Randomized Testing in PLT Redex

Casey Klein
University of Chicago
clklein@cs.uchicago.edu

Robert Bruce Findler
Northwestern University
robby@eecs.northwestern.edu

Abstract

This paper presents new support for randomized testing in PLT Redex, a domain-specific language for formalizing operational semantics. In keeping with the overall spirit of Redex, the testing support is as lightweight as possible—Redex programmers simply write down predicates that correspond to facts about their calculus and the tool randomly generates program expressions in an attempt to falsify the predicates. Redex's automatic test case generation begins with simple expressions, but as time passes, it broadens its search to include increasingly complex expressions. To improve test coverage, test generation exploits the structure of the model's metafunction and reduction relation definitions.

The paper also reports on a case-study applying Redex's testing support to the latest revision of the Scheme standard. Despite a community review period, as well as a comprehensive, manually-constructed test suite, Redex's random test case generation was able to identify several bugs in the semantics.

Categories and Subject Descriptors D.2.5 [*Software Engineering*]: Testing and Debugging—testing tools; F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—assertions, invariants, mechanical verification; D.2.4 [*Software Engineering*]: Software / Program Verification—assertion checkers; D.3.1 [*Programming Languages*]: Formal Definitions and Theory

General Terms Languages, Design

Keywords Randomized test case generation, lightweight formal models, operational semantics

1. Introduction

Much like software engineers have to cope with maintaining a program over time with changing requirements, semantics engineers have to maintain formal systems as they evolve over time. In order to help maintain such formal systems, a number of tools that focus on providing support for either proving or checking proofs of such systems have been built (Hol [13], Isabelle [15], Twelf [16], and Coq [22] being some of the most prominent).

In this same spirit, we have built PLT Redex [8, 12]. Unlike other tools, however, Redex's goal is to be as lightweight as possible. In particular, our goal is that Redex programmers should write down little more than they would write in a formal model of their system in a paper and to still provide them with a suite of tools for working with their semantics. Specifically, Redex programmers write down the language, reduction rules, and any relevant metafunctions for their calculi, and Redex provides a stepper, hand-written unit test suite support, automatic typesetting support, and a number of other tools.

To date, Redex has been used with dozens of small, paper-size models and a few large models, the most notable of which is the formal semantics in the current standard of Scheme [21]. Redex is also the subject of a book on operational semantics [7].

Inspired by QuickCheck [5], we recently added a random test case generator to Redex and this paper reports on our experience with it. The test case generator has found bugs in every model we have tested with it, even the most well-tested and widely used models (as discussed in section 4).

The rest of the paper is organized as follows. Section 2 introduces Redex by presenting the formalization of a toy programming language. Section 3 demonstrates the application of Redex's randomized testing facilities. Section 4 presents our experience applying randomized testing to a formal model of R^6RS Scheme. Section 5 describes the general process and specific tricks that Redex uses to generate random terms. Finally, section 6 discusses related work, and section 7 concludes.

2. Redex by Example

Redex is a domain-specific language, embedded in PLT Scheme. It inherits the syntactic and lexical structure from PLT Scheme and allows Redex programmers to embed full-fledged Scheme code into a model, where appropriate. It also inherits DrScheme, the program development environment, as well as a large standard library. This section introduces Redex and context-sensitive reduction semantics through a series of examples, and makes only minimal assumptions about the reader's knowledge of operational semantics. In an attempt to give a feel for how programming in Redex works, this section is peppered with code fragments; each of these expressions runs exactly as given (assuming that earlier definitions have been evaluated) and the results of evaluation are also as shown (although we are using a printer that uses a notation that matches the input notation for values, instead of the standard Scheme printer).

Our goal with this section is to turn the formal model specified in figure 1 into a running Redex program; in section 3, we will test the model. The language in the figure 1 is expression-based, containing application expressions (to invoke functions), conditional expressions, values (i.e., fully simplified expressions), and variables. Values include functions, the plus operator, and numbers.

The `eval` function gives the meaning of each program (either a number or the special token `proc`), and it is defined via a binary relation $\longrightarrow$ on the syntax of programs. This relation, commonly referred to as a standard reduction, gives the behavior of programs in a machine-like way, showing the ways in which an expression can fruitfully take a step towards a value.

Proceedings of the 2009 Scheme and Functional Programming Workshop
California Polytechnic State University Technical Report CPSLO-CSC-09-03

Language

$$e ::= (e\ e \cdots) \mid (\text{if0}\ e\ e\ e) \mid v \mid x$$
$$v ::= \lambda(x \cdots).e \mid + \mid \mathbb{N}$$
$$E ::= [] \mid (v \cdots\ E\ e \cdots) \mid (\text{if0}\ E\ e\ e)$$

Evaluator

$$\text{eval} : e \to \mathbb{N} \cup \{\texttt{proc}\}$$
$$\text{eval}(e) = n, \text{ if } e \longrightarrow^* \lceil n \rceil \text{ for some } n \in \mathbb{N}$$
$$\text{eval}(e) = \texttt{proc}, \text{ if } \begin{cases} e \longrightarrow^* \lambda(x \cdots).e', \text{ or} \\ e \longrightarrow^* + \end{cases}$$

Reduction relation

$$\begin{array}{lll} E[(\text{if0}\ \lceil 0 \rceil\ e_1\ e_2)] & \longrightarrow & E[e_1] \\ E[(\text{if0}\ v\ e_1\ e_2)] & \longrightarrow & E[e_2] \qquad v \neq \lceil 0 \rceil \\ E[((\lambda(x \cdots).e)\ v \cdots)] & \longrightarrow & E[e\{x \leftarrow v, \cdots\}] \\ E[(+\ \lceil n \rceil \cdots)] & \longrightarrow & E[\lceil \textstyle\sum(n \cdots) \rceil] \end{array}$$

Figure 1. Mathematical Model of Core Scheme

The non-terminal E defines evaluation contexts. It gives the order in which expressions are evaluated by providing a rule for decomposing a program into a context—an expression containing a "hole"—and the sub-expression to reduce. The context's hole, written $[]$, may appear either inside an application expression, when all the expressions to the left are already values, or inside the test position of an if0 expression.

The first two reduction rules dictate that an if0 expression can be reduced to either its "then" or its "else" subexpression, based on the value of the test. The third rule says that function applications can be simplified by substitution, and the final rule says that fully simplified addition expressions can be replaced with their sums.

We use various features of Redex (as below) to illuminate the behavior of the model as it is translated to Redex, but just to give a feel for the calculus, here is a sample reduction sequence illustrating how the rules and the evaluation contexts work together.

$$\begin{array}{l} (+\ (\text{if0}\ 0\ 1\ 2)\ (\text{if0}\ 2\ 1\ 0)) \\ \longrightarrow (+\ 1\ (\text{if0}\ 2\ 1\ 0)) \\ \longrightarrow (+\ 1\ 0) \\ \longrightarrow 1 \end{array}$$

Consider the step between the first and second term. Both of the if0 expressions are candidates for reduction, but the evaluation contexts only allow the first to be reduced. Since the rules for if0 expressions are written with $E[]$ outside of the if0 expression, the expression must decompose into some E with the if0 expression in the place where the hole appears. This decomposition is what fails when attempting to reduce the second if0 expression. Specifically, the case for application expressions requires values to the left of the hole, but this is not the case for the second if0 expression.

Like a Scheme program, a Redex program consists of a series of definitions. Redex programmers have all of the ordinary Scheme definition forms (variable, function, structure, etc.) available, as well as a few new definition forms that are specific to operational semantics. For clarity, when we show code fragments, we italicize Redex keywords, to make clear where Redex extends Scheme.

Redex's first definition form is *define-language*. It uses a parenthesized version of BNF notation to define a tree grammar,[1] consisting of non-terminals and their productions. The following defines the same grammar as in figure 1, binding it to the Scheme-level variable L.

```
(define-language L
  (e (e e ...)
     (if0 e e e)
     v
     x)
  (v +
     n
     (λ (x ...) e))
  (E hole
     (v ... E e ...)
     (if0 E e e))
  (n number)
  (x variable-not-otherwise-mentioned))
```

In addition to the non-terminals `e`, `v`, and `E` from the figure, this grammar also provides definitions for numbers `n` and variables `x`. Unlike the traditional notation for BNF grammars, Redex encloses a non-terminal and its productions in a pair of parentheses and does not use vertical bars to separate productions, simply juxtaposing them instead.

Following the mathematical model, the first non-terminal in `L` is `e`, and it has four productions: application expressions, `if0` expressions, values, and variables. The ellipsis is a form of Kleene-star; i.e., it admits repetitions of the pattern preceding it (possibly zero). In this case, this means that application expressions must have at least one sub-expression, corresponding to the function position of the application, but may have arbitrarily many more, corresponding to the function's arguments.

The `v` non-terminal specifies the language's values; it has three productions—one each for the addition operator, numeric literals, and functions. As with application expressions, function parameter lists use an ellipsis, this time indicating that a function can have zero or more parameters.

The `E` non-terminal defines the contexts in which evaluation can occur. The *hole* production gives a place where evaluation can occur, in this case, the top-level of the term. The second production allows evaluation to occur anywhere in an application expression, as long as all of the terms to the left of the have been fully evaluated. In other words, this indicates a left-to-right order of evaluation. The third production dictates that evaluation is allowed only in the test position of an `if0` expression.

The `n` non-terminal generates numbers using the built-in Redex pattern *number*. Redex exploits Scheme's underlying support for numbers, allowing arbitrary Scheme numbers to be embedded in Redex terms.

Finally, the `x` generates all variables except λ, `+`, and `if0`, using *variable-not-otherwise-mentioned*. In general, the pattern *variable-not-otherwise-mentioned* matches all variables except those that are used as literals elsewhere in the grammar.

Once a grammar has been defined, a Redex programmer can use *redex-match* to test whether a term matches a given pattern. It accepts three arguments—a language, a pattern, and an expression—and returns `#f` (Scheme's false), if the pattern does not match, or bindings for the pattern variables, if the term does match. For example, consider the following interaction:

```
> (redex-match L e (term (if0 (+ 1 2) 0)))
#f
```

This expression tests whether `(if0 (+ 1 2) 0)` is an expression according to `L`. It is not, because `if0` must have three subexpressions.

When *redex-match* succeeds, it returns a list of match structures, as in this example.

```
> (redex-match
```

[1] See *Tree Automata Techniques and Applications* [6] for an excellent summary of the properties of tree grammars.

```
   L
   (if0 v e_1 e_2)
   (term (if0 3 0 (λ (x) x))))
(list (make-match
       (list (make-bind 'v 3)
             (make-bind 'e_1 0)
             (make-bind 'e_2 (term (λ (x) x))))))
```

Each element in the list corresponds to a distinct way to match the pattern against the expression. In this case, there is only one way to match it, and so there is only one element in the list. Each match structure gives the bindings for the pattern's variables. In this case, `v` matched `3`, `e_1` matched `0`, and `e_2` matched `(λ (x) x)`. The *term* constructor is absent from the `v` and `e_1` matches because numbers are simultaneously Redex terms and ordinary Scheme values (and this will come in handy when we define the reduction relation for this language).

Of course, since Redex patterns can be ambiguous, there might be multiple ways for the pattern to match the expression. This can arise in two ways: an ambiguous grammar, or repeated ellipses. Consider the following use of repeated ellipses.

```
> (redex-match L
               (n_1 ... n_2 n_3 ...)
               (term (1 2 3)))
(list (make-match
       (list (make-bind 'n_1 (list))
             (make-bind 'n_2 1)
             (make-bind 'n_3 (list 2 3))))
      (make-match
       (list (make-bind 'n_1 (list 1))
             (make-bind 'n_2 2)
             (make-bind 'n_3 (list 3))))
      (make-match
       (list (make-bind 'n_1 (list 1 2))
             (make-bind 'n_2 3)
             (make-bind 'n_3 (list)))))
```

The pattern matches any sequence of numbers that has at least a single element, and it matches such sequences as many times as there are elements in the sequence, each time binding `n_2` to a distinct element of the sequence.

Now that we have defined a language, we can define the reduction relation for that language. The *reduction-relation* form accepts a language and a series of rules that define the relation casewise. For example, here is a reduction relation for `L`. In preparation for Redex's automatic test case generation, we have intentionally introduced a few errors into this definition. The explanatory text does not contain any errors;[2] it simply avoids mention of the mistakes.

```
(define eval-step
  (reduction-relation
   L
   (--> (in-hole E (if0 0 e_1 e_2))
        (in-hole E e_1)
        "if0 true")
   (--> (in-hole E (if0 v e_1 e_2))
        (in-hole E e_2)
        "if0 false")
   (--> (in-hole E ((λ (x ...) e) v ...))
        (in-hole E (subst (x v) ... e))
        "beta value")
   (--> (in-hole E (+ n_1 n_2))
        (in-hole E ,(+ (term n_1) (term n_2)))
        "+")))
```

[2] We hope.

Each case begins with the arrow `-->` and includes a pattern, a term template, and a name for the case. The pattern indicates when the rule will fire and the term indicates what it should be replaced with.

Each rule begins with an *in-hole* pattern that decomposes a term into an evaluation context `E` and some instruction. For example, consider the first rule. We can use *redex-match* to test its pattern against a sample expression.

```
> (redex-match L
               (in-hole E (if0 0 e_1 e_2))
               (term (+ 1 (if0 0 2 3))))
(list (make-match
       (list (make-bind 'E (term (+ 1 hole)))
             (make-bind 'e_1 2)
             (make-bind 'e_2 3))))
```

Since the match succeeded, the rule applies to the term, with the substitutions for the pattern variables shown. Thus, this term will reduce to `(+ 1 2)`, since the rule replaces the `if0` expression with `e_1`, the "then" branch, inside the context `(+ 1 hole)`. Similarly, the second reduction rule replaces an `if0` expression with its "else" branch.

The third rule defines function application in terms of a meta-function `subst` that performs capture-avoiding substitution; its definition is not shown, but standard.

The relation's final rule is for addition. It exploits Redex's embedding in Scheme to use the Scheme-level `+` operator to perform the Redex-level addition. Specifically, the comma operator is an escape to Scheme and its result is replaced into the term at the appropriate point. The *term* constructor does the reverse, going from Scheme back to a Redex term. In this case, we use it to pick up the bindings for the pattern variables `n_1` and `n_2`.

This "escape" from the object language that we are modeling in Redex to the meta-language (Scheme) mirrors a subtle detail from the mathematical model in figure 1, specifically the use of the $\lceil \cdot \rceil$ operator. In the model that operator translates a number into its textual representation. Consider its use in the addition rule; it defers the definition of addition to the summation operator, much like we defer the definition to Scheme's `+` operator.

Once a Redex programmer has defined a reduction relation, Redex can build reduction graphs, via `traces`. The `traces` function takes a reduction relation and a term and opens a GUI window showing the reduction graph rooted at the given term. Figure 2 shows such a graph, generated from `eval-step` and an `if0` expression. As the screenshot shows, the `traces` window also lets the user adjust the font size and connects to dot [9] to lay out the graphs. Redex can also detect cycles in the reduction graph, for example when running an infinite loop, as shown in figure 3.

In addition to `traces`, Redex provides a lower-level interface to the reduction semantics via the *apply-reduction-relation* function. It accepts a reduction relation and a term and returns a list of the next states, as in the following example.

```
> (apply-reduction-relation eval-step
                            (term (if0 1 2 3)))
(list 3)
```

For the `eval-step` reduction relation, this should always be a singleton list but, in general, multiple rules may apply to the same term, or a single rule may even apply in multiple different ways.

3. Random Testing in Redex

If we intend `eval-step` to model the deterministic evaluation of expressions in our toy language, we might expect `eval-step` to define exactly one reduction for any expression that is not already a value. This is certainly the case for the expressions in figures 2 and 3.

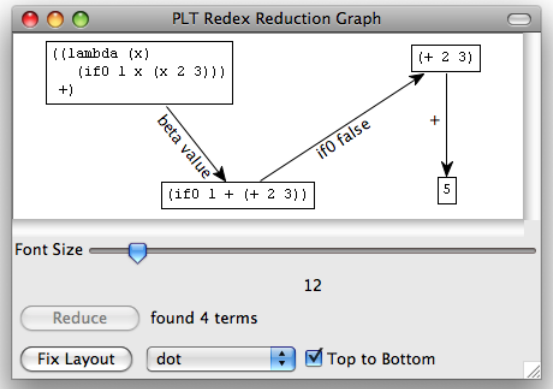


Figure 2. A reduction graph with four expressions

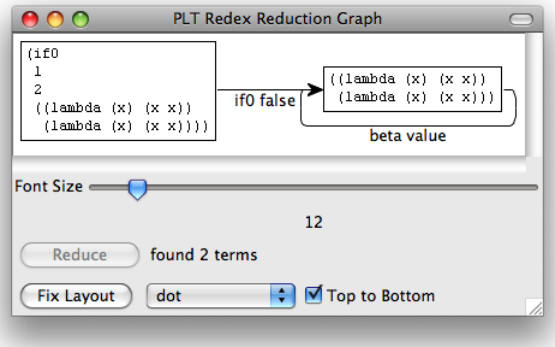


Figure 3. A reduction graph with an infinite loop

To test this, we first formulate a Scheme function that checks this property on one example. It accepts a term and returns true when the term is a value, or when the term reduces just one way, using *redex-match* and *apply-reduction-relation*.

```
;; value-or-unique-step? : term → boolean
(define (value-or-unique-step? e)
  (or (redex-match L v e)
      (= 1 (length (apply-reduction-relation
                    eval-step e)))))
```

Once we have a predicate that should hold for every term, we can supply it to *redex-check*, Redex's random test case generation tool. It accepts a language, in this case L, a pattern to generate terms from, in this case just e, and a boolean expression, in this case, an invocation of the value-or-unique-step? function with the randomly generated term.

```
> (redex-check
   L e
   (value-or-unique-step? (term e)))
counterexample found after 1 attempt:
q
```

Immediately, we see that the property does not hold for open terms. Of course, this means that the property does not even hold for our mathematical model! Often, such terms are referred to as "stuck" states and are ruled out by either a type-checker (in a typed language) or are left implicit by the designer of the model. In this case, however, since we want to uncover all of the mistakes in the model, we instead choose to add explicit error transitions, following how most Scheme implementations actually behave. These rules generally reduces to something of the form (error *description*). For unbound variables, this is the rule:

```
(--> (in-hole E x)
     (error "unbound-id"))
```

It says that when the next term to reduce is a variable (i.e., the term in the hole of the evaluation context is x), then instead reduce to an error. Note that on the right-hand side of the rule, the evaluation context E is omitted. This means that the entire context of the term is simply erased and (error "unbound-id") becomes the complete state of the computation, thus aborting the computation.

With the improved relation in hand, we can try again to uncover bugs in the definition.

```
> (redex-check
   L e
   (value-or-unique-step? (term e)))
counterexample found after 6 attempts:
(+)
```

This result represents a true bug. While the language's grammar allows addition expressions to have an arbitrary number of arguments, our reduction rule only covers the case of two arguments. Redex reports this failure via the simplest expression possible: an application of the plus operator to no arguments at all.

There are several ways to fix this rule. We could add a few rules that would reduce n-ary addition expressions to binary ones and then add special cases for unary and zero-ary addition expressions. Alternatively, we can exploit the fact that Redex is embedded in Scheme to make a rule that is very close in spirit to the rule given in figure 1.

```
(--> (in-hole E (+ n ...))
     (in-hole E ,(apply + (term (n ...))))
     "+")
```

But there still may be errors to discover, and so with this fix in place, we return to *redex-check*.

```
> (redex-check L
               e
               (value-or-unique-step? (term e)))
checking ((λ (i) 0)) raises an exception
syntax: incompatible ellipsis match counts
 for template in: ...
```

This time, *redex-check* is not reporting a failure of the predicate but instead that the input example ((λ (i) 0)) causes the model to raise a Scheme-level runtime error. The precise text of this error is a bit inscrutable, but it also comes with source location highlighting that pinpoints the relation's application case. Translated into English, the error message says that the this rule is ill-defined in the case when the number of formal and actual parameters do not match. The ellipsis in the error message indicates that it is the ellipsis operator on the right-hand side of the rule that is signaling the error, since it does not know how to construct a term unless there are the same number of xs and vs.

To fix this rule, we can add subscripts to the ellipses in the application rule

```
(--> (in-hole E ((λ (x ..._1) e) v ..._1))
     (in-hole E (subst (x v) ... e))
     "beta value")
```

Duplicating the subscript on the ellipses indicates to Redex that it must match the corresponding sequences with the same length.

Again with the fix in hand, we return to *redex-check*:

```
> (redex-check L
               e
```

```
              (value-or-unique-step? (term e)))
counterexample found after 196 attempts:
(if0 0 m +)
```

This time, Redex reports that the expression `(if0 0 m +)` fails, but we clearly have a rule for that case, namely the first `if0` rule. To see what is happening, we apply `eval-step` to the term directly, using *apply-reduction-relation*, which shows that the term reduces two different ways.

```
> (apply-reduction-relation eval-step
                            (term (if0 0 m +)))
(list (term +)
      (term m))
```

Of course, we should only expect the second result, not the first. A closer look reveals that, unlike the definition in figure 1, the second `eval-step` rule applies regardless of the particular `v` in the conditional. We fix this oversight by adding a *side-condition* clause to the earlier definition.

```
(--> (in-hole E (if0 v e_1 e_2))
     (in-hole E e_2)
     (side-condition (not (equal? (term v) 0)))
     "if0 false")
```

Side-conditions are written as ordinary Scheme code, following the keyword *side-condition*, as a new clause in the rule's definition. If the side-condition expression evaluates to `#f`, then the rule is considered not to match.

At this point, *redex-check* fails to discover any new errors in the semantics. The complete, corrected reduction relation is shown in figure 4.

In general, after this process fails to uncover (additional) counterexamples, the task becomes assessing *redex-check*'s success in generating well-distributed test cases. Redex has some introspective facilities, including the ability to count the number of reductions that fire. With this reduction system, we discover that nearly 60% of the time, the random term exercises the free variable rule. To get better coverage, Redex can take into account the structure of the reduction relation. Specifically, providing the `#:source` keyword tells Redex to use the left-hand sides of the rules in `eval-step` as sources of expressions.

```
> (redex-check L
               e
               (value-or-unique-step? (term e))
               #:source eval-step)
```

With this invocation, Redex distributes its effort across the relation's rules by first generating terms matching the first rule's left-hand side, then terms matching the second term's left-hand side, etc. Note that this also gives Redex a bit more information; namely that all of the left-hand sides of the `eval-step` relation should match the non-terminal `e`, and thus Redex also reports such violations. In this case, however, Redex discovers no new errors, but it does get an even distribution of the uses of the various rewriting rules.

4. Case Study: R^{6}RS Formal Semantics

The most recent revision of the specification for the Scheme programming language (R^{6}RS) [21] includes a formal, operational semantics defined in PLT Redex. The semantics was vetted by the editors of the R^{6}RS and was available for review by the Scheme community at large for several months before it was finalized.

In an attempt to avoid errors in the semantics, it came with a hand-crafted test suite of 333 test expressions. Together these tests explore 6,930 distinct program states; the largest test case explores 307 states. The semantics is non-deterministic in order to avoid over-constraining implementations. That is, an implementation conforms to the semantics if it produces any one of the possible results given by the semantics. Accordingly the test suite contains terms that explore multiple reduction sequence paths. There are 58 test cases that contain at least some non-determinism and, the test case with the most non-determinism visits 17 states that each have multiple subsequent states.

```
(define complete-eval-step
  (reduction-relation
   L

   ;; corrected rules
   (--> (in-hole E (if0 0 e_1 e_2))
        (in-hole E e_1)
        "if0 true")
   (--> (in-hole E (if0 v e_1 e_2))
        (in-hole E e_2)
        (side-condition (not (equal? (term v) 0)))
        "if0 false")
   (--> (in-hole E ((λ (x ..._1) e) v ..._1))
        (in-hole E (subst (x v) ... e))
        "beta value")
   (--> (in-hole E (+ n ...))
        (in-hole E ,(apply + (term (n ...))))
        "+")

   ;; error rules
   (--> (in-hole E x)
        (error "unbound-id"))
   (--> (in-hole E ((λ (x ...) e) v ...))
        (error "arity")
        (side-condition
         (not (= (length (term (x ...)))
                 (length (term (v ...)))))))
   (--> (in-hole E (+ n ... v_1 v_2 ...))
        (error "+")
        (side-condition (not (number? (term v_1)))))
   (--> (in-hole E (v_1 v_2 ...))
        (error "app")
        (side-condition
         (and (not (redex-match L + (term v_1)))
              (not (redex-match L
                                (λ (x ...) e)
                                (term v_1)))))))))
```

Figure 4. The complete, corrected reduction relation

Despite all of the careful scrutiny, Redex's randomized testing found four errors in the semantics, described below. The remainder of this section introduces the semantics itself (section 4.1), describes our experience applying Redex's randomized testing framework to the semantics (sections 4.2 and 4.3), discusses the current state of the fixes to the semantics (section 4.4), and quantifies the size of the bug search space (section 4.5).

4.1 The R^{6}RS Formal Semantics

In addition to the features modeled in Section 2, the formal semantics includes: mutable variables, mutable and immutable pairs, variable-arity functions, object identity-based equivalence, quoted expressions, multiple return values, exceptions, mutually recursive bindings, first-class continuations, and `dynamic-wind`. The formal semantics's grammar has 41 non-terminals, with a total of 144 productions, and its reduction relation has 105 rules.

The core of the formal semantics is a relation on program states that, in a manner similar to `eval-step` in Section 2, gives the

behavior of a Scheme abstract machine. For example, here are two of the key rules that govern function application.

```
(--> (in-hole P_1 ((λ (x_1 x_2 ..._1) e_1 e_2 ...)
                   v_1 v_2 ..._1))
     (in-hole P_1 ((r6rs-subst-one
                    (x_1 v_1
                     (λ (x_2 ...) e_1 e_2 ...)))
                   v_2 ...))
     "6appN"
     (side-condition
       (not (term (Var-set!d?
                   (x_1
                    (λ (x_2 ...) e_1 e_2 ...)))))))

(--> (in-hole P_1 ((λ () e_1 e_2 ...)))
     (in-hole P_1 (begin e_1 e_2 ...))
     "6app0")
```

These rules apply only to applications that appear in an evaluation context `P_1`. The first rule turns the application of an n-ary function into the application of an $n-1$-ary function by substituting the first actual argument for the first formal parameter, using the metafunction `r6rs-subst-one`. The side-condition ensures that this rule does not apply when the function's body uses the primitive `set!` to mutate the first parameter's binding; instead, another rule (not shown) handles such applications by allocating a fresh location in the store and replacing each occurrence of the parameter with a reference to the fresh location. Once the first rule has substituted all of the actual parameters for the formal parameters, we are left with a nullary function in an empty application, which is covered by the second rule above. This rule removes both the function and the application, leaving behind the body of the function in a `begin` expression.

The R^{6}RS does not fully specify many aspects of evaluation. For example, the order of evaluation of function application expressions is left up to the implementation, as long as the arguments are evaluated in a manner that is consistent with some sequential ordering (i.e., evaluating one argument halfway and then switching to another argument is disallowed). To cope with this in the formal semantics, the evaluation contexts for application expressions are not like those in section 2, which force left to right evaluation, nor do they have the form `(e_1 ... E e_2 ...)`, which would allow non-sequential evaluation; instead, the contexts that extend into application expressions take the form `(v_1 ... E v_2 ...)` and thus only allow evaluation when there is exactly one argument expression to evaluate. To allow evaluation in other application contexts, the reduction relation includes the following rule.

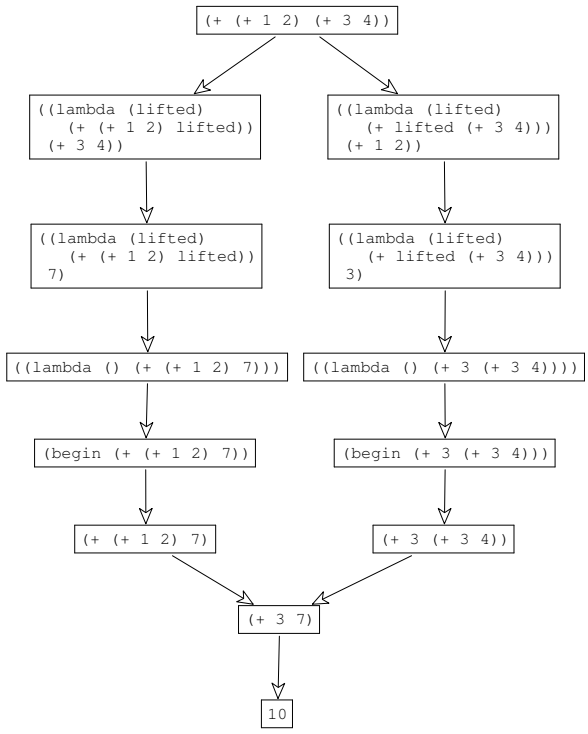
```
(--> (in-hole P_1 (e_1 ... e_i e_i+1 ...))
     (in-hole P_1
       ((λ (x) (e_1 ... x e_i+1 ...)) e_i))
     "6mark"
     (fresh x)
     (side-condition (not (v? (term e_i))))
     (side-condition
      (ormap (λ (e) (not (v? e)))
             (term (e_1 ... e_i+1 ...)))))
```

This rule non-deterministically lifts one subexpression out of the application, placing it in an evaluation context where it will be immediately evaluated then substituted back into the original expression, by the rule `"6appN"`. The *fresh* clause binds `x` such that it does not capture any of the free variables in the original application. The first side-condition ensures that the lifted term is not yet a value, and the second ensures that there is at least one other non-value in the application expression (otherwise the evaluation contexts could just allow evaluation there, without any lifting).

As an example, consider this expression:

```
(+ (+ 1 2)
   (+ 3 4))
```

It contains two nested addition expressions. The `"6mark"` rule applies to both of them, generating two lifted expressions, which then reduce in parallel and eventually merge, as shown in this reduction graph (generated and rendered by Redex).



4.2 Testing the Formal Semantics, a First Attempt

In general, a reduction relation like $\rightarrow$ satisfies the following two properties, commonly known as progress and preservation:

progress If p is a closed program state, consisting of a store and a program expression, then either p is either a final result (i.e., a value or an uncaught exception) or p reduces (i.e., there exists a p' such that $p \rightarrow p'$).

preservation If p is a closed program state and $p \rightarrow p'$, then p' is also a closed program state.

Together these properties ensure that the semantics covers all of the cases and thus an implementation that matches the semantics always produces a result (for every terminating program).

4.2.1 Progress

These properties can be formulated directly as predicates on terms. Progress is a simple boolean combination of a `result?` predicate (defined via a *redex-match* that determines if a term is a final result), an `open?` predicate, and a test to make sure that *apply-reduction-relation* finds at least one possible step. The `open?` predicate uses a `free-vars` function (not shown, but 29 lines of Redex code) that computes the free variables of an R^{6}RS expression.

```
;; progress? : program → boolean
(define (progress? p)
  (or (open? p)
      (result? p)
      (not (= 0 (length
                 (apply-reduction-relation
```

```
                reductions
                p))))))

;; open? : program → boolean
(define (open? p)
  (not (= 0 (length (free-vars p)))))
```

Given that predicate, we can use *redex-check* to test it on the R^{6}RS semantics, using the top-level non-terminal (p∗).

```
(redex-check r6rs p∗ (progress? (term p∗)))
```

Bug one This test reveals one bug, a problem in the interaction between `letrec∗` and `set!`. Here is a small example that illustrates the bug.

```
(store ()
       (letrec∗ ([y 1]
                 [x (set! y 1)])
         y))
```

All R^{6}RS terms begin with a store. In general, the store binds variable to values representing the current mutable state in a program. In this example, however, the store is empty, and so `()` follows the keyword `store`.

After the store is an expression. In this case, it is a `letrec∗` expression that binds `y` to `1` then binds `x` to the result of the assignment expression `(set! y 1)`. The informal report does not specify the value produced by an assignment expression, and the formal semantics models this under-specification by rewriting these expressions to an explicit `unspecified` term, intended to represent any Scheme value. The bug in the formal semantics is that it neglects to provide a rule that covers the case where an `unspecified` value is used as the initial value of a `letrec∗` binding.

Although the above expression triggers the bug, it does so only after taking several reduction steps. The `progress?` property, however, checks only for a first reduction step, and so Redex can only report a program state like the following, which uses some internal constructs in the R^{6}RS semantics.

```
(store ((lx-x bh))
  (l! lx-x unspecified))
```

Here (and in the presentation of subsequent bugs) the actual program state that Redex identifies is typically somewhat larger than the example we show. Manual simplification to simpler states is straightforward, albeit tedious.

4.2.2 Preservation

The `preservation?` property is a bit more complex. It holds if the expression has free variables or if each each expression it reduces to is both well-formed according to the grammar of the R^{6}RS programs and has no free variables.

```
;; preservation? : program → boolean
(define (preservation? p)
  (or (open? p)
      (andmap (λ (q)
                (and (well-formed? q)
                     (not (open? q))))
              (apply-reduction-relation
               reductions p))))

(redex-check r6rs p∗ (preservation? (term p∗)))
```

Running this test fails to discover any bugs, even after tens of thousands of random tests. Manual inspection of just a few random program states reveals why: with high probability, a random program state has a free variable and therefore satisfies the property vacuously.

4.3 Testing the Formal Semantics, Take 2

A closer look at the semantics reveals that we can usually perform at least one evaluation step on an open term, since a free variable is only a problem when the reduction system immediately requires its value. This observation suggests testing the following property, which subsumes both progress and preservation: for any program state, either

- it is a final result (either a value or an uncaught exception),
- it does not reduce and it is open, or
- it does reduce, all of the terms it reduces to have the same (or fewer) free variables, and the terms it reduces to are also well-formed R^{6}RS expressions.

The Scheme translation mirrors the English text, using the helper functions `result?` and `well-formed?`, both defined using *redex-match* and the corresponding non-terminal in the R^{6}RS grammar, and `subset?`, a simple Scheme function that compares two lists to see if the elements of the first list are all in the second.

```
(define (safety? p)
  (define fvs (free-vars p))
  (define nexts (apply-reduction-relation
                 reductions p))
  (or (result? p)
      (and (= 0 (length nexts))
           (open? p))
      (and (not (= 0 (length nexts)))
           (andmap (λ (p2)
                     (and (well-formed? p2)
                          (subset? (free-vars p2)
                                   fvs)))
                   nexts))))

(redex-check r6rs p∗ (safety? (term p∗)))
```

The remainder of this subsection details our use of the `safety?` predicate to uncover three additional bugs in the semantics, all failures of the preservation property.

Bug two The second bug is an omission in the formal grammar that leads to a bad interaction with substitution. Specifically, the keyword `make-cond` was allowed to be a variable. This, by itself, would not lead directly to a violation of our safety property, but it causes an error in combination with a special property of `make-cond`—namely that `make-cond` is the only construct in the model that uses strings. It is used to construct values that represent error conditions. Its argument is a string describing the error condition.

Here is an example term that illustrates the bug.

```
(store () ((λ (make-cond) (make-cond ""))
           null)))
```

According to the grammar of R^{6}RS, this is a legal expression because the `make-cond` in the parameter list of the λ expression is treated as a variable, but the `make-cond` in the body of the λ expression is treated as the keyword, and thus the string is in an illegal position. After a single step, however, we are left with this term `(store () (null ""))` and now the string no longer follows `make-cond`, which is illegal.

The fix is simply to disallow `make-cond` as a variable, making the original expression illegal.

Bug three The next bug triggers a Scheme-level error when using the substitution metafunction. When a substitution encounters a λ expression with a repeated parameter, it fails. For example, supplying this expression

```
(store () ((λ (x) (λ (x x) x))
```

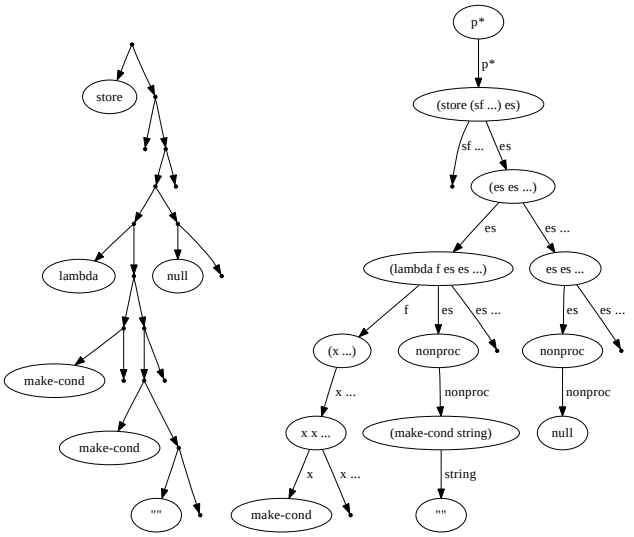


Figure 5. Smallest example of bug two, as a binary tree (left) and as an R^{6}RS expression (right)

```
          1))
```

to the `safety?` predicate results in this error:

```
r6rs-subst-one: clause 3 matched
 (r6rs-subst-one (x 1 (lambda (x) x)))
2 different ways
```

The error indicates that the metafunction `r6rs-subst-one`, one of the substitution helper functions from the semantics, is not well-defined for this input.

According to the grammar given in the informal portion of the R^{6}RS, this program state is not well-formed, since the names bound by the inner λ expression are not distinct. Thus, the fix is not to the metafunction, but to the grammar of the language, restricting the parameter lists of λ expressions to variables that are all distinct.

One could also find this bug by testing the metafunction `r6rs-subst-one` directly. Specifically, testing that the metafunction is well-defined on its input domain also reveals this bug.

Bug four The final bug actually is an error in the definition of the substitution function. The expression

```
(store () ((λ (x) (letrec ([x 1]) 1))
          1))
```

reduces to this (bogus) expression:

```
(store () ((λ () (letrec ((3 1)) 2))))
```

That is, the substitution function replaced the `x` in the binding position of the `letrec` as if the `letrec`-binder was actually a reference to the variable. Ultimately the problem is that `r6rs-subst-one` lacked the cases that handle substitution into `letrec` and `letrec*` expressions.

Redex did not discover this bug until we supplied the `#:source` keyword, which prompted it to generate many expressions matching the left-hand side of the `"6appN"` rule described in section 4.1, on page 31.

4.4 Status of fixes

The version of the R^{6}RS semantics used in this exploration does not match the official version at `http://www.r6rs.org`, due to version skew of Redex. Specifically, the semantics was written for an older version of Redex and *`redex-check`* was not present in that version. Thus, in order to test the model, we first ported it to the latest version of Redex. We have verified that all four of the bugs are present in the original model, and we used *`redex-check`* to be sure that every concrete term in the ported model is also in the original model (the reverse is not true; see the discussion of bug three).

Finally, the R^{6}RS is going to appear as book published by Cambridge Press [20] and the fixes listed here will be included.

Bug #	Uniform, S-expression grammar	R^{6}RS one var, no dups	R^{6}RS one var, with dups	R^{6}RS keywords as vars
1	$D_1(6) > 2^{2^8}$	$p^*(3) > 2^{11}$		
2	$D_0(9) > 2^{2^{11}}$			$p_k^*(6) \approx 2^{556}$
3	$D_1(11) > 2^{2^{13}}$		$p_d^*(8) > 2^{2,969}$	
			$mf(5) > 2^{501}$	
4	$D_1(12) > 2^{2^{14}}$	$p^*(5) > 2^{110}$		

Figure 6. Exhaustive search space sizes for the four bugs

4.5 Search space sizes

Although all four of the bugs in section 4.3 can be discovered with fairly small examples, the search space corresponding to the bug can still be fairly large. In this section we attempt to quantify the size of that search space.

The simplest way to measure the search space is to consider the terms as if they were drawn from an uniform, s-expression representation, i.e., each term is either a pair of terms or a symbol, using repeated pairs to form lists. As an example, consider the left-hand side of figure 5. It shows the parse tree for the smallest expression that discovers bug two, where the dots with children are the pair nodes and the dots without children are the list terminators.

The D_x function computes the number of such trees at a given depth (or smaller), where there are x variables in the expression.

$$D_x(0) = 61 + 1 + x$$
$$D_x(n) = 61 + 1 + x + D_x(n-1)^2$$

The 61 in the definition is the number of keywords in the R^{6}RS grammar, which just count as leaf nodes for this function; the 1 accounts for the list terminator. For example, the parse tree for bug two has depth 9, and there are more than $2^{2^{11}}$ other trees with that depth (or smaller).

Of course, using that grammar can lead to a much larger state space than necessary, since it contains nonsense expressions like `((λ) (λ) (λ))`. To do a more accurate count, we should determine the depth of each of these terms when viewed by the actual R^{6}RS grammar. The right-hand side of figure 5 shows the parse tree for bug two, but where the internal nodes represent expansions of the non-terminals from the R^{6}RS semantics's grammar. In this case, each arrow is labeled with the non-terminal being expanded, the contents of the nodes show what the non-terminal was expanded into, and the dot nodes correspond to expansions of ellipses that terminate the sequence being expanded.

We have computed the size of the search space needed for each of the bugs, as shown in figure 6. The first column shows the size of the search space under the uniform grammar. The second column shows the search space for the first and fourth bugs, using a variant of the R^{6}RS grammar that contains only a single variable and does not allow duplicate variables, i.e., it assumes that bug three has already been fixed, which makes the search space smaller. Still, the search space is fairly large and the function governing its size is complex, just like the R^{6}RS grammar itself. The function is shown in figure 7, along with the helper functions it uses. Each

function computes the size of the search space for one of the non-terminals in the grammar. Because `p*` is the top-level non-terminal, the function p^* computes the total size.

Of course it does not make sense to use that grammar to measure the search space for bug three, since it required duplicate variables. Accordingly we used a slightly different grammar to account for it, as shown in the third column in figure 6. The size function we used, p_d^*, has a subscript d to indicate that it allows duplicate variables and otherwise has a similar structure to the one given in figure 7.

Bug three is also possible to discover by testing the metafunction directly, as discussed in section 4.3. In that case, the search space is given by the *mf* function which computes the size of the patterns used for `r6rs-subst-one`'s domain. Under that metric, the height of the smallest example that exposes the bug is 5. This corresponds to testing a different property, but would still find the bug, in a much smaller search space.

Finally, our approximation to the search space size for bug two is shown in the rightmost column. The k subscript indicates that variables are drawn from the entire set of keywords. Counting this space precisely is more complex than the other functions, because of the restriction that variables appearing in a parameter list must be distinct. Indeed, our p_k^* function over-counts the number of terms in that search space for that reason.[3]

5. Effective Random Term Generation

At a high level, Redex's procedure for generating a random term matching a given pattern is simple: for each non-terminal in the pattern, choose one of its productions and proceed recursively on that pattern. Of course, picking naively has a number of obvious shortcomings. This sections describes how we made the randomized test generation effective in practice.

5.1 Choosing Productions

As sketched above, this procedure has a serious limitation: with non-negligible probability, it produces enormous terms for many inductively defined non-terminals. For example, consider the following language of binary trees:

```
(define-language binary-trees
  (t nil
     (t t)))
```

Each failure to choose the production `nil` expands the problem to the production of two binary trees. If productions are chosen uniformly at random, this procedure will easily construct a tree that exhausts available memory. Accordingly, we impose a size bound on the trees as we generate them. Each time Redex chooses a production that requires further expansion of non-terminals, it decrements the bound. When the bound reaches zero, Redex's restricts its choice to those productions that generate minimum height expressions.

For example, consider generating a term from the `e` non-terminal in the grammar `L` from section 2, on page 27. If the bound is non-zero, Redex freely chooses from all of the productions. Once it reaches zero, Redex no longer chooses the first two productions because those require further expansion of the `e` non-terminal; instead it chooses between the `v` and `x` productions. It is easy to see why `x` is okay; it only generates variables. The `v` non-terminal is also okay, however, because it contains the atomic production `+`.

In general, Redex classifies each production of each non-terminal with a number indicating the minimum number of non-terminal expansion required to generate an expression from the production. Then, when the bound reaches zero, it chooses from one of the productions that have the smallest such number.

Although this generation technique does limit the expressions Redex generates to be at most a constant taller than the bound, it also results in a poor distribution of the leaf nodes. Specifically, when Redex hits the size bound for the `e` non-terminal, it will never generate a number, preferring to generate `+` from `v`. Although Redex will generate some expressions that contain numbers, the vast majority of leaf nodes will be either `+` or a variable.

In general, the factoring of the grammar's productions into non-terminals can have a tremendous effect on the distribution of randomly generated terms because the collection of several productions behind a new non-terminal focuses probability on the original non-terminal's other productions. We have not, however, been able to detect a case where Redex's poor distribution of leaf nodes impedes its ability to find bugs, despite several attempts. Nevertheless, such situations probably do exist, and so we are investigating a technique that produces better distributed leaves.

$$
\begin{array}{ll}
p^*(0) = 1 & p^*(n+1) = (es(n) * sfs(n)) + v(n) + 1 \\
\hat{es}(0) = 1 & \hat{es}(n+1) = (\hat{es}(n) * es(n)) + 1 \\
\hat{\lambda}(0) = 1 & \hat{\lambda}(n+1) = (\hat{\lambda}(n) * \lambda(n)) + 1 \\
Qs(0) = 1 & Qs(n+1) = (Qs(n) * s(n)) + 1 \\
\hat{e}(0) = 1 & \hat{e}(n+1) = (\hat{e}(n) * e(n)) + 1 \\
\hat{v}(0) = 1 & \hat{v}(n+1) = (\hat{v}(n) * v(n)) + 1 \\
\mathcal{E}(0) = 1 & \mathcal{E}(n+1) = (\mathcal{E}(n) * \mathcal{E}^*(n)) \\
 & \quad + (\mathcal{E}(n) * \mathcal{F}o(n)) + 1 \\
\mathcal{E}^*(0) = 0 & \mathcal{E}^*(n+1) = \hat{\lambda}(n) + (e(n)^2 * \chi(n)) + \mathcal{F}^*(n) \\
\mathcal{F}^*(0) = 0 & \mathcal{F}^*(n+1) = \hat{e}(n) + (\hat{e}(n) * \hat{v}(n)) \\
 & \quad + (\hat{e}(n) * v(n)) + (\hat{e}(n) * e(n) * 2) \\
\mathcal{F}o(0) = 0 & \mathcal{F}o(n+1) = (\chi(n) * 2) + \hat{v}(n)^2 + e(n)^2 \\
b(0) = 1 & b(n+1) = v(n) + 1 \\
e(0) = 1 & e(n+1) = (\hat{\lambda}(n) * e(n)) \\
 & \quad + (\hat{e}(n) * e(n) * lb(n) * 2) \\
 & \quad + (\hat{e}(n) * e(n) * 3) + (e(n) * \chi(n) * 2) \\
 & \quad + (e(n)^3 * \chi(n)) + (\chi(n) * 2) + e(n)^3 \\
 & \quad + non\lambda(n) + \lambda(n) + 1 \\
es(0) = 2 & es(n+1) = (\hat{es}(n) * es(n) * f(n)) \\
 & \quad + (\hat{\lambda}(n) * e(n)) \\
 & \quad + (\hat{es}(n) * es(n) * lbs(n) * 2) \\
 & \quad + (\hat{es}(n) * es(n) * 3) \\
 & \quad + (es(n) * \chi(n) * 2) + (\mathcal{E}(n) * \chi(n)^2) \\
 & \quad + (e(n)^3 * \chi(n)) + (\chi(n) * 2) + es(n)^3 \\
 & \quad + non\lambda(n) + p\lambda(n) + seq(n) + sqv(n) \\
 & \quad + 2 \\
f(0) = 1 & f(n+1) = (\chi(n) * 2) + 1 \\
lb(0) = 1 & lb(n+1) = (e(n) * \chi(n)) + 1 \\
lbs(0) = 1 & lbs(n+1) = (es(n) * \chi(n)) + 1 \\
non\lambda(0) = 2 & non\lambda(n+1) = pp(n) + sqv(n) + \chi(n) + 2 \\
pp(0) = 0 & pp(n+1) = \chi(n) * 2 \\
p\lambda(0) = 4 & p\lambda(n+1) = proc1(n) + 15 \\
\lambda(0) = 0 & \lambda(n+1) = (\hat{e}(n) * e(n) * f(n)) \\
 & \quad + (\mathcal{E}(n) * \chi(n)^2) + p\lambda(n) \\
proc1(0) = 7 & proc1(n+1) = 9 \\
s(0) = 1 & s(n+1) = seq(n) + sqv(n) + \chi(n) + 1 \\
seq(0) = 0 & seq(n+1) = (Qs(n) * s(n) * sqv(n)) \\
 & \quad + (Qs(n) * s(n) * \chi(n)) \\
 & \quad + (Qs(n) * s(n)) \\
sf(0) = 0 & sf(n+1) = (b(n) * \chi(n)) + (v(n)^2 * pp(n)) \\
sfs(0) = 1 & sfs(n+1) = sf(n) + 1 \\
sqv(0) = 2 & sqv(n+1) = 3 \\
v(0) = 0 & v(n+1) = non\lambda(n) + \lambda(n) \\
\chi(0) = 0 & \chi(n+1) = 1
\end{array}
$$

Figure 7. Size of the search space for R[6]RS expressions

[3] Amusingly, if we had not found bug three, this would have been an accurate count.

5.2 Non-linear patterns

Redex supports patterns that only match when two parts of the term are syntactically identical. For example, this revision of the binary tree grammar only matches perfect binary trees

```
(define-language perfect-binary-trees
  (t nil
     (t_1 t_1)))
```

because the subscripts in the second production insists that the two sub-trees are identical. Additionally, Redex allows subscripts on the ellipses (as we used in section 3 on page 29) indicating that the length of the matches must be the same.

These two features can interact in subtle ways that affect term generation. For example, consider the following pattern:

```
(x_1 ... y ..._2 x_1 ..._2)
```

This matches a sequence of xs, followed by a sequence of ys followed by a second sequence of xs. The _1 subscripts dictate that the xs must be the same (when viewed as a complete sequence—the individual members of each sequence may be distinct) and the _2 subscripts dictate that the number of ys must be the same as the number of xs. Taken together, this means that the length of the first sequence of x's must be the same as the length of the sequence of ys, but an left-to-right generation of the term will not discover this constraint until after it has already finished generating the ys.

Even worse, Redex supports subscripts with exclamation marks which insist same-named subscripts match different terms; e.g. `(x_!_1 x_!_1)` matches sequences of length two where the elements are different.

To support this in the random test case generator, Redex preprocesses the term to normalize the underscores. In the pattern above, Redex rewrites the pattern to this one

```
(x_1 ..._2 y ..._2 x_1 ..._2)
```

simply changing the first ellipsis to `..._2`.

5.3 Generation Heuristics

Typically, random test case generators can produce very large test inputs for bugs that could also have been discovered with small inputs.[4] To help mitigate this problem, the term generator employs several heuristics to gradually increase the size and complexity of the terms it produces (this is why the generator generally found small examples for the bugs in section 3).

- The term-height bound increases with the logarithm of the number of terms generated.
- The generator chooses the lengths of ellipsis-produced sequences and the lengths of variable names using a geometric distribution, increasing the distribution's expected value with the logarithm of the number of attempts.
- The alphabet from which the generator constructs variable names gradually grows from the English alphabet to the ASCII set and then to the entire unicode character set. Eventually the generator explicitly considers choosing the names of the language's terminals as variables, in hopes of catching rules which confuse the two. The R^{6}RS semantics makes such a mistake, as discussed in section 4.3 (page 4.3), but discovering it is difficult with this heuristic.
- When generating a number, the generator chooses first from the naturals, then from the integers, the reals, and finally the complex numbers, while also increasing the expected magnitude of the chosen number. The complex numbers tend to be especially interesting because comparison operators such as `<=` are not defined on complex numbers.
- Eventually, the generator biases its production choices by randomly selecting a preferred production for each non-terminal. Once the generator decides to bias itself towards a particular production, it generates terms with more deeply nested version of that production, in hope of catching a bug with deeply nested occurrences of some construct.

6. Related Work

Our work was inspired by QuickCheck [5], a tool for doing random test case generation in Haskell. Unlike QuickCheck, however, Redex's test case generation goes to some pains to generate tests automatically, rather than asking the user to specify test case generators. This choice reduces the overhead in using Redex's test case generation, but generators for tests cases with a particular property (e.g., closed expressions) still requires user intervention. QuickCheck also supports automatic test case simplification, a feature not yet provided in Redex. Our work is not the only follow-up to QuickCheck; there are several systems in Haskell [3, 19], Clean [11], and even one for the ACL2 integration with PLT Scheme [14].

There are a number of other tools that test formal semantics. Berghofer and Nipkow [1] have applied random testing to semantics written in Isabelle, with the goal of discovering shallow errors in the language's semantics before embarking on a time-consuming proof attempt. αProlog [2] and Twelf [16] both support Prolog-like search for counterexamples to claims. Most recently, Roberson et al. [17] developed a series of techniques to shrink the search space when searching for counterexamples to type soundness results, with impressive results. Rosu et al. [18] use a rewriting logic semantics for C to test memory safety of individual programs.

There is an ongoing debate in the testing community as to the relative merits of randomized testing and bounded exhaustive testing, with the a priori conclusion that randomized testing requires less work to apply, but that bounded exhaustive testing is otherwise superior. Indeed, while most papers on bounded exhaustive testing include a nominal section on the relative merits of randomized testing (typically showing it to be far inferior), there are also few, more careful, studies that do show the virtues of randomized testing. Visser et al. [23] conducted a case study that concludes (among other things) that randomized testing generally does well, but falls down when testing complex data structures like Fibonacci heaps. Randomized testing in Redex mitigates this somewhat, due to the way programs are written in Redex. Specifically, if such heaps were coded up in Redex, there would be one rule for each different configuration of the heap, enabling Redex to easily generate test cases that would cover all of the interesting configurations. Of course, this does not work in general, due to side-conditions on rules. For example, we were unable to automatically generate many tests for the rule [6applyce][5] in the R^{6}RS formal semantics, due to its side-condition. Ciupa et al. [4] conducted another study that finds randomized testing to be reasonably effective, and Groce et al. [10] conducted a study finding that random test case generation is especially effective early in the software's lifecycle.

7. Conclusion and Future Work

Randomized test generation has proven to be a cheap and effective way to improve models of programming languages in Redex. With only a 13-line predicate (plus a 29-line free variables function), we were able to find bugs in one of the biggest, most well-tested (even

[4] Indeed, for this reason, QuickCheck supports a form of automatic test case simplification that tries to shrink a failing test case.

[5] The is the third rule in figure 11: `http://www.r6rs.org/final/html/r6rs/r6rs-Z-H-15.html#node_sec_A.9`

community-reviewed), mechanized models of a programming language in existence.

Still, we realize that there are some models for which these simple techniques are insufficient, so we don't expect this to be the last word on testing such models. We have begun work to extend Redex's testing support to allow the user to have enough control over the generation of random expressions to ensure minimal properties, e.g. the absence of free variables.

Our plan is to continue to explore how to generate programs that have interesting structural properties, especially well-typed programs. Generating well-typed programs that have interesting distributions is particularly challenging. While it is not too difficult to generate well-typed terms, generating interesting sets of well-typed terms is tricky since there is a lot of freedom in the choice of the generation of types for intermediate program variables, and using those variables in interesting ways is non-trivial.

Acknowledgments Thanks to Matthias Felleisen for his comments on an earlier draft of this paper and to Sam Tobin-Hochstadt for feedback on `redex-check`.

References

[1] S. Berghofer and T. Nipkow. Random testing in Isabelle/HOL. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pages 230–239, 2004.

[2] J. Cheney and A. Momigliano. Mechanized metatheory model-checking. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, pages 75–86, 2007.

[3] J. Christiansen and S. Fischer. Easycheck – test data for free. In *Proceedings of the International Symposium on Functional and Logic Programming*, pages 322–336, 2008.

[4] I. Ciupa, A. Leitner, M. Oriol, and B. Meyer. Experimental assessment of random testing for object-oriented software. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 84–94, 2007.

[5] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 268–279, 2000.

[6] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on: `http://www.grappa.univ-lille3.fr/tata`, 2007. Release October, 12th 2007.

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

[8] R. B. Findler. Redex: Debugging operational semantics. Reference Manual PLT-TR2009-redex-v4.2, PLT Scheme Inc., June 2009. `http://plt-scheme.org/techreports/`.

[9] E. R. Gansner and S. C. North. An open graph visualization system and its applications. *Software Practice and Experience*, 30:1203–1233, 1999.

[10] A. Groce, G. Holzmann, and R. Joshi. Randomized differential testing as a prelude to formal verification. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 621–631, 2007.

[11] P. Koopman, A. Alimarine, J. Tretmans, and R. Plasmeijer. Gast: Generic automated software testing. In *Proceedings of the International Workshop on the Implementation of Functional Languages*, pages 84–100, 2003.

[12] J. Matthews, R. B. Findler, M. Flatt, and M. Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *International Conference on Rewriting Techniques and Applications*, pages 301–312, 2004.

[13] M. Norrish and K. Slind. Hol4, 2007. `http://hol.sourceforge.net/`.

[14] R. Page, C. Eastlund, and M. Felleisen. Functional programming and theorem proving for undergraduates: a progress report. In *Proceedings of the International Workshop on Functional and Declarative Programming in Education*, pages 21–30, 2008.

[15] L. C. Paulson and T. Nipkow. Isabelle. `http://isabelle.in.tum.de/`, 2005.

[16] F. Pfenning and C. Schürmann. Twelf user's guide. Technical Report CMU-CS-98-173, Carnegie Mellon University, 1998.

[17] M. Roberson, M. Harries, P. T. Darga, and C. Boyapati. Efficient software model checking of soundness of type systems. In *Proceedings of the ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages and Applications*, pages 493–504, 2008.

[18] G. Rosu, W. Schulte, and T. F. Serbanuta. Runtime verification of c memory safety. In *Proceedings of the International Workshop on Runtime Verification*, 2009. to appear.

[19] C. Runciman, M. Naylor, and F. Lindblad. Smallcheck and lazy smallcheck: automatic exhaustive testing for small values. In *Proceedings of the ACM SIGPLAN Symposium on Haskell*, pages 37–48, 2008.

[20] M. Sperber, editor. *Revised^6 report on the algorithmic language Scheme*. Cambridge University Press, 2009. to appear.

[21] M. Sperber, R. K. Dybvig, M. Flatt, and A. van Straaten (editors). The Revised^6 Report on the Algorithmic Language Scheme. `http://www.r6rs.org/`, 2007.

[22] The Coq Development Team. The Coq proof assistant reference manual, version 8.0. `http://coq.inria.fr/`, 2004–2006.

[23] W. Visser, C. S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 37–48, 2006.