Unno. Predicate abstraction and CEGAR for higher-order model checking. PLDI, 2011.

E. Larson and T. Austin. High coverage detection of input-related security faults. USENIX Security, 2003.

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

B. Meyer. *Eiffel : The Language*. 1991.

P. C. Nguyễn, S. Tobin-Hochstadt, and D. Van Horn. Soft contract verification. *CoRR*, 2014. URL `http://arxiv.org/abs/1307.6239`.

C. H. L. Ong. On Model-Checking trees generated by Higher-Order recursion schemes. LICS, 2006.

R. Plosch. Design by contract for Python. 1997. APSEC/ICSC'97.

P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. PLDI, 2008.

O. Shivers. Control flow analysis in Scheme. PLDI, 1988.

T. S. Strickland, S. Tobin-Hochstadt, R. B. Findler, and M. Flatt. Chaperones and impersonators: run-time support for reasonable interposition. OOPSLA, 2012.

T. Terauchi. Dependent types from counterexamples. POPL, 2010.

S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. ICFP, 2010.

S. Tobin-Hochstadt and D. Van Horn. Higher-order symbolic execution via contracts. OOPSLA, 2012.

S. Tobin-Hochstadt, V. St-Amour, R. Culpepper, M. Flatt, and M. Felleisen. Languages as libraries. PLDI, 2011.

T. Tsukada and N. Kobayashi. Untyped recursion schemes and infinite intersection types. FoSSaCS, 2010.

D. Van Horn and M. Might. Abstracting abstract machines. ICFP, 2010.

D. Van Horn and M. Might. Systematic abstraction of abstract machines. *Journal of Functional Programming*, 2012.

N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. ESOP, 2013.

N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. ICFP, 2014.

D. Vytiniotis, S. Peyton Jones, K. Claessen, and D. Rosén. HALO: Haskell to logic through denotational semantics. POPL, 2013.

A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Trans. Program. Lang. Syst.*, 1997.

D. N. Xu. Hybrid contract checking via symbolic simplification. PEPM, 2012.

D. N. Xu, S. Peyton Jones, and S. Claessen. Static contract checking for Haskell. POPL, 2009.

H. Zhu and S. Jagannathan. Compositional and lightweight dependent type inference for ML. 2013.