Pruning Contracts with Rosette

Ben Greenman
Northeastern University
benjaminlgreenman@gmail.com

Abstract
Contracts are a pragmatic tool for managing software systems, but programs using contracts suffer runtime overhead. If this overhead becomes a performance bottleneck, programmers must manually edit or remove their contracts. This is no good. Rather, the contracts should identify their own inefficiencies and remove unnecessary dynamic checks. Implementing contracts with Rosette is a promising way to build such self-aware contracts.

1. The Trouble with Contracts
Racket’s higher-order contracts [3] are an expressive and easy-to-use specification language. Any programmer can design a contract, as the language for building contracts is essentially Racket. For example, the following program defines a function even++ that accepts only even numbers and returns only odd numbers. The contract combinator -> specifies the behavior of even++ using the predicates even? and odd?.

```
#lang racket/base
(require racket/contract)

(define (even? x)
  (zero? (modulo x 2)))

(define (odd? x)
  (even? (+ x 1)))

(define/contract (even++ n)
  (-> even? odd?)
  (+ n 1))
```

Beyond Racket’s particular implementation, this idea of Design by Contract [3] scales to large programs [11] and complex specifications [4]. The trouble with a guarded function such as even++ is that each one call to the function triggers three function calls. An application (even++ 4) must first assert that 4 is an even number, then apply even++, and finally assert that 5 is an odd number. Checking the precondition even? is a necessary overhead, but clearly the postcondition odd? will never fail. In other words, (even? v) implies (odd? (even++ v)) for any possible input v.

Contracts on higher-order functions may lead to even worse overhead. To illustrate the potential issue, consider an identity function specified over functions similar to even++:

```
(define/contract (identity f)
  (-> (-> even? odd?) (-> even? odd?))
  f)
```

The application (identity even++) consequently applies the contract (-> even? odd?) twice more to even++. Calling ((identity even++) 4) asserts that 4 is an even number three times. In more realistic programs, similar higher-order contracts have led to order-of-magnitude [10] and exponential [9] performance overhead.

2. Applying Rosette
Rosette [12] is a solver-aided programming language. Programs written in Rosette and containing so-called symbolic variables may query an SMT solver to learn about the behavior of their components. For example, the following program fragment asks the solver for an even integer x such that (even++ x) is not odd.

```
(require rosette)
(define-symbolic x integer?)
(solve (assert (even? x)
             (not (odd? (even++ x)))))
```

The solver cannot find an appropriate substitute for x, therefore solve reports an unsatisfiable problem. In terms of the contract on even++, this unsatisfiability result implies that the postcondition odd? is always satisfied. As far as the solver can tell, the function even++ faithfully converts even numbers to odd numbers.

Given two function contracts, a similar query can determine whether the precondition of one contract is stronger than the precondition of another. If so, the weaker precondition is unnecessary. Likewise, Rosette can identify redundant postconditions and ultimately show that a function such as identity need not apply two contracts to its input value.

The above observations suggest that a contract library implemented using Rosette could avoid some inefficiencies. We are currently implementing one such contract library as an extension of Racket’s contract library. This library has two key ingredients:

- Contract attachment forms (e.g. define/contract) generate code for solver queries at compile-time based on a given contract (e.g. (-> even? odd?)).
- At runtime, the generated code invokes the SMT solver before applying a contract to a value. If part of a contract is redundant, the library replaces that part with a no-op.

Crucially, the library API mimics racket/contract.

3. Model
Figure 1 describes a pure λ calculus with natural numbers, booleans, and a simple form for attaching contracts to functions. In particular, an expression e \(\otimes\) \(\langle c_1, c_2 \rangle\) guards the function e with precondition c_1 and postcondition c_2. Note that c_1 and c_2 must be typed predicates of the form \(\{ v \in S \mid (e v) \}\). Without the type S, Rosette cannot efficiently search for values that satisfy the predicate e.
The purpose of the model is to describe when it is safe to remove part of a contract. The operational rules E-POST and E-WRAP specify two such cases.\textsuperscript{[a]} The former rule states that a postcondition is unnecessary when there is no value that both passes the precondition and whose image under \(e\) fails the postcondition. The latter rule states that an entire contract is unnecessary if applied to a value guarded with a stronger contract. Intuitively, a predicate \(c_1\) is at least as strong as \(c_1'\) if there are no values that pass \(c_1\) and fail \(c_1'\). Both these rules yield have simple implementations in Rosette.

4. Challenges, and some Solutions

At least four technical challenges stand between the model and a practical Rosette-backed contract library for Racket.

First, Rosette is only sound up to a given bitwidth. When asked to generate values in Nat, Rosette searches for integers within a finite range. Because Racket allows arbitrary-precision numbers\textsuperscript{[8]}, it is not safe to replace contracts with no-ops. Instead of \(\lambda x.\ \text{true}\) as shown in E-POST, a sound implementation must use a predicate \(c^+\) such that \(\langle e \otimes (c_1, c_1')\rangle v\) first checks that \(v\) is within Rosette’s range, and only then skips the postcondition. Similarly, E-WRAP cannot discard \(c_1'\) and \(c_2'\).

Second, the implementation must associate types to contracts statically and at runtime. The latter requirement means that the Rosette library cannot solve for Racket contracts dynamically. Rather, the library requires a subtype of such contracts tagged with explicit type information. Along the same lines, it must be possible to extract the precondition and postcondition from the runtime representation of a function contract.

Third, the set of types that Rosette can efficiently solve for may be too small to yield a practical implementation. For example, Rosette cannot currently solve for tuples, strings, or lists.

Finally, invoking an SMT solver at runtime may cause a program to run slower than it did with normal contracts. The hope is that the investment of running the solver removes many subsequent predicate checks. But it is currently unclear how many checks are required to offset the cost of the solver.

Regarding the bitwidth, our library currently uses Racket’s dependent function contracts to ensure soundness. A short-term goal is to implement a low-level contract combinator that performs the same check more efficiently. This low-level combinator will also carry its type at runtime, solving the second issue. To measure the cost of the solver, we plan to add feature-specific profiling\textsuperscript{[9]} around solver calls within Rosette.

5. Related Work

Nguyen et al.\textsuperscript{[8]} pioneered the use of symbolic execution to verify that a program component satisfies its contract. Their success inspires this project.

\textsuperscript{[a]} There are four ways to remove contracts from an expression of the form \(e \otimes (c_1, c_2) \otimes (c_1', c_2')\).

References