Experience with Randomized Testing in Programming Language Metatheory

Casey Klein

August 6, 2009

Abstract

We explore the use of QuickCheck-style randomized testing in programming languages metatheory, a methodology proposed to reduce development time by revealing shallow errors early, before a formal proof attempt. This exploration begins with the development of a randomized testing framework for PLT Redex, a domain-specific language for specifying and debugging operational semantics. In keeping with the spirit of Redex, the framework is as lightweight as possible—the user encodes a conjecture as a predicate over the terms of the language, and guided by the structure of the language's grammar, reduction relation, and metafunctions, Redex attempts to falsify the conjecture automatically.

In addition to the details of this framework, we present a tutorial demonstrating its use and two case studies applying it to large language specifications. The first study, a postmortem, applies randomized testing to the formal semantics published with the latest revision of the Scheme language standard. Despite a community review period and a comprehensive, manually-constructed test suite, randomized testing in Redex revealed four bugs in the semantics. The second study presents our experience applying the tool concurrently with the development of a formal model for the MzScheme virtual machine and bytecode verifier. In addition to many errors in our formalization, randomized testing revealed six bugs in the core bytecode verification algorithm in production use. The results of these studies suggest that randomized testing is a cheap and effective technique for finding bugs in large programming language metatheories.

1 Introduction

Most software engineers spend much time testing and little time proving; most semantics engineers, on the other hand, spend much time proving and little time testing. Bringing formal methods into the mainstream of software engineering is a widely held goal, but few advocate a balance between formal and informal methods in the study of programming languages. This paper presents a small step counter to the latter trend, exploring the hypothesis that *randomized testing is a cheap and effective technique for finding bugs in large programming language metatheories*.

To validate this hypothesis, we develop a randomized testing framework for PLT Redex [19, 47], a domain-specific language for context-sensitive reduction systems.

This framework is inspired by the popular QuickCheck library [9] but tailored to testing operational semantics. For example, the test case generator can often improve test coverage by specifically targeting reduction rules that are unlikely to apply to terms generated purely at random (e.g., call-by-value β-reduction, which requires a λ-term of appropriate arity in function position and values in the argument positions). We apply this framework in two large case studies. The first tests the R^{6}RS formal semantics [64], easily finding four bugs that eluded a comprehensive, hand-crafted test suite, in addition to the report's editors and community reviewers. The second integrates randomized testing into the development of a formal model for the MzScheme virtual machine and bytecode verifier, finding six long-standing bugs in the core verification algorithm and twenty-two bugs in our formalization, despite careful manual testing of each.

Our experience in this second case study suggests another role for randomized testing. Late in the model's development, I made a global change to the structure of the bytecode verification algorithm. Knowing this change required several other changes, I systematically peformed the other changes, ran the hand-crafted test suite, and on a whim, ran the test case generator, which revealed a forgotten case, much to my surprise. This bug suggested two additional classes of changes, which I made before running a second round of randomized tests. This round too discovered a bug, which in turn suggested one more class of changes. Randomized tests following this latest fix revealed yet another neglected case. Indeed, this happened *three times*, after one seemingly simple change. Cases like this one illustrate the value of randomized testing as a complement to manual testing.

The rest of this 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 describes the general process and specific tricks that Redex uses to generate random terms. Section 5 and 6 presents the case studies. Section 7 reviews related work, and section 8 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.

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*.

[1]See *Tree Automata Techniques and Applications* [13] for an excellent summary of the properties of tree grammars.

Language

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

Evaluator

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

Reduction relation

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

Figure 1: Mathematical Model of Core Scheme

```
(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
   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 case-wise. 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)))
        "+")))
```

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-**

[2]We hope.

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 metafunction *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*.

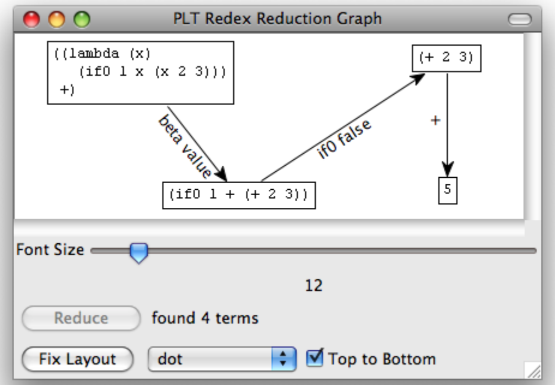
This "escape" from the object language that we are modeling in Redex to the meta-language (Scheme) mirrors a subtle detail from the mathematical model in figure 1, specifically the use of the $\lceil \cdot \rceil$ operator. In the model that operator translates a number into its textual representation. 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 [23] 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.



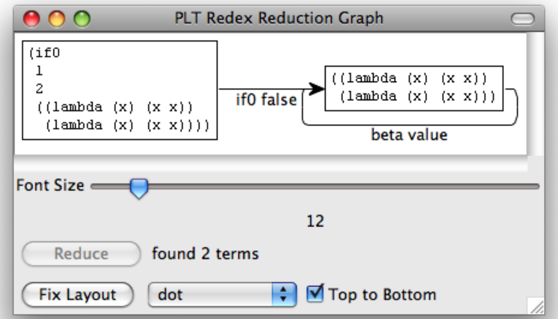
```
((lambda (x)
   (if0 1 x (x 2 3)))
 +)
```

```
(if0 1 + (+ 2 3))
```

```
(+ 2 3)
```

```
5
```

Figure 2: A reduction graph with four expressions



```
(if0
 1
 2
 ((lambda (x) (x x))
  (lambda (x) (x x))))
```

```
((lambda (x) (x x))
 (lambda (x) (x x)))
```

Figure 3: A reduction graph with an infinite loop

3 Randomized 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.

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 *x*s and *v*s.

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 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.

```
              (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

4.1 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 3. 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.

4.2 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 10) 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 *x*s, followed by a sequence of *y*s followed by a second sequence of *x*s. The *_1* subscripts dictate that the *x*s 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 *y*s must be the same as the number of *x*s. 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 *y*s, but an left-to-right generation of the term will not discover this constraint until after it has already finished generating the *y*s.

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.

4.3 Generation Heuristics

Typically, random test case generators can produce very large test inputs for bugs that could also have been discovered with small inputs.[3] 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

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

the two. The R^{6}RS semantics makes such a mistake, as discussed in section 5.3 (page 5.3), 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.

5 Case Study: R^{6}RS Formal Semantics

The most recent revision of the specification for the Scheme programming language (R^{6}RS) [64] 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.

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 5.1), describes our experience applying Redex's randomized testing framework to the semantics (sections 5.2 and 5.3), discusses the current state of the fixes to the semantics (section 5.4), and quantifies the size of the bug search space (section 5.5).

5.1 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.

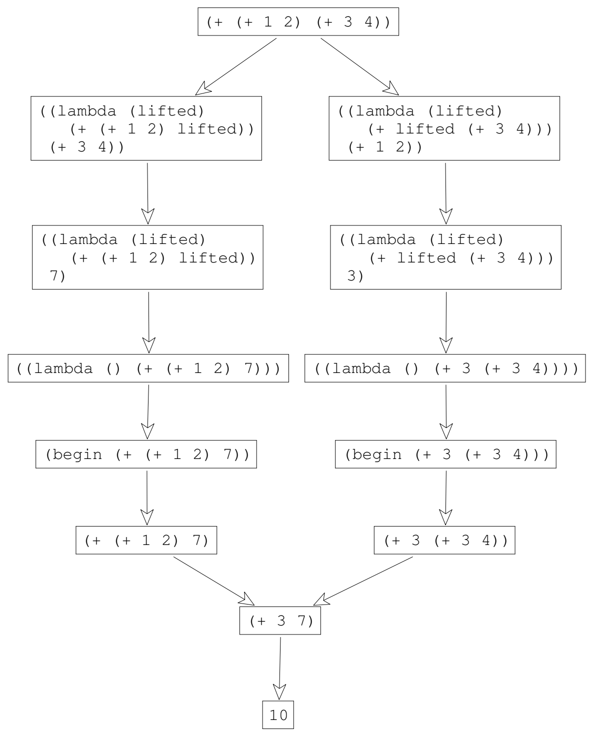
```
(--> (in-hole P_1 (e_1 ... e_i e_i+1 ...))
     (in-hole P_1
        ((λ (x) (e_1 ... x e_i+1 ...)) e_i))
     "6mark"
     (fresh x)
     (side-condition (not (v? (term e_i))))
     (side-condition
      (ormap (λ (e) (not (v? e)))
             (term (e_1 ... e_i+1 ...)))))
```

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).


(+ (+ 1 2) (+ 3 4))
((lambda (lifted) (+ (+ 1 2) lifted)) (+ 3 4))
((lambda (lifted) (+ lifted (+ 3 4))) (+ 1 2))
((lambda (lifted) (+ (+ 1 2) lifted)) 7)
((lambda (lifted) (+ lifted (+ 3 4))) 3)
((lambda () (+ (+ 1 2) 7)))
((lambda () (+ 3 (+ 3 4))))
(begin (+ (+ 1 2) 7))
(begin (+ 3 (+ 3 4)))
(+ (+ 1 2) 7)
(+ 3 (+ 3 4))
(+ 3 7)
10

5.2 Testing the Formal Semantics, a First Attempt

In general, a reduction relation like $\longrightarrow$ 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 $p \longrightarrow p'$).

preservation If p is a closed program state and $p \longrightarrow p'$, 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).

5.2.1 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^6RS 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^6RS 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.

5.2.2 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^6RS 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.

5.3 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

$$(\mathit{store}\ ()\ ((\lambda\ (x)\ (\lambda\ (x\ x)\ x))\ 1))$$

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

```
r6rs-subst-one: clause 3 matched
 (r6rs-subst-one (x 1 (lambda (x 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

$$(\mathit{store}\ ()\ ((\lambda\ (x)\ (\mathit{letrec}\ ([x\ 1])\ 1))\ 1))$$

reduces to this (bogus) expression:

$$(\mathit{store}\ ()\ ((\lambda\ ()\ (\mathit{letrec}\ ((3\ 1))\ 2))))$$

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 5.1, on page 16.

5.4 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 [63] and the fixes listed here will be included.

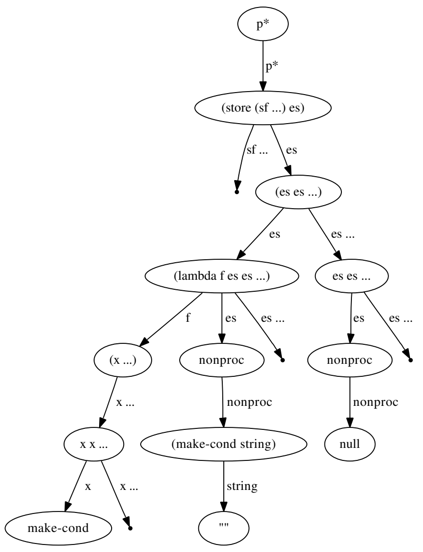


Figure 5: Smallest example of bug two, as a binary tree (left) and as an R^{6}RS expression (right)

5.5 Search space sizes

Although all four of the bugs in section 5.3 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.

$$\begin{aligned} D_x(0) &= 61 + 1 + x \\ D_x(n) &= 61 + 1 + x + D_x(n-1)^2 \end{aligned}$$

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

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

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 5.3. 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.[4]

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

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

Figure 7: Size of the search space for R^{6}RS expressions

6 Case Study: The MzScheme Machine and Bytecode Verifier

Our experience with the R^{6}RS formal semantics suggests that randomized testing may be fruitfully applied to an off-the-shelf semantics, without the need for significant changes to accommodate testing. To explore the use of randomized testing *during* the development process, we integrated Redex's randomized testing features into the development of a formal model of the MzScheme virtual machine. This model provides an operational semantics for an abstract machine and a formalization of the bytecode verification algorithm used in the production virtual machine. Using randomized testing, we checked two properties of the model. First, if the bytecode verifier accepts a program, then the abstract machine does not get stuck while evaluating that program. Second, optimizations modeled in the abstract machine do not change the meaning of programs accepted by the bytecode verifier. Section 6.9 states these properties formally.

Our usual process for developing such models includes the manual construction of a substantial test suite. We continued this practice for the virtual machine model, performing randomized tests only after a change or new feature passed the existing test suite. To provide some idea of the size of the hand-crafted test suite, the suite comprises 192 tests: 90 tests for the 71 cases in the definition of the abstract machine, and 102 tests for the 86 cases in the definition of the verification algorithm.

Despite these hand-crafted tests, randomized testing discovered 22 errors in our formalization of the abstract machine and verification algorithm (i.e., bugs present in our model but not the virtual machine's production implementation). Our formalization of the virtual machine included the first code review of its verification algorithm, and a fresh set of eyes discovered 7 bugs in the algorithm before we were able to run randomized tests on the model. To help gauge the effectiveness of Redex's randomized testing framework, we intentionally left these bugs in the model to see if they would be found.

The remainder of this section provides an overview of the MzScheme bytecode language (section 6.1), defines the abstract machine's operational semantics (sections 6.2–6.7), formalizes the bytecode verification algorithm (section 6.8), and presents the results of randomized testing (section 6.9).

6.1 Bytecode Overview

The MzScheme virtual machine is a stack-based machine. It has neither programmer-visible registers (e.g., as in the JVM[45]) nor explicit variables (e.g., as in the SECD machine[40]); instead, bytecode specifies its operands as offsets from the top of a stack of values maintained by the machine. Figure 8 gives the grammar. The first six expression forms load the value stored at a given stack offset, the next three push a value on the stack, and the four after those update the value at a given offset. The remaining productions correspond to forms in MzScheme's surface-level syntax.

The rest of this section demonstrates the bytecode language by example, showing surface-level expressions and their translations to bytecode, beginning with the following procedure.

```
(λ (x y) (begin (x) (x) (y)))
```

This procedure's body translates to the following bytecode.

```
(seq (application (loc 0))
     (application (loc-clr 0))
     (application (loc-noclr 1)))
```

The **loc**, **loc-clr**, and **loc-noclr** forms load the value stored at the given stack offset. In this case, the procedure's caller pushes *x* and *y* on the stack, and the procedure's body retrieves them using the offsets 0 and 1. The body's second reference to *x* uses **loc-clr** rather than **loc** because **loc-clr** clears the target slot after reading it, allowing the compiler to produce safe-for-space bytecode [1, 10]. The **loc-noclr** behaves just like **loc** at runtime; the "noclr" annotation serves only as a promise that no subsequent instruction clears this slot, helping to ensure the safety of the machine's optimizations.

In the example above, the procedure's local variables remain at fixed offsets from top of the stack, but in general, a variable's relative location may shift as the procedure executes. For example, consider the following Scheme procedure.

```
(λ (x)
  (begin
    x
    (let ([y x])
      (begin y x))))
```

Its body corresponds to the following bytecode, in which the **seq** and **let-one** expressions correspond respectively to the input's **begin** and **let** expressions.

```
(seq (loc 0) ; x
     (let-one (loc 1) ; x
       (seq (loc 0) ; y
            (loc 1)))) ; x
```

The first *x* reference uses offset 0, but the third reference uses offset 1, because the **let-one** expression pushes *y*'s value on the stack before execution reaches the body of the **let-one** expression. In fact, this push occurs even before execution reaches the **let-one**'s first sub-expression, and consequently the second *x* reference also uses offset 1.

When a **let**-bound variable is the target of a **set!** expression, the MzScheme compiler represents that variable as a heap-allocated box. Consider the body of the following procedure, for example.

```
(λ (x y)
  (let ([z x])
    (begin (set! z y)
           z)))
```

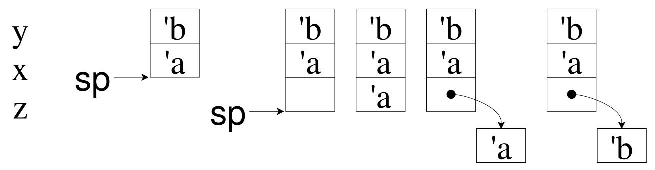
With this representation, the expression corresponds to the following bytecode.

```
(let-void 1
  (install-value 0 (loc 1)
    (boxenv 0
      (install-value-box 0 (loc 2)
        (loc-box 0)))))
```

```
e ::= (loc n)
    | (loc-noclr n)
    | (loc-clr n)
    | (loc-box n)
    | (loc-box-noclr n)
    | (loc-box-clr n)

    | (let-one e e)
    | (let-void n e)
    | (let-void-box n e)

    | (boxenv n e)
    | (install-value n e e)
    | (install-value-box n e e)

    | (application e e ...)          l ::= (lam (τ ...) (n ...) e)
    | (seq e e e ...)                v ::= number
    | (branch e e e)                     | void
    | (let-rec (l ...) e)                | 'variable
    | (indirect x)                       | b
    | (proc-const (τ ...) e)         τ ::= val | ref
    | (case-lam l ...)               n ::= natural
    | l                              b ::= #t | #f
    | v                           x, y ::= variable
```

Figure 8: The grammar for bytecode expressions e.

And this is the progression of the values stack as the machine evaluates the bytecode, assuming the procedure's caller supplies 'a for *x* and 'b for *y*.



First, the **let-void** expression pushes 1 uninitialized slot on the stack. Second, an **install-value** expression initializes that slot with *x*'s value, which is now at offset 1. Third, a **boxenv** expression allocates a fresh box containing the value at offset 0 then writes a pointer to that box at offset 0. Fourth, an **install-value-box** expression writes *y*'s value, now at offset 2, into the box referenced by the pointer at offset 0. Finally, a **loc-box** expression retrieves the value in the box.

The **application** form has one subtlety. As the machine evaluates an expression (**application** $e_0 \ldots e_n$), it must record the result from each sub-expression e_i that it evaluates. To accommodate these intermediate results, the machine pushes n uninitialized slots on the stack before evaluating any sub-expression. This space suffices to hold all prior results while the machine evaluates the final sub-expression. For example, consider the bytecode for the body of the procedure (λ (*x y z*) (*x y z*)).

(**application** (**loc** 2) (**loc** 3) (**loc** 4))

This **application** produces two intermediate results, the values fetched by (**loc** 2) and (**loc** 3), and so the machine pushes two uninitialized slots when it begins evaluating the **application**. This push shifts *x*'s offset from 0 to 2, *y*'s offset from 1 to 3, and *z*'s offset from 2 to 4.

A **lam** expression denotes a procedure. This form includes the stack locations of the procedure's free variables. Evaluating a **lam** expression captures the values at these offsets, and applying the result unpacks the captured values onto the stack, above the caller's arguments. For example, the surface-level procedure (λ (*x y*) (**begin** (*f*) (*x*) (*g*) (*y*))) compiles to the following bytecode, assuming *f* and *g* respectively reside at offsets 2 and 9 when evaluating the **lam** expression.

```
(lam (val val) (2 9)
  (seq (application (loc-clr 0))
       (application (loc-clr 2))
       (application (loc-clr 1))
       (application (loc-clr 3))))
```

The **lam**'s first component, described in more detail in section 6.8, gives a coarse-grained type annotation for each of the procedure's parameters. The second component lists the offsets of the procedure's free variables.

The machine dynamically allocates a closure record even for a **lam** expression that capture no values. To allow the compiler to avoid this runtime cost, the machine provides the **proc-const** form. A **proc-const** expression denotes a closed procedure; unlike a **lam** expression, it does not close over anything, and it is preallocated when the code is loaded into the machine.

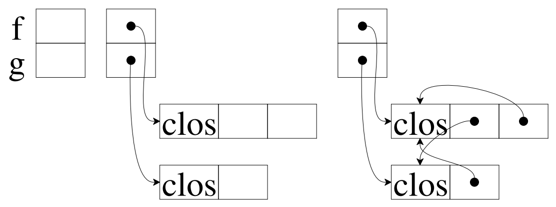
The bytecode **let-rec** form represents a surface-level **letrec** in which the right-hand side of each definition is a λ-expression. For example, consider the following recursive definition.

```
(letrec ([f (λ (x) (begin (f x) (g x)))]
         [g (λ (x) (g x))])
  f)
```

This definition corresponds to the following bytecode.

```
(let-void 2
  (let-rec ((lam (val) (0 1)
                 (seq (application (loc-clr 1) (loc 3))
                      (application (loc 2) (loc 3))))
            (lam (val) (1)
                 (application (loc 1) (loc 2))))
    (loc 0)))
```

This **let-rec** expression heap-allocates an uninitialized closure for each **lam** and writes pointers to these closures in the space pushed by **let-void**. Next, the **let-rec** closes the **lam** expressions—the first captures both closure pointers, while the second captures only itself. Finally, the body of the **let-rec** returns the pointer to the first closure. The following shows the machine's stack and the closure records as the machine evaluates the **let-rec** expression above.



When at least one of the surface-level right-hand sides is not a λ-expression, the MzScheme compiler reverts to boxes to tie the knots. For example, consider the adding the clause [*x* (*f g*)] to the definition above.

```
(letrec ([f (λ (x) (begin (f x) (g x)))]
         [g (λ (x) (g x))]
         [x (f g)])
  f)
```

This is the corresponding bytecode.

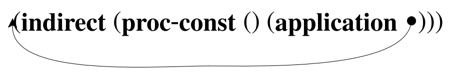
```
(let-void-box 3
  (install-value-box 0
    (lam (val) (0 1)
      (seq (application (loc-box-clr 1) (loc 3))
           (application (loc-box 2) (loc 3))))
    (install-value-box 1
      (lam (val) (1)
        (application (loc-box 1) (loc 2)))
      (install-value-box 2
        (application (loc-box 0) (loc-box-clr 1))
        (loc-box 0)))))
```

The **let-void-box** form is similar to **let-void**, but instead of pushing uninitialized slots, it pushes pointers to fresh boxes initialized with the black hole value **undefined**.

To improve performance, the bytecode language supports one other representation of recursive procedures: cycles in the bytecode itself. Cycles in bytecode are marked by **indirect** expressions; such expressions are the only ones that can be the target of a cycle. For example the procedure

```
(letrec ([loop (λ () (loop))])
  loop)
```

corresponds this cyclic bytecode:



In the grammar of figure 8, the bytecode's cycle is replaced by the expression (**indirect** *x1*), along with the following entry in a separate table of named cycles, described in section 6.2.

```
(x1 (proc-const () 0 (application (indirect x1))))
```

```
                          p ::= (V S H T C) | error

V ::= v | uninit | (box x)        C ::= (i ...)                              l ::= (lam n (n ...) x)
S ::= (u ... s)                   i ::= e                                    v ::= ....
s ::= ε | S                          | (swap n) | (reorder i (e m) ...)         | undefined
u ::= v | uninit | (box x)           | (set n) | (set-box n)                    | (clos x)
H ::= ((x h) ...)                    | (branch e e)                          e ::= ....
h ::= v | ((clos n (u ...) x) ...)   | framepop | framepush                     | (self-app x e0 e1 ...)
T ::= ((x e) ...)                    | (call n) | (self-call x)              m ::= n | ?
```

Figure 9: The grammar for machine states.

The remaining bytecode forms are straightforward. The **branch** and **case-lam** forms represent surface-level **if** and **case-lambda** expressions, **loc-box-clr** and **loc-box-noclr** are the box analogs of **loc-clr** and **loc-noclr**, and the non-terminal w defines bytecode constants.

6.2 Bytecode Loading

The bytecode language evaluated by the MzScheme machine is slightly different than the language produced by the compiler and analyzed by the verifier. This section describes those differences and the loader that transforms the bytecode in preparation for evaluation.

Figure 9 gives the grammar of machine states. This grammar extends the grammar in figure 8, adding a number of non-terminals relevant to the machine states, as well as extending the w and e non-terminals to support closure values and an optimization described in section 6.4.

A machine state p is either an error or a tuple of five components, one for each of the registers in the machine: V, S, H, T, and C. The first four registers are described in the left-hand column of figure 9. The value (V) register holds the result of the most recently evaluated expression. It can be either uninitialized, a value, or a box that refers to some value in the heap. The S register represents the machine's stack. It is essentially a list (of u), but segmented into frames that simplify pushing and popping sequences of values. Like the value register, each position can be either uninitialized, a value, or a box. The H register represents the machine's heap, a table mapping names to values or to closure records. A closure record contains an arity annotation, the values captured by the closure, and a pointer into the machine's text segment T. The text segment holds entries representing bytecode cycles and the bodies of all **lam** and **proc-const** expressions. The C register, shown in the middle column of figure 9, represents the machine's control stack. It consists of a sequence of instructions, i, which are either whole bytecode expressions or one of several tokens that record the work remaining in a partially evaluated expression.

The final column of figure 9 shows how the runtime representation of bytecode differs from the form generated by the compiler and accepted by the verifier. First, bytecode expressions (e) now include a **self-app** form that identifies recursive tail-calls.

MzScheme's JIT compiler optimizes these applications, as described in section 6.4. Second, values (v) now include a **clos** form, representing pointers to closures, and the blackhole value **undefined**. Third, the redefinition of **lam** replaces type annotations with an arity label and replaces the procedure body with a pointer into the text segment.

The load function constructs an initial machine state from an expression e where **indirect** cycles have been rewritten into an initial text segment T.

$$
\begin{array}{l}
\mathsf{load} : e\ T \rightarrow (V\ S\ H\ T\ C) \\
\mathsf{load}[[e, ((x_0\ e_0)\ \ldots)]] = (\mathbf{uninit} \\
\qquad\qquad\qquad\qquad\quad (((\varepsilon))) \\
\qquad\qquad\qquad\qquad\quad \mathsf{concat}[[H, H_0, \ldots]] \\
\qquad\qquad\qquad\qquad\quad \mathsf{concat}[[T, ((x_0\ e_{0*})\ \ldots), T_0, \ldots]] \\
\qquad\qquad\qquad\qquad\quad (e_*)) \\
\text{ where } (e_*\ H\ T) = \mathsf{load^*}[[e, \text{-}]],\ ((e_{0*}\ H_0\ T_0)\ \ldots) = (\mathsf{load^*}[[e_0, \text{-}]]\ \ldots)
\end{array}
$$

The value register begins uninitialized, and the values stack begins with three empty segments. This stack configuration corresponds to the evaluation of the body of a procedure with no arguments, with no values in its closure, and with no local variables pushed by its body. The initial value of the final three registers are built via the load* function, shown in figure 10.

In addition to a bytecode expression e, the load* function accepts an accumulator ϕ that controls when **application** expressions are transformed into **self-app** expressions. This function produces a new bytecode expression suitable for evaluation, as well as initial values for the machine's heap and text segment registers. The initial heap contains statically allocated closures for each **proc-const** in the input, and the text segment contains the (translated) bodies of the **proc-const** and **lam** expressions in the input, as well as the entries that break the expressions cycles.

The first two cases deal with **self-app** expressions. When e is in tail position with respect to a recursive procedure, the ϕ parameter is a triple of two numbers and a variable. The first number is the position in the stack of the procedure's self-pointer, the second number is the arity of the procedure, and the variable is the location in the text segment holding the procedure's body. The ϕ parameter is not a triple when the loader is not transforming a recursive procedure and when e is not in tail position.

Using ϕ, the first case of load* transforms an **application** expression into a **self-app** expression when the arity of the application matches the arity recorded in ϕ, and when the function being called is located at the proper position on the stack. The second case of load* calls load-lam-rec to construct new values of ϕ and recur with the bindings in the **let-rec**.

$$
\begin{array}{l}
\mathsf{load\text{-}lam\text{-}rec} : e\ n \rightarrow e\ H\ T \\
\mathsf{load\text{-}lam\text{-}rec}[[(\mathbf{lam}\ (\tau_0\ \ldots)\ (n_0\ \ldots\ n_i\ n_{i+1}\ \ldots)\ e), n_i]] = \\
((\mathbf{lam}\ n\ (n_0\ \ldots\ n_i\ n_{i+1}\ \ldots)\ x)\ H\ ((x\ e_*)\ (x_0\ e_0)\ \ldots)) \\
\text{ where } n = \#(\tau_0\ \ldots),\ x = \textit{a fresh variable}, \\
\qquad (e_*\ H\ ((x_0\ e_0)\ \ldots)) = \mathsf{load^*}[[e, (\#(n_0\ \ldots)\quad n\ x)]], \\
\qquad n_i \notin \{n_{i+1}, \ldots\} \\
\mathsf{load\text{-}lam\text{-}rec}[[l, n_j]] = \\
\mathsf{load^*}[[l, \text{-}]]
\end{array}
$$

```
load* : e φ → e H T                                  φ ::= - | (n n x)
load*[[(application (loc-noclr n) e_1 ...), (n_p n_a x)]]  = ((self-app x (loc-noclr n) e_1* ...)
                                                              concat[[H_1, ...]]
                                                              concat[[T_1, ...]])
 where  n = n_p + #(e_1 ...),  n_a = #(e_1 ...), ((e_1* H_1 T_1) ...) = (load*[[e_1, -]] ...)
load*[[(let-rec (l_0 ...) e), φ]]                           = ((let-rec (l_0* ...) e_*)
                                                              concat[[H, H_0, ...]]
                                                              concat[[T, T_0, ...]])
 where (e_* H T) = load*[[e, φ]], (n_0 ...) = (0 ... #(l_0 ...) -1),
       ((l_0* H_0 T_0) ...) = (load-lam-rec[[l_0, n_0]] ...)
load*[[(application e_0 e_1 ...), φ]]                       = ((application e_0* ...)
                                                              concat[[H_0, ...]]
                                                              concat[[T_0, ...]])
 where ((e_0* H_0 T_0) ...) = (load*[[e_0, -]] load*[[e_1, -]] ...)
load*[[(let-one e_r e_b), φ]]                               = ((let-one e_r* e_b*) concat[[H_r, H_b]] concat[[T_r, T_b]])
 where (e_r* H_r T_r) = load*[[e_r, -]], (e_b* H_b T_b) = load*[[e_b, φ+[[φ, 1]]]]
load*[[(let-void n e), φ]]                                  = ((let-void n e_*) H T)
 where (e_* H T) = load*[[e, φ+[[φ, n]]]]
load*[[(let-void-box n e), φ]]                              = ((let-void-box n e_*) H T)
 where (e_* H T) = load*[[e, φ+[[φ, n]]]]
load*[[(boxenv n e), φ]]                                    = ((boxenv n e_*) H T)
 where (e_* H T) = load*[[e, φ]]
load*[[(install-value n e_r e_b), φ]]                       = ((install-value n e_r* e_b*)
                                                              concat[[H_r, H_b]]
                                                              concat[[T_r, T_b]])
 where (e_r* H_r T_r) = load*[[e_r, -]], (e_b* H_b T_b) = load*[[e_b, φ]]
load*[[(install-value-box n e_r e_b), φ]]                   = ((install-value-box n e_r* e_b*)
                                                              concat[[H_r, H_b]]
                                                              concat[[T_r, T_b]])
 where (e_r* H_r T_r) = load*[[e_r, -]], (e_b* H_b T_b) = load*[[e_b, φ]]
load*[[(seq e_0 ... e_n), φ]]                               = ((seq e_0* ... e_n*)
                                                              concat[[H_0, ..., H_n]]
                                                              concat[[T_0, ..., T_n]])
 where ((e_0* H_0 T_0) ...) = (load*[[e_0, -]] ...), (e_n* H_n T_n) = load*[[e_n, φ]]
load*[[(branch e_c e_t e_f), φ]]                            = ((branch e_c* e_t* e_f*)
                                                              concat[[H_c, H_t, H_f]]
                                                              concat[[T_c, T_t, T_f]])
 where (e_c* H_c T_c) = load*[[e_c, -]], (e_t* H_t T_t) = load*[[e_t, φ]], (e_f* H_f T_f) = load*[[e_f, φ]]
load*[[(lam (τ_0 ...) (n_0 ...) e), φ]]                     = ((lam n (n_0 ...) x) H ((x e_*) (x_0 e_0) ...))
 where x = a fresh variable, n = #(τ_0 ...), (e_* H ((x_0 e_0) ...)) = load*[[e, -]]
load*[[(proc-const (τ_0 ...) e), φ]]                        = ((clos x)
                                                              ((x ((clos n () x_*))) (x_0 h_0) ...)
                                                              ((x_* e_*) (x_i e_i) ...))
 where x = a fresh variable, x_* = a fresh variable, n = #(τ_0 ...),
       (e_* ((x_0 h_0) ...) ((x_i e_i) ...)) = load*[[e, -]]
load*[[(case-lam l_0 ...), φ]]                              = ((case-lam l_0* ...) concat[[H_0, ...]] concat[[T_0, ...]])
 where ((l_0* H_0 T_0) ...) = (load*[[l_0, φ]] ...)
load*[[e, φ]]                                               = (e () ())
```

Figure 10: Construction of the initial machine state.

The load-lam-rec function accepts an expression from the right-hand side of a **let-rec** and a number, n_i, indicating the position where the function occurs in the **let-rec**. If it is given a **lam** expression whose closure also contains n_i, then the function closes over itself and thus load-lam-rec invokes load* with ϕ as a triple. The second case of load-lam-rec just calls load*, with an empty ϕ.

The remaining cases in load* recursively process the structure of the bytecode, using $\phi+$ to adjust ϕ as the expressions push values onto the stack.

$$
\begin{array}{lcl}
\phi+ : \phi\ n \longrightarrow \phi & & \\
\phi+[[\text{-}, n]] & = & \text{-} \\
\phi+[[(n_p\ n_a\ x), n]] & = & (n + n_p \quad n_a\ x)
\end{array}
$$

Finally, the cases for **lam**, **proc-const**, and **case-lam** move the procedure bodies into the text segment, and the case for **proc-const** also moves its argument into the initial heap. Each of the cases also uses the concat metafunction to combine the heaps and text segments from loading sub-expressions.

6.3 Bytecode Evaluation

The MzScheme machine is given as series of transition rules that dispatch on the first element in the C register. Figure 11 gives the machine transitions related to stack references. The [loc] rule copies the value at the given stack offset into the machine's value register, via the stack-ref metafunction, shown at the bottom of figure 11. Note that the stack-ref metafunction only returns v and (**box** x); if the relevant position on the stack holds **uninit**, then stack-ref is undefined and the machine is stuck.

The [loc-noclr] rule is just like the [loc] rule, replacing the value register with the corresponding stack position. The [loc-clr] rule moves the value out of the stack into the value register as well, but it also clears the relevant position in the stack to facilitate garbage collection. The [loc-box] rule performs an indirect load, following the pointer at the given offset to retrieve a heap allocated value. The [loc-box-noclr] and [loc-box-clr] rules are similar to [loc-noclr] and [loc-clr] but operate on slots containing boxes.

Figure 12 gives the rules for the stack manipulation instructions that are not bytecode expressions. These instructions are not available to the bytecode programmer because they allow free-form manipulation of the stack; instead, various other instructions reduce to uses of these instructions. The [set] rule sets a location on the stack to the contents of the value register. Similarly, the [set-box] rule sets the contents of a box on the stack to the contents of the value register. The [swap] rule swaps the value register with the contents of a stack slot.

The last two rules in figure 12 push and pop frames on the stack. In each case, the instructions work on frames three at a time, to mimic the stack structure that supports procedure invocation (described in section 6.4). As Steele advocates [68], these instructions are used before and after the evaluation of any non-tail expression, and procedure application always pops the active frame.

The rules in figure 13 change the contents of stack locations. The **install-value** and **install-value-box** instructions both evaluate their first argument and store the result either directly in the stack or into a box on the stack (respectively) and then evaluate

$$(V\ S\ H\ T\ ((\mathbf{loc}\ n)\ i\ \mathbf{...})) \longrightarrow (\textsf{stack-ref}[[n, S]]\ S\ H\ T\ (i\ \mathbf{...})) \qquad \textsf{[loc]}$$

$$(V\ S\ H\ T\ ((\mathbf{loc\text{-}noclr}\ n)\ i\ \mathbf{...})) \longrightarrow (\textsf{stack-ref}[[n, S]]\ S\ H\ T\ (i\ \mathbf{...})) \qquad \textsf{[loc-noclr]}$$

$$\begin{array}{ll} & (V\ S\ H\ T\ ((\mathbf{loc\text{-}clr}\ n)\ i\ \mathbf{...})) \qquad \textsf{[loc-clr]} \\ \longrightarrow & (\textsf{stack-ref}[[n, S]]\ \textsf{stack-set}[[\mathbf{uninit}, n, S]]\ H\ T\ (i\ \mathbf{...})) \end{array}$$

$$\begin{array}{ll} & (V\ S\ H\ T\ ((\mathbf{loc\text{-}box}\ n)\ i\ \mathbf{...})) \qquad \textsf{[loc-box]} \\ \longrightarrow & (\textsf{heap-ref}[[\textsf{stack-ref}[[n, S]], H]]\ S\ H\ T\ (i\ \mathbf{...})) \end{array}$$

$$\begin{array}{ll} & (V\ S\ H\ T\ ((\mathbf{loc\text{-}box\text{-}noclr}\ n)\ i\ \mathbf{...})) \qquad \textsf{[loc-box-noclr]} \\ \longrightarrow & (\textsf{heap-ref}[[\textsf{stack-ref}[[n, S]], H]]\ S\ H\ T\ (i\ \mathbf{...})) \end{array}$$

$$\begin{array}{ll} & (V\ S\ H\ T\ ((\mathbf{loc\text{-}box\text{-}clr}\ n)\ i\ \mathbf{...})) \qquad \textsf{[loc-box-clr]} \\ \longrightarrow & (\textsf{heap-ref}[[\textsf{stack-ref}[[n, S]], H]]\ \textsf{stack-set}[[\mathbf{uninit}, n, S]]\ H\ T\ (i\ \mathbf{...})) \end{array}$$

$\textsf{stack-ref} : n\ S \rightarrow s$

$$\begin{array}{lll} \textsf{stack-ref}[[0, (v\ u\ \mathbf{...}\ s)]] & = v & \\ \textsf{stack-ref}[[0, ((\mathbf{box}\ x)\ u\ \mathbf{...}\ s)]] & = (\mathbf{box}\ x) & \\ \textsf{stack-ref}[[n, (u_0\ u_1\ \mathbf{...}\ s)]] & = \textsf{stack-ref}[[n - 1\ ,\ (u_1\ \mathbf{...}\ s)]] & \text{where } n > 0 \\ \textsf{stack-ref}[[n, ((u\ \mathbf{...}\ s))]] & = \textsf{stack-ref}[[n, (u\ \mathbf{...}\ s)]] & \end{array}$$

$\textsf{stack-set} : u\ n\ S \rightarrow S$

$$\begin{array}{lll} \textsf{stack-set}[[u, n, (u_0\ \mathbf{...}\ u_n\ u_{n+1}\ \mathbf{...}\ s)]] & = (u_0\ \mathbf{...}\ u\ u_{n+1}\ \mathbf{...}\ s) & \text{where } n = \#(u_0\ \mathbf{...}) \\ \textsf{stack-set}[[u, n, (u_0\ \mathbf{...}\ s)]] & = (u_0\ \mathbf{...}\ \textsf{stack-set}[[u, n - \#(u_0\ \mathbf{...})\ ,\ s]]) & \end{array}$$

$\textsf{heap-ref} : H\ x \rightarrow h$

$$\textsf{heap-ref}[[(\mathbf{box}\ x_i), ((x_0\ h_0)\ \mathbf{...}\ (x_i\ h_i)\ (x_{i+1}\ h_{i+1})\ \mathbf{...})]] = h_i$$

$\textsf{heap-set} : H\ x\ h \rightarrow H$

$$\textsf{heap-set}[[h, (\mathbf{box}\ x_i), ((x_0\ h_0)\ \mathbf{...}\ (x_i\ h_i)\ (x_{i+1}\ h_{i+1})\ \mathbf{...})]] = ((x_0\ h_0)\ \mathbf{...}\ (x_i\ h)\ (x_{i+1}\ h_{i+1})\ \mathbf{...})$$

Figure 11: Machine transitions related to looking at the stack.

their bodies. The **boxenv** instruction allocates a new box with the value at the specified stack location and writes a pointer to the box at the same stack location.

Figure 14 shows the rules that allocate more space on the stack. The [let-one] rule pushes an uninitialized slot, evaluates its right-hand side, storing the result in the uninitialized slot, then evaluates its body. The [let-void] rule pushes a fixed number of slots onto the stack, also initializing them with **uninit**. The [let-void-box] rule pushes n slots onto the stack, filling them with boxes initialized to the **undefined** value.

Figure 15 cover the rules for the creation of procedures. The first two close **lam** and **case-lam** expressions appearing in arbitrary contexts, putting new closure records into the heap and copying the contents of captured stack locations into the newly created closures. The [let-rec] rule allocates closures for the **lam** expressions in its first argument, after filling the top of the stack with pointers to the closures.

Figure 16 gives the rules for immediate values, branches, sequences and indirect expressions. Values are moved in the value register. A **branch** expression pushes its test position onto the control stack, followed by a **branch** *instruction* containing the "then" and "else" branches. Once the test positions has been evaluated and its result

$$(V\ S\ H\ T\ ((\mathbf{set}\ n)\ i\ \mathbf{...})) \qquad [\textsf{set}]$$
$$\longrightarrow (V\ \textsf{stack-set}[\![V, n, S]\!]\ H\ T\ (i\ \mathbf{...}))$$

$$(v\ S\ H\ T\ ((\mathbf{set\text{-}box}\ n)\ i\ \mathbf{...})) \qquad [\textsf{set-box}]$$
$$\longrightarrow (v\ S\ \textsf{heap-set}[\![v, \textsf{stack-ref}[\![n, S]\!], H]\!]\ T\ (i\ \mathbf{...}))$$

$$(V\ S\ H\ T\ ((\mathbf{swap}\ n)\ i\ \mathbf{...})) \qquad [\textsf{swap}]$$
$$\longrightarrow (\textsf{stack-ref}[\![n, S]\!]\ \textsf{stack-set}[\![V, n, S]\!]\ H\ T\ (i\ \mathbf{...}))$$

$$(V\ (u_0\ \mathbf{...}\ (u_i\ \mathbf{...}\ (u_j\ \mathbf{...}\ s)))\ H\ T\ (\mathbf{framepop}\ i\ \mathbf{...})) \qquad [\textsf{framepop}]$$
$$\longrightarrow (V\ s\ H\ T\ (i\ \mathbf{...}))$$

$$(V\ S\ H\ T\ (\mathbf{framepush}\ i\ \mathbf{...})) \qquad [\textsf{framepush}]$$
$$\longrightarrow (V\ (((S)))\ H\ T\ (i\ \mathbf{...}))$$

Figure 12: Rules for implicit stack manipulation.

$$(V\ S\ H\ T\ ((\mathbf{install\text{-}value}\ n\ e_r\ e_b)\ i\ \mathbf{...})) \qquad [\textsf{install-value}]$$
$$\longrightarrow (V\ S\ H\ T\ (\mathbf{framepush}\ e_r\ \mathbf{framepop}\ (\mathbf{set}\ n)\ e_b\ i\ \mathbf{...}))$$

$$(V\ S\ H\ T\ ((\mathbf{install\text{-}value\text{-}box}\ n\ e_r\ e_b)\ i\ \mathbf{...})) \qquad [\textsf{install-value-box}]$$
$$\longrightarrow (V\ S\ H\ T\ (\mathbf{framepush}\ e_r\ \mathbf{framepop}\ (\mathbf{set\text{-}box}\ n)\ e_b\ i\ \mathbf{...}))$$

$$(V\ S\ ((x_0\ h_0)\ \mathbf{...})\ T\ ((\mathbf{boxenv}\ n\ e)\ i\ \mathbf{...})) \qquad [\textsf{boxenv}]$$
$$\longrightarrow (V\ \textsf{stack-set}[\![(\mathbf{box}\ x), n, S]\!]\ ((x\ v)\ (x_0\ h_0)\ \mathbf{...})\ T\ (e\ i\ \mathbf{...}))$$
$$\text{where } v = \textsf{stack-ref}[\![n, S]\!],\ x \text{ fresh}$$

Figure 13: Machine transitions related to changing the contents of the stack.

stored in the value register, either the [branch-true] or [branch-false] rule applies, dispatching to the appropriate sub-expression. The [seq-two] and [seq-many] rules handle sequences and the [indirect] rule extracts an expression from the text segment to continue evaluation.

6.4 Bytecode Application

The rules for application expressions are more complex than the previous rules in order to model two of the optimizations in the MzScheme JIT compiler, namely the ability to reorder sub-expressions of an **application** and special support for recursive tail-calls, dubbed self-apps.

To model those optimizations, our machine includes both reduction sequences that do not perform the optimizations (modeling how MzScheme behaves when the interpreter runs) and those that do (modeling how MzScheme behaves when the JIT compiler runs). To properly explain these, we first show how a straightforward application reduces and then discuss how the optimizations change the reduction sequences.

Consider the following sequence of machine states, showing an application of the value at the second position in the stack to the values at the third and fourth positions. Since **application** expressions push temporary space before evaluating their

$(V\ S\ H\ T\ ((\textbf{let-one}\ e_r\ e_b)\ i\ \textbf{...}))$ [let-one]

$\longrightarrow (V\ \text{push-uninit}[[1, S]]\ H\ T\ (\textbf{framepush}\ e_r\ \textbf{framepop}\ (\textbf{set}\ 0)\ e_b\ i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{let-void}\ n\ e)\ i\ \textbf{...}))$ [let-void]

$\longrightarrow (V\ \text{push-uninit}[[n, S]]\ H\ T\ (e\ i\ \textbf{...}))$

$(V\ S\ ((x_0\ h_0)\ \textbf{...})\ T\ ((\textbf{let-void-box}\ n\ e)\ i\ \textbf{...}))$ [let-void-box]

$\longrightarrow (V\ \text{push}[[((\textbf{box}\ x_n)\ \textbf{...}), S]]\ ((x_n\ \textbf{undefined})\ \textbf{...}\ (x_0\ h_0)\ \textbf{...})\ T\ (e\ i\ \textbf{...}))$

where $(x_n\ \textbf{...}) = n$ fresh variables

$\text{push-uninit} : n\ S \rightarrow S$

$\text{push-uninit}[[0, S]] = S$

$\text{push-uninit}[[n, (u\ \textbf{...}\ s)]] = \text{push-uninit}[[n - 1\ ,\ (\textbf{uninit}\ u\ \textbf{...}\ s)]]$

$\text{push} : (s...)\ S \rightarrow S$

$\text{push}[[(u_0\ \textbf{...}), (u_i\ \textbf{...}\ s)]] = (u_0\ \textbf{...}\ u_i\ \textbf{...}\ s)$

Figure 14: Machine transitions related to pushing onto stack.

$(V\ S\ ((x_0\ h_0)\ \textbf{...})\ T\ ((\textbf{lam}\ n\ (n_0\ \textbf{...})\ x_i)\ i\ \textbf{...}))$ [lam]

$\longrightarrow ((\textbf{clos}\ x)\ S\ ((x\ ((\textbf{clos}\ n\ (\text{stack-ref}[[n_0, S]]\ \textbf{...})\ x_i)))\ (x_0\ h_0)\ \textbf{...})\ T\ (i\ \textbf{...}))$

where x fresh

$(V\ S\ ((x_0\ h_0)\ \textbf{...})\ T\ ((\textbf{case-lam}\ (\textbf{lam}\ n\ (n_0\ \textbf{...})\ x_i)\ \textbf{...})\ i\ \textbf{...}))$ [case-lam]

$\longrightarrow ((\textbf{clos}\ x)\ S\ ((x\ ((\textbf{clos}\ n\ (\text{stack-ref}[[n_0, S]]\ \textbf{...})\ x_i)\ \textbf{...}))\ (x_0\ h_0)\ \textbf{...})\ T\ (i\ \textbf{...}))$

where x fresh

$(V\ S\ ((x_0\ h_0)\ \textbf{...})\ T\ ((\textbf{let-rec}\ (l_0\ \textbf{...})\ e)\ i\ \textbf{...}))$ [let-rec]

$\longrightarrow (V\ S_*\ ((x_0\ h_0)\ \textbf{...}\ (x\ ((\textbf{clos}\ n_0\ (\text{stack-ref}[[n_{00}, S_*]]\ \textbf{...})\ y_0)))\ \textbf{...})\ T\ (e\ i\ \textbf{...}))$

where $(n\ \textbf{...}) = (0\ ...\ \#(l_0\ \textbf{...})\ \text{-}1)$, $S_* = \text{stack-set*}[[((\textbf{clos}\ x)\ n), \textbf{...}, S]]$,
$l_0 = (\textbf{lam}\ n_0\ (n_{00}\ \textbf{...})\ y_0)$, $(x\textbf{...})$ fresh

$\text{stack-set*} : (u\ n)\ \ldots\ S \rightarrow S$

$\text{stack-set*}[[S]] = S$

$\text{stack-set*}[[(u_0\ n_0), (u_1\ n_1), \textbf{...}, S]] = \text{stack-set*}[[(u_1\ n_1), \textbf{...}, \text{stack-set}[[u_0, n_0, S]]]]$

Figure 15: Machine transitions for procedure definition.

$$(V\ S\ H\ T\ (v\ i\ \ldots)) \longrightarrow (v\ S\ H\ T\ (i\ \ldots)) \qquad \text{[value]}$$

$$(V\ S\ H\ T\ ((\mathbf{branch}\ e_c\ e_t\ e_f)\ i\ \ldots)) \qquad \text{[branch]}$$
$$\longrightarrow (V\ S\ H\ T\ (\mathbf{framepush}\ e_c\ \mathbf{framepop}\ (\mathbf{branch}\ e_t\ e_f)\ i\ \ldots))$$

$$(v\ S\ H\ T\ ((\mathbf{branch}\ e_t\ e_f)\ i\ \ldots)) \longrightarrow (v\ S\ H\ T\ (e_t\ i\ \ldots)) \qquad \text{[branch-true]}$$
$$\text{where}\ \ v \neq \#\mathrm{f}$$

$$(\#\mathrm{f}\ S\ H\ T\ ((\mathbf{branch}\ e_t\ e_f)\ i\ \ldots)) \longrightarrow (\#\mathrm{f}\ S\ H\ T\ (e_f\ i\ \ldots)) \qquad \text{[branch-false]}$$

$$(V\ S\ H\ T\ ((\mathbf{seq}\ e_1\ e_2\ e_3\ e_4\ \ldots)\ i\ \ldots)) \qquad \text{[seq-many]}$$
$$\longrightarrow (V\ S\ H\ T\ (\mathbf{framepush}\ e_1\ \mathbf{framepop}\ (\mathbf{seq}\ e_2\ e_3\ e_4\ \ldots)\ i\ \ldots))$$

$$(V\ S\ H\ T\ ((\mathbf{seq}\ e_1\ e_2)\ i\ \ldots)) \qquad \text{[seq-two]}$$
$$\longrightarrow (V\ S\ H\ T\ (\mathbf{framepush}\ e_1\ \mathbf{framepop}\ e_2\ i\ \ldots))$$

$$(V\ S\ H\ T\ ((\mathbf{indirect}\ x_i)\ i\ \ldots)) \longrightarrow (V\ S\ H\ T\ (e_i\ i\ \ldots)) \qquad \text{[indirect]}$$
$$\text{where}\ T = ((x_0\ e_0)\ \ldots\ (x_i\ e_i)\ (x_{i+1}\ e_{i+1})\ \ldots)$$

Figure 16: Machine transitions for values, branches, sequences, and indirect expressions

sub-expressions, this expression will apply the closure *f* to the arguments 22 and 33.

```
(uninit                     (uninit
 ((clos f) 22 33 ((ε)))      (uninit uninit (clos f) 22 33 ((ε)))
 ((f ((clos 2 (11) fb))))    ((f ((clos 2 (11) fb))))
 ((fb (loc 0)))              ((fb (loc 0)))
 ((application               ((reorder
   (loc 2)                     (call 2)
   (loc 3)                     ((loc 2) ?)
   (loc 4))))                  ((loc 3) 0)
                               ((loc 4) 1))))
```

```
(uninit
 (uninit uninit (clos f) 22 33 ((ε)))
 ((f ((clos 2 (11) fb))))
 ((fb (loc 0)))
 (framepush (loc 2) framepop
  (set 1)
  framepush (loc 3) framepop
  (set 0)
  framepush (loc 4) framepop
  (swap 1)
  (call 2)))
```

First, the machine pushes two slots on the stack to hold temporary values while evaluating the application's sub-expressions. At the same time, it reduces to an artificial state, **reorder**. The **reorder** state helps to set up the reordering optimization. For this example, we assume no reordering occurs, and so the **reorder** state immediately reduces to a series of instructions that evaluate the function and argument sub-expressions. The instructions to evaluate and record the sub-expressions push and pop the stack around each evaluation, because these sub-expressions are not in tail position. To facilitate reordering, the **reorder** instruction records not only the sub-expressions, but also where the results should end up—either a number, for a stack location, or the token *?*, for the value register. Ultimately, the result of the function expression should end up in the value register, though it may be temporarily stored in the stack while other sub-expressions are evaluated.

```
(33                                   ((clos f)                              ((clos f)
 (22 (clos f) (clos f) 22 33 ((ϵ)))    (22 33 (clos f) 22 33 ((ϵ)))          ((11 (22 33 ϵ)))
 ((f ((clos 2 (11) fb))))              ((f ((clos 2 (11) fb))))              ((f ((clos 2 (11) fb))))
 ((fb (loc 0)))                        ((fb (loc 0)))                        ((fb (loc 0)))
 ((swap 1)                             ((call 2)))                           ((loc 0)))
  (call 2)))
```

After the last sub-expression has been evaluated, its result is in the value register, and the function position's result is in the stack slot originally assigned to the last sub-expression. The **swap** instruction swaps the function and argument values, leaving the closure pointer in the value register. This swap step is shown in the first transition above.

The **call** instruction records the arity of the procedure to be called, to detect arity mismatches. In the second state above, the arity in the **call** instruction matches the arity of the procedure in the value register, and so evaluation continues, by replacing the **call** instruction with the body of the procedure and by updating the stack.

The stack is always maintained as a sequence of three frames. The innermost frame contains the arguments to the current procedure. The next frame contains the unpacked closure for the current procedure. The final frame is scratch space for the procedure body. In this example, since the initial stack was three empty frames, the call replaces those frames with 22 33 for the arguments, 11 for the unpacked closure, and an extra set of parentheses for local scratch space.

6.5 The reordering optimization: an overview

The reordering optimization sometimes moves **loc-noclr** references to the end of an application to avoid extra stack operations. For example, if the function position expression had been (**loc-noclr** 2), then the **reorder** instruction above can also reduce as follows.

```
(uninit
 (uninit uninit (clos f) 22 33 ((ϵ)))
 ((f ((clos 2 (11) fb))))
 ((fb (loc 0)))
 ((reorder
   (call 2)
   ((loc-noclr 2) ?)
   ((loc 3) 0)
   ((loc 4) 1))))
```

```
(uninit
 (uninit uninit (clos f) 22 33 ((ϵ)))
 ((f ((clos 2 (11) fb))))
 ((fb (loc 0)))
 ((reorder
   (call 2)
   ((loc 3) 0)
   ((loc 4) 1)
   ((loc-noclr 2) ?))))
```

```
(uninit
 (uninit uninit (clos f) 22 33 ((ϵ)))
 ((f ((clos 2 (11) fb))))
 ((fb (loc 0)))
 (framepush (loc 3) framepop
  (set 0)
  framepush (loc 4) framepop
  (set 1)
  framepush
  (loc-noclr 2)
  framepop
  (call 2)))
```

The first step in this reduction simply moves the **loc-noclr** operation to the end of the **reorder** expression. Then, the **reorder** operation reduces to a series of pushes and pops to evaluate sub-expressions, as before. This time, however, the final sub-expression is the function position, and so no **swap** instruction is needed before the call.

In general, the **reorder** rule moves **loc-noclr** expressions later in an application expression. This reordering can avoid a **swap** operation, and it also simulates how MzScheme's JIT can achieve similar improvements for known primitives, such as addition. Consequently, the reduction graphs for application expressions often look like the one in figure 17. The figure shows the reduction graph for an example like the one above, but where all of the sub-expressions of the application expression are **loc-noclr** expressions instead of **loc** expressions. To save space, only the name of the first instruction in the control register is shown. Overall, the graph begins with a nest of reordering reductions that move the sub-expressions of the application expression into all possible orderings. After an order is chosen, different reductions proceed in lock-step until all sub-expressions are evaluated, at which point some of the traces perform **swap** instructions and some do not. Eventually, all reductions converge to the same **call** state.

6.6 The self-app optimization: an overview

As discussed in section 6.2, some application expressions are transformed into **self-app** expressions by the loader. In short, recursive calls in tail-position are rewritten into **self-app** expressions. Evaluation of the call can then assume that the closure record is already unpacked on the stack, allowing it to skip this step of procedure call setup.

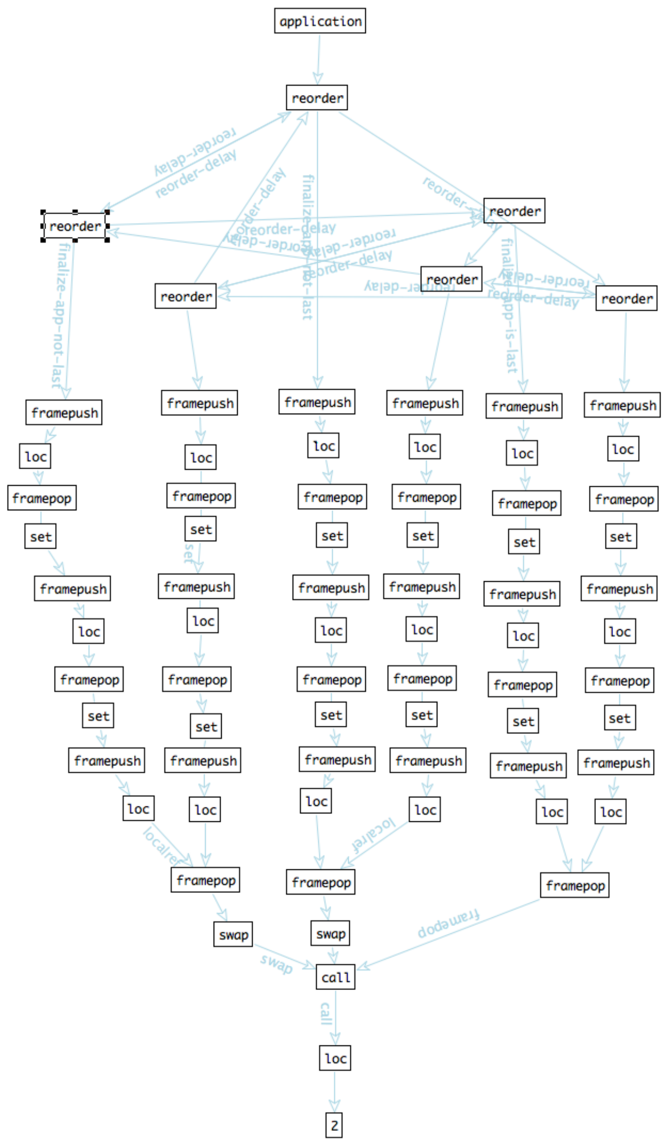


Figure 17: Reordering optimization reduction graph

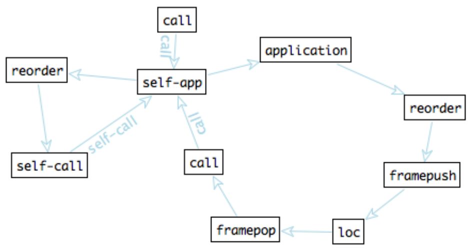


Figure 18: Self-app optimization reduction graph

For example, the Scheme function to the left below corresponds to the bytecode expression in the middle. The loader converts the middle bytecode to produce the bytecode on the right, replacing the application in the body of *f* with a **self-app** that points directly to *fb*.

```
(letrec ((f (λ ()        (let-void 1
               (f))))       (let-rec ((lam () (0)
  (f))                                     (application (loc-noclr 0))))
                                  (application (loc-noclr 0))))
```

```
(uninit
 (((ε)))
 ()
 ((fb (self-app fb (loc-noclr 0))))
 ((let-void 1
    (let-rec ((lam 0 (0) fb))
      (application (loc-noclr 0))))))
```

Evaluation of a **self-app** expression begins as an ordinary **application**, but it immediately discards the expression in function position, because its result is already known. Then, instead of reducing to a **call** instruction, the **reorder** state reduces to a **self-call** instruction that retains the pointer to the body of the procedure. When control eventually reaches this **self-call**, the machine pops the active invocation's temporary space, installs the new arguments, and jumps to the position recorded in the **self-call** instruction.

Figure 18 shows, in graph form, the two reduction sequences for the **self-app** above. The longer cycle shows the instructions that the ordinary application executes. The shorter cycle shows the instructions that the self application executes.

$(V\ S\ H\ T\ ((\textbf{application}\ e_0\ e_1\ \textbf{...})\ i\ \textbf{...}))$ [application]

$\longrightarrow (V\ \textsf{push-uninit}[[n, S]]\ H\ T\ ((\textbf{reorder}\ (\textbf{call}\ n)\ (e_0\ \textbf{?})\ (e_1\ n_1)\ \textbf{...})\ i\ \textbf{...}))$

where $n = \#(e_1\ \textbf{...})$, $(n_1\ \textbf{...}) = (0\ ...\ n\text{-}1)$

$(V\ S\ H\ T\ ((\textbf{self-app}\ x\ e_0\ e_1\ \textbf{...})\ i\ \textbf{...})) \longrightarrow (V\ S\ H\ T\ ((\textbf{application}\ e_0\ e_1\ \textbf{...})\ i\ \textbf{...}))$ [self-app]

$(V\ S\ H\ T\ ((\textbf{self-app}\ x\ e_0\ e_1\ \textbf{...})\ i\ \textbf{...}))$ [self-app-opt]

$\longrightarrow (V\ \textsf{push-uninit}[[n, S]]\ H\ T\ ((\textbf{reorder}\ (\textbf{self-call}\ x)\ (e_1\ n_1)\ \textbf{...})\ i\ \textbf{...}))$

where $n = \#(e_1\ \textbf{...})$, $(n_1\ \textbf{...}) = (0\ ...\ n\text{-}1)$

$(V\ S\ H\ T\ ((\textbf{reorder}\ i_r\ (e_0\ m_1)\ \textbf{...}\ ((\textbf{loc-noclr}\ n)\ m_i)\ (e_{i+1}\ m_{i+1})\ (e_{i+2}\ m_{i+2})\ \textbf{...})\ i\ \textbf{...}))$ [reorder]

$\longrightarrow (V\ S\ H\ T\ ((\textbf{reorder}\ i_r\ (e_0\ m_1)\ \textbf{...}\ (e_{i+1}\ m_{i+1})\ (e_{i+2}\ m_{i+2})\ \textbf{...}\ ((\textbf{loc-noclr}\ n)\ m_i))\ i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{reorder}\ (\textbf{call}\ n)\ (e_0\ n_0)\ \textbf{...}\ (e_i\ \textbf{?})\ (e_{i+1}\ n_{i+1})\ \textbf{...}\ (e_j\ n_j))\ i\ \textbf{...}))$ [finalize-app-not-last]

$\longrightarrow (V\ S\ H\ T\ (\textsf{flatten}[[((\textbf{framepush}\ e_0\ \textbf{framepop}\ (\textbf{set}\ n_0))\ \textbf{...})]]$

$\quad \textbf{framepush}\ e_i\ \textbf{framepop}\ (\textbf{set}\ n_j)$

$\quad \textsf{flatten}[[((\textbf{framepush}\ e_{i+1}\ \textbf{framepop}\ (\textbf{set}\ n_{i+1}))\ \textbf{...})]]$

$\quad \textbf{framepush}\ e_j\ \textbf{framepop}$

$\quad (\textbf{swap}\ n_j)\ (\textbf{call}\ n)\ i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{reorder}\ (\textbf{call}\ n)\ (e_0\ n_0)\ \textbf{...}\ (e_n\ \textbf{?}))\ i\ \textbf{...}))$ [finalize-app-is-last]

$\longrightarrow (V\ S\ H\ T\ (\textsf{flatten}[[((\textbf{framepush}\ e_0\ \textbf{framepop}\ (\textbf{set}\ n_0))\ \textbf{...})]]$

$\quad \textbf{framepush}\ e_n\ \textbf{framepop}\ (\textbf{call}\ n)\ i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{reorder}\ (\textbf{self-call}\ x)\ (e_0\ n_0)\ \textbf{...})\ i\ \textbf{...}))$ [finalize-self-app]

$\longrightarrow (V\ S\ H\ T\ (\textsf{flatten}[[((\textbf{framepush}\ e_0\ \textbf{framepop}\ (\textbf{set}\ n_0))\ \textbf{...})]]$

$\quad (\textbf{self-call}\ x)\ i\ \textbf{...}))$

$(V\ (u_0\ \textbf{...}\ u_i\ \textbf{...}\ (u_j\ \textbf{...}\ (u_k\ \textbf{...}\ s)))\ H\ T\ ((\textbf{self-call}\ x_i)\ i\ \textbf{...}))$ [self-call]

$\longrightarrow (V\ ((u_j\ \textbf{...}\ (u_0\ \textbf{...}\ s)))\ H\ T\ (e_i\ i\ \textbf{...}))$

where $\#(u_0\ \textbf{...}) = \#(u_k\ \textbf{...})$, $T = ((x_0\ e_0)\ \textbf{...}\ (x_i\ e_i)\ (x_{i+1}\ e_{i+1})\ \textbf{...})$

$((\textbf{clos}\ x_i)\ (u_1\ \textbf{...}\ u_{n+1}\ \textbf{...}\ (u_m\ \textbf{...}\ (u_k\ \textbf{...}\ s)))\ H\ T\ ((\textbf{call}\ n_i)\ i\ \textbf{...}))$ [call]

$\longrightarrow ((\textbf{clos}\ x_i)\ ((u_i\ \textbf{...}\ (u_1\ \textbf{...}\ s)))\ H\ T\ (e_i\ i\ \textbf{...}))$

where $n_i \notin \{n_0, \textbf{...}\}$, $n_i = \#(u_1\ \textbf{...})$, $H = ((x_0\ h_0)\ \textbf{...}$,

$\quad (x_i\ ((\textbf{clos}\ n_0\ (u_0\ \textbf{...})\ y_0)\ \textbf{...}$

$\quad\quad (\textbf{clos}\ n_i\ (u_i\ \textbf{...})\ y_i)$

$\quad\quad (\textbf{clos}\ n_{i+1}\ (u_{i+1}\ \textbf{...})\ y_{i+1})\ \textbf{...}))$

$\quad (x_{i+1}\ h_{i+1})\ \textbf{...})$

$\quad T = ((y_j\ e_j)\ \textbf{...}\ (y_i\ e_i)\ (y_k\ e_k)\ \textbf{...})$

$(v\ S\ H\ T\ ((\textbf{call}\ n)\ i\ \textbf{...})) \longrightarrow \textbf{error}$ [non-closure]

where $v \neq (\textbf{clos}\ x)$

$((\textbf{clos}\ x_i)$ [app-arity]

S

$((x_0\ h_0)\ \textbf{...}\ (x_i\ ((\textbf{clos}\ n_0\ (u_0\ \textbf{...})\ y_0)\ \textbf{...}))\ (x_{i+1}\ h_{i+1})\ \textbf{...})$

T

$((\textbf{call}\ n)\ i\ \textbf{...})) \longrightarrow \textbf{error}$

where $n \notin \{n_0, \textbf{...}\}$

Figure 19: Machine transitions for procedure application.

6.7 The complete rules

Figure 19 gives the precise rules for procedure application. The [application] rule pushes n temporary slots for an n-ary application and inserts a **reorder** instruction that pairs each sub-expression with the location that should hold its result. The [self-app] rule reduces a **self-app** expression to an ordinary **application** expression, so that both the optimized and the unoptimized reduction sequences are present in the reduction graphs. The [self-app-opt] rule reduces directly to **reorder** with a **self-call** instruction.

The [reorder] rule shuffles sub-expressions according to the following principle: if a sub-expression is a **loc-noclr**, then that sub-expression may be evaluated last.

Together, the rules [finalize-app-is-last] and [finalize-app-not-last] terminate **reorder** states reached from **application** expressions. The former applies when the sub-expression in function position will be evaluated last; it schedules the evaluation and storage of each sub-expression and, finally, a **call** instruction. The latter applies in all other cases; it schedules a **swap** instruction before the **call** but after the evaluation and storage of the sub-expressions, to move the result of the function position into the value register and the most recent result to its assigned stack position. The [finalize-self-app] rule handles self-calls, which never require a **swap**, since self-calls do not need to evaluate the application's function position. All three rules use the flatten metafunction, which takes a sequence of sequences of instructions and flattens them into a surrounding instruction sequence.

The [call] rule handles a call to a procedure with the correct arity, updating the stack and replacing itself with the body of the procedure. The [self-call] adjusts the stack similarly, but leaves the closure portion of the stack intact.

The remaining two rules, [non-closure] and [app-arity], handle the cases when function application receives a non-procedure or a procedure with incorrect arity as its first argument.

6.8 Bytecode Verification

Evaluation of an unconstrained bytecode expression may get stuck in many ways. For example, consider the following expression, which attempts to branch on a box instead of the value inside the box.

```
(let-one #t
  (boxenv 0
    (branch (loc 0) 'yes 'no)))
```

For this expression, the machine eventually reaches the following state.

```
((box x)
 ((box x) ((ε)))
 ((x #t))
 ()
 ((branch 'yes 'no)))
```

Neither the *branch-true* rule nor the *branch-false* rule applies to this state, because (**box** x) is not itself a value, and so the machine is stuck. Similarly, the machine has no transitions for states in which the program mistakes a value for a box, attempts to read an uninitialized slot, or accesses the stack beyond its current bounds.

The bytecode verifier identifies (and rejects) programs that reach such states. It simulates the program's evaluation though abstract interpretation, maintaining a conservative approximation of the machine's values stack and checking that its approximation satisfies the assumptions implicit in each evaluation step. For the program above, the analysis reveals that the top of the stack contains a box when control reaches the program's **loc** expression; since the (**loc** 0) expression requires a value in that position, the verifier rejects the program.

The verification analysis does not have to be especially general; it must only handle the kind of bytecode that the MzScheme compiler generates. For example, the compiler might generate a **let-void** followed by an **install-value** to create a slot and initialize it, but the compiler will never generate a **let-void** whose corresponding **install-value** is inside a nested **branch**. Thus, to simplify the tracking of abstract values, the verifier can rule out certain patterns that might be valid otherwise.

The MzScheme compiler and JIT rely on support for reordering of stack accesses, as reflected by the **reorder** instruction generated during evaluation, and so promises never to clear a slot must be tracked. That is, the verifier must ensure that stack slots accessed through a **loc-noclr** or **loc-box-noclr** expression are, in fact, never cleared—at least within the region where accesses may be reordered. The verifier implementation exploits the fact that the reordering region never spans different branches of a conditional.

For example, the verifier rejects the following program for violating its promise not to clear the value on top of the stack.

```
(proc-const (val)
  (seq (loc 0) (loc-clr 0)))
```

On the other hand, the verifier accepts a program like the following, in which one branch clears a slot that the other promised not to clear.

```
(proc-const (val val)
  (branch (loc 0) (loc 1) (loc-clr 1)))
```

The abstract value of a stack slot tracks whether the slot is cleared, contains an immediate value, or contains a boxed value. In the latter two cases, the abstract value also tracks a promise that the slot will never be cleared. Only certain transitions are allowed among the abstract states of a given stack slot. The states and allowed transitions are as follows:

- **not**: not directly readable, and the slot cannot change state. This state is used for a temporary slot that the evaluator uses to store application-argument values, and it is also used for a slot that is cleared to enable space safety.

- **uninit**: not directly readable, but a value can be installed to change the slot state to **imm**.

- **imm**: contains an immediate value. The slot can change to **not** if it is cleared, it can change to **box** if a **boxenv** instruction boxes the value, or it can change to **imm-nc** if it is accessed with **loc-noclr**.

- **imm-nc**: contains an immediate value, and the slot cannot change state further.

- **box**: contains a boxed value. The slot can change to **not** if it is cleared, and it can change to **box-nc** if it is accessed with **loc-noclr**.
- **box-nc**: contains a boxed value, and the slot cannot change state further.

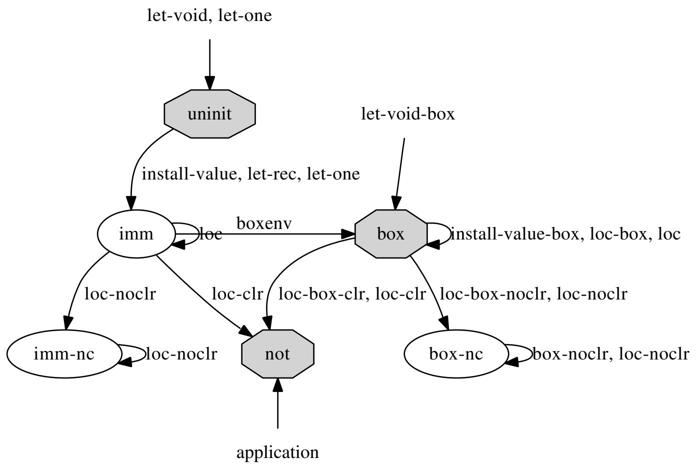


Figure 20: Abstract slot states and transitions.

Figure 20 summarizes these states and transitions. The shared, octagon states are possible initial states, and the labels on a transitions indicate the bytecode forms that can trigger the transition.

When abstract evaluation joins after a branch, effects on the stack from the two branches must be merged and checked for consistency. The **uninit** state is consistent only with itself. The **imm** and **imm-nc** states are consistent with each other, and the **box** and **box-nc** are consistent with each other; the merge operation effectively uses whichever of the two result from the "then" branch. The loss of precise "never cleared" information is acceptable at branch joins, because that information is used for reordering only within a branch. The **not** state is consistent with any state except **uninit**, and the merge operation reduces **not** with any other value to **not**; that is, branches of a conditional can clear different stack slots, but code after the join must assume that any slot cleared by either branch is cleared.

An abstract stack is consumed and produced by the verify function, which is the core of the verification algorithm. The definition of verify is split across figures 22 through 27, while figure 21 gives the function's full type:

- The input *e* is a bytecode expression to verify.
- The input *s* is an abstract stack, which is either a sequence of abstract values, beginning at the top of the stack, or the symbol *invalid*, representing the result of illegal bytecode. The abstract stack is updated during verification of an expression, and the updated stack is the first result of verify.

$$verify : e \times s \times n \times b \times \gamma \times \eta \times f \rightarrow s \times \gamma \times \eta$$

$$\begin{aligned}
s &::= (\hat{u}\ \textbf{...}) \mid \textbf{invalid} \\
\hat{u} &::= \textbf{uninit} \mid \textbf{imm} \mid \textbf{box} \mid \textbf{imm-nc} \mid \textbf{box-nc} \mid \textbf{not} \\
\gamma &::= ((n\ \hat{u})\ \textbf{...}) \\
\eta &::= (n\ \textbf{...}) \\
f &::= (n\ n\ (\hat{u}\ \textbf{...})) \mid \varnothing \\
m &::= n \mid \textbf{?}
\end{aligned}$$

Figure 21: The language of the verifier's abstract interpretation.

- The input n indicates the depth of the current stack that resides within the same branch of the nearest enclosing conditional. This depth is used to track abstract-state changes that must be unwound and merged with the other branch. This depth is also used to rule out certain abstract-state changes (so that they do not have to be tracked and merged).
- The input b indicates whether the expression appears as a non-final sub-expression in a **seq** form, in which case its result will be ignored. This information accommodates a quirk in the MzScheme compiler; in rare cases, the compiler can generate a direct reference to a boxed value within a **seq** sequence. The direct reference normally would be disallowed, but it causes no problem when the result of the reference is ignored.
- The input γ accumulates information about cleared stack slots, so that the clearing operations can be merged at branches. Updated information is returned as the second result of verify.
- The input η accumulates information about "never cleared" annotations on stack slots, so that the annotations can be merged at branches. Updated information is returned as the third result of verify.
- The input f tracks the stack location of a self-reference, so that self tail calls can be checked specially, much like the ϕ parameter in the loader. An empty value indicates that a self-reference is not available or that a call using the self-reference would not be in tail position.

Figure 22 shows the parts of verify's definition that cover stack references. The first three clauses verify **loc** and **loc-box** expressions. The first of these confirms that the target of the **loc** expression is in range and that it contains an immediate value; if it does not, the definition's final catch-all clause (shown later in figure 26) produces **invalid**, causing the verifier to reject the program containing this expression. The second clause accommodates an ignored direct reference to a box within a **seq** form by matching #t for the fourth input. The definition's third clause is the box analog of the first clause.

The next three clauses of verify handle **loc-noclr** and **loc-box-noclr** expressions. Verifying such expressions changes the target slot to **imm-nc** or **box-nc**. Verification also records the "never cleared" annotation in the verify function's η result using the

$\mathsf{verify}[[(\mathbf{loc}\ n), (\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots), n_l, \#\mathrm{f}, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots)\ \gamma\ \eta)$
where $\#(\hat{u}_0 \ldots) = n,\ \hat{u}_n \in \{\mathbf{imm}, \mathbf{imm\text{-}nc}\}$

$\mathsf{verify}[[(\mathbf{loc}\ n), (\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots), n_l, \#\mathrm{t}, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots)\ \gamma\ \eta)$
where $\#(\hat{u}_0 \ldots) = n,\ \hat{u}_n \in \{\mathbf{imm}, \mathbf{imm\text{-}nc}, \mathbf{box}, \mathbf{box\text{-}nc}\}$

$\mathsf{verify}[[(\mathbf{loc\text{-}box}\ n), (\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots), n_l, b, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots)\ \gamma\ \eta)$
where $\#(\hat{u}_0 \ldots) = n,\ \hat{u}_n \in \{\mathbf{box}, \mathbf{box\text{-}nc}\}$

$\mathsf{verify}[[(\mathbf{loc\text{-}noclr}\ n), (\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots), n_l, \#\mathrm{f}, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \mathsf{nc}[[\hat{u}_n]]\ \hat{u}_{n+1} \ldots)\ \gamma\ \mathsf{log\text{-}noclear}[[n, n_l, \eta]])$
where $\#(\hat{u}_0 \ldots) = n,\ \hat{u}_n \in \{\mathbf{imm}, \mathbf{imm\text{-}nc}\}$

$\mathsf{verify}[[(\mathbf{loc\text{-}noclr}\ n), (\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots), n_l, \#\mathrm{t}, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \mathsf{nc}[[\hat{u}_n]]\ \hat{u}_{n+1} \ldots)\ \gamma\ \mathsf{log\text{-}noclear}[[n, n_l, \eta]])$
where $\#(\hat{u}_0 \ldots) = n,\ \hat{u}_n \in \{\mathbf{imm}, \mathbf{imm\text{-}nc}, \mathbf{box}, \mathbf{box\text{-}nc}\}$

$\mathsf{verify}[[(\mathbf{loc\text{-}box\text{-}noclr}\ n), (\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots), n_l, b, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \mathbf{box\text{-}nc}\ \hat{u}_{n+1} \ldots)\ \gamma\ \mathsf{log\text{-}noclear}[[n, n_l, \eta]])$
where $\#(\hat{u}_0 \ldots) = n,\ \hat{u}_n \in \{\mathbf{box}, \mathbf{box\text{-}nc}\}$

$\mathsf{verify}[[(\mathbf{loc\text{-}clr}\ n), (\hat{u}_0 \ldots \mathbf{imm}\ \hat{u}_{n+1} \ldots), n_l, \#\mathrm{f}, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \mathbf{not}\ \hat{u}_{n+1} \ldots)\ \mathsf{log\text{-}clear}[[n, \mathbf{imm}, n_l, \gamma]]\ \eta)$
where $\#(\hat{u}_0 \ldots) = n$

$\mathsf{verify}[[(\mathbf{loc\text{-}clr}\ n), (\hat{u}_0 \ldots \hat{u}_n\ \hat{u}_{n+1} \ldots), n_l, \#\mathrm{t}, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \mathbf{not}\ \hat{u}_{n+1} \ldots)\ \mathsf{log\text{-}clear}[[n, \hat{u}_n, n_l, \gamma]]\ \eta)$
where $\#(\hat{u}_0 \ldots) = n,\ \hat{u}_n \in \{\mathbf{imm}, \mathbf{box}\}$

$\mathsf{verify}[[(\mathbf{loc\text{-}box\text{-}clr}\ n), (\hat{u}_0 \ldots \mathbf{box}\ \hat{u}_{n+1} \ldots), n_l, b, \gamma, \eta, f]] = ((\hat{u}_0 \ldots \mathbf{not}\ \hat{u}_{n+1} \ldots)\ \mathsf{log\text{-}clear}[[n, \mathbf{box}, n_l, \gamma]]\ \eta)$
where $\#(\hat{u}_0 \ldots) = n$

$\mathsf{nc}[[\mathbf{imm}]] = \mathbf{imm\text{-}nc}$
$\mathsf{nc}[[\mathbf{imm\text{-}nc}]] = \mathbf{imm\text{-}nc}$
$\mathsf{nc}[[\mathbf{box}]] = \mathbf{box\text{-}nc}$
$\mathsf{nc}[[\mathbf{box\text{-}nc}]] = \mathbf{box\text{-}nc}$

$\mathsf{log\text{-}noclear}[[n_p, n_l, (n_0 \ldots)]] = (n_p - n_l\ \ n_0 \ldots)$ where $n_p >= n_l$
$\mathsf{log\text{-}noclear}[[n_p, n_l, \eta]] = \eta$

$\mathsf{log\text{-}clear}[[n_p, \hat{u}, n_l, ((n_0\ \hat{u}_0) \ldots)]] = ((n_p - n_l\ \ \hat{u})\ (n_0\ \hat{u}_0) \ldots)$ where $n_p >= n_l$
$\mathsf{log\text{-}clear}[[n_p, \hat{u}, n_l, \gamma]] = \gamma$

Figure 22: The verification rules for variable references

$$\begin{aligned}
&\text{verify}[[(\textbf{branch}\ e_c\ e_t\ e_e), s, n_l, b, \gamma, \eta, f]] = (\text{redo-clears}[[\gamma_3, \text{trim}[[s_3, s]]]]\ \gamma_1\ \eta_3)\\
&\quad \text{where } (s_1\ \gamma_1\ \eta_1) = \text{verify}[[e_c, s, n_l, \#\text{f}, \gamma, \eta, \varnothing]],\\
&\qquad\quad (s_2\ \gamma_2\ \eta_2) = \text{verify}[[e_t, \text{trim}[[s_1, s]], 0, b, (), (), f]],\\
&\qquad\quad (s_3\ \gamma_3\ \eta_3) = \text{verify}[[e_e, \text{undo-noclears}[[\eta_2, \text{undo-clears}[[\gamma_2, \text{trim}[[s_2, s]]]]]], 0, b, \gamma_2, \eta_1, f]]
\end{aligned}$$

$$\begin{aligned}
\text{undo-clears}[[\gamma, \textbf{invalid}]] &= \textbf{invalid}\\
\text{undo-clears}[[(), s]] &= s\\
\text{undo-clears}[[((n_0\ \hat{u}_0)\ (n_1\ \hat{u}_1)\ ...), s]] &= \text{undo-clears}[[((n_1\ \hat{u}_1)\ ...), \text{set}[[\hat{u}_0, n_0, s]]]]
\end{aligned}$$

$$\begin{aligned}
\text{undo-noclears}[[\eta, \textbf{invalid}]] &= \textbf{invalid}\\
\text{undo-noclears}[[(), s]] &= s\\
\text{undo-noclears}[[(n_0\ n_1\ ...), (\hat{u}_0\ ...\ \textbf{imm-nc}\ \hat{u}_i\ ...)]] &= \text{undo-noclears}[[(n_1\ ...), (\hat{u}_0\ ...\ \textbf{imm}\ \hat{u}_i\ ...)]]\\
\text{where } \#(\hat{u}_0\ ...) = n_0 &\\
\text{undo-noclears}[[(n_0\ n_1\ ...), (\hat{u}_0\ ...\ \textbf{box-nc}\ \hat{u}_i\ ...)]] &= \text{undo-noclears}[[(n_1\ ...), (\hat{u}_0\ ...\ \textbf{box}\ \hat{u}_i\ ...)]]\\
\text{where } \#(\hat{u}_0\ ...) = n_0 &\\
\text{undo-noclears}[[(n_0\ n_1\ ...), s]] &= \text{undo-noclears}[[(n_1\ ...), s]]
\end{aligned}$$

$$\begin{aligned}
\text{redo-clears}[[\gamma, \textbf{invalid}]] &= \textbf{invalid}\\
\text{redo-clears}[[(), s]] &= s\\
\text{redo-clears}[[((n_0\ \hat{u}_0)\ (n_1\ \hat{u}_1)\ ...), s]] &= \text{redo-clears}[[((n_1\ \hat{u}_1)\ ...), \text{set}[[\textbf{uninit}, n_0, s]]]]
\end{aligned}$$

$$\text{set}[[\hat{u}, n, (\hat{u}_0\ ...\ \hat{u}_n\ \hat{u}_{n+1}\ ...)]] = (\hat{u}_0\ ...\ \hat{u}\ \hat{u}_{n+1}\ ...) \quad \text{where } \#(\hat{u}_0\ ...) = n$$

Figure 23: The verification rules for branches

log-noclear function, unless the slot is local to the nearest enclosing branch (as indicated by the verify function's n parameter).

The last three clauses of verify in figure 22 handle **loc-clr** and **loc-box-clr** forms. Verification of these forms rejects any attempt to clear a **imm-nc** or **box-nc** slot, and they change a **imm** or **box** slot to **not**. Verification also records the clear operation in the verify function's γ result using the log-clear function—again, only for slots that are not local to the enclosing branch.

The branch clause in figure 23 shows how abstract stacks are "merged," although the verify function does not use an explicit merge operation. Instead, the verifier takes the stack produced by the first branch, truncates it to its original size using trim, reverts clear operations performed by the first branch (as recorded in γ), reverts "never cleared" annotations inserted by the first branch (as recorded in η), and finally feeds the result into verification of the second branch. The abstract stack from the second branch is again trimmed to the original size, and the clear operations from the first branch are re-applied for the result of the entire **branch** form. Verification of **branch** does not have to check consistency of the abstract stack in any other way, because the verifier constrains transitions past the branch-local part of the stack to clearing and adding "never cleared" annotations. [5]

[5]The model could be simplified by independently verifying the two branches of a conditional and then more explicitly merging the abstract stacks. Threading a single stack representation through the "then" and then "else" clauses more closely matches the actual implementation, so that the model can help us detect

$\text{verify}[\![(\mathbf{application}\ e_0\ e_1\ \mathbf{...}),\ s,\ n_l,\ b_i,\ \gamma,\ \eta,\ (n_f\ n_s\ (\hat{u}\ \mathbf{...}))]\!] = \text{verify-self-app}[\![(\mathbf{application}\ e_0\ e_1\ \mathbf{...}),\ s,\ n_l,\ \gamma,\ \eta,\ (n_f\ n_s\ (\hat{u}\ \mathbf{...}))]\!]$
where $e_0 = (\mathbf{loc\text{-}noclr}\ n),\ \ n = n_f + \#(e_1\ \mathbf{...})$
$\text{verify}[\![(\mathbf{application}\ (\mathbf{lam}\ (\tau_0\ \mathbf{...})\ (n_0\ \mathbf{...})\ e)\ e_0\ \mathbf{...}),\ s,\ n_l,\ b,\ \gamma,\ \eta,\ f]\!] = \text{verify*-ref}[\![(e_0\ \mathbf{...}),\ (\tau_0\ \mathbf{...}),\ s_1,\ n_{l*},\ \gamma,\ \eta]\!]$
where $s = (\hat{u}_0\ \mathbf{...}),\ n = \#(e_0\ \mathbf{...}),\ n_{l*} = n + n_l,\ s_1 = \text{abs-push}[\![n,\ \mathbf{not},\ s]\!],\ \text{verify-lam}[\![(\mathbf{lam}\ (\tau_0\ \mathbf{...})\ (n_0\ \mathbf{...})\ e),\ s_1,\ \mathbf{?}]\!]$
$\text{verify}[\![(\mathbf{application}\ (\mathbf{proc\text{-}const}\ (\tau_0\ \mathbf{...})\ e)\ e_0\ \mathbf{...}),\ s,\ n_l,\ b,\ \gamma,\ \eta,\ f]\!] = \text{verify}[\![(\mathbf{application}\ (\mathbf{lam}\ (\tau_0\ \mathbf{...})\ ()\ e)\ e_0\ \mathbf{...}),\ s,\ n_l,\ b,\ \gamma,\ \eta,\ f]\!]$
$\text{verify}[\![(\mathbf{application}\ e_0\ e_1\ \mathbf{...}),\ s,\ n_l,\ b,\ \gamma,\ \eta,\ f]\!] = \text{verify*}[\![(e_0\ e_1\ \mathbf{...}),\ \text{abs-push}[\![n,\ \mathbf{not},\ s]\!],\ n_{l*},\ \#\text{f},\ \gamma,\ \eta]\!]$
where $s = (\hat{u}_0\ \mathbf{...}),\ n = \#(e_1\ \mathbf{...}),\ n_{l*} = n + n_l$

$\text{verify-self-app}[\![(\mathbf{application}\ e_0\ e_1\ \mathbf{...}),\ s,\ n_l,\ \gamma,\ \eta,\ (n_f\ n_s\ (\hat{u}_j\ \mathbf{...}))]\!] = (s_1\ \gamma_1\ \eta_1)$
where $s = (\hat{u}_0\ \mathbf{...}),\ n = \#(e_1\ \mathbf{...}),\ n_{l*} = n + n_l,$
$\quad (s_1\ \gamma_1\ \eta_1) = \text{verify*}[\![(e_0\ e_1\ \mathbf{...}),\ \text{abs-push}[\![n,\ \mathbf{not},\ s]\!],\ n_{l*},\ \#\text{f},\ \gamma,\ \eta]\!],$
$\quad s_1 \neq \mathbf{invalid},\ (n_j\ \mathbf{...}) = (0\ ...\ \#(\hat{u}_j\ \mathbf{...})\ \text{-}1),$
$\quad \text{closure-intact?}[\![(\text{stack-ref}[\![n_j + n_s,\ s_1]\!]\ \mathbf{...}),\ (\hat{u}_j\ \mathbf{...})]\!]$
$\text{verify-self-app}[\![e,\ s,\ n_l,\ \gamma,\ \eta,\ f]\!] = (\mathbf{invalid}\ \gamma\ \eta)$

$\text{verify*}[\![(),\ s,\ n_l,\ b,\ \gamma,\ \eta]\!] = (s\ \gamma\ \eta)$
$\text{verify*}[\![(e_0\ e_1\ \mathbf{...}),\ s,\ n_l,\ b,\ \gamma,\ \eta]\!] = \text{verify*}[\![(e_1\ \mathbf{...}),\ \text{trim}[\![s_1,\ s]\!],\ n_l,\ b,\ \gamma_1,\ \eta_1]\!]$
where $(s_1\ \gamma_1\ \eta_1) = \text{verify}[\![e_0,\ s,\ n_l,\ b,\ \gamma,\ \eta,\ \varnothing]\!]$

$\text{verify*-ref}[\![(),\ (),\ s,\ n_l,\ \gamma,\ \eta]\!] = (s\ \gamma\ \eta)$
$\text{verify*-ref}[\![(e_0\ e_1\ \mathbf{...}),\ (\mathbf{val}\ \tau_1\ \mathbf{...}),\ s,\ n_l,\ \gamma,\ \eta]\!] = \text{verify*-ref}[\![(e_1\ \mathbf{...}),\ (\tau_1\ \mathbf{...}),\ \text{trim}[\![s_1,\ s]\!],\ n_l,\ \gamma_1,\ \eta_1]\!]$
where $(s_1\ \gamma_1\ \eta_1) = \text{verify}[\![e_0,\ s,\ n_l,\ \#\text{f},\ \gamma,\ \eta,\ \varnothing]\!]$
$\text{verify*-ref}[\![(e_0\ e_1\ \mathbf{...}),\ (),\ s,\ n_l,\ \gamma,\ \eta]\!] = \text{verify*}[\![(e_0\ e_1\ \mathbf{...}),\ s,\ n_l,\ \#\text{f},\ \gamma,\ \eta]\!]$
$\text{verify*-ref}[\![(),\ (\tau_0\ \tau_1\ \mathbf{...}),\ s,\ n_l,\ \gamma,\ \eta]\!] = (s\ \gamma\ \eta)$
$\text{verify*-ref}[\![((\mathbf{loc}\ n)\ e_1\ \mathbf{...}),\ (\mathbf{ref}\ \tau_1\ \mathbf{...}),\ s,\ n_l,\ \gamma,\ \eta]\!] = \text{verify*-ref}[\![(e_1\ \mathbf{...}),\ (\tau_1\ \mathbf{...}),\ s_1,\ n_l,\ \gamma_1,\ \eta_1]\!]$
where $(s_1\ \gamma_1\ \eta_1) = \text{verify}[\![(\mathbf{loc\text{-}box}\ n),\ s,\ n_l,\ \#\text{f},\ \gamma,\ \eta,\ \varnothing]\!]$
$\text{verify*-ref}[\![((\mathbf{loc\text{-}noclr}\ n)\ e_1\ \mathbf{...}),\ (\mathbf{ref}\ \tau_1\ \mathbf{...}),\ s,\ n_l,\ \gamma,\ \eta]\!] = \text{verify*-ref}[\![(e_1\ \mathbf{...}),\ (\tau_1\ \mathbf{...}),\ s_1,\ n_l,\ \gamma_1,\ \eta_1]\!]$
where $(s_1\ \gamma_1\ \eta_1) = \text{verify}[\![(\mathbf{loc\text{-}box\text{-}noclr}\ n),\ s,\ n_l,\ \#\text{f},\ \gamma,\ \eta,\ \varnothing]\!]$
$\text{verify*-ref}[\![((\mathbf{loc\text{-}clr}\ n)\ e_1\ \mathbf{...}),\ (\mathbf{ref}\ \tau_1\ \mathbf{...}),\ s,\ n_l,\ \gamma,\ \eta]\!] = \text{verify*-ref}[\![(e_1\ \mathbf{...}),\ (\tau_1\ \mathbf{...}),\ s_1,\ n_l,\ \gamma_1,\ \eta_1]\!]$
where $(s_1\ \gamma_1\ \eta_1) = \text{verify}[\![(\mathbf{loc\text{-}box\text{-}clr}\ n),\ s,\ n_l,\ \#\text{f},\ \gamma,\ \eta,\ \varnothing]\!]$
$\text{verify*-ref}[\![(e\ \mathbf{...}),\ (\tau\ \mathbf{...}),\ s,\ n_l,\ \gamma,\ \eta]\!] = (\mathbf{invalid}\ \gamma\ \eta)$

$\text{stack-ref}[\![n,\ (\hat{u}_0\ \mathbf{...}\ \hat{u}_n\ \hat{u}_{n+1}\ \mathbf{...})]\!] = \hat{u}_n$
where $\#(\hat{u}_0\ \mathbf{...}) = n$

$\text{abs-push}[\![0,\ \hat{u},\ (\hat{u}_0\ \mathbf{...})]\!] = (\hat{u}_0\ \mathbf{...})$
$\text{abs-push}[\![n,\ \hat{u},\ (\hat{u}_0\ \mathbf{...})]\!] = \text{abs-push}[\![n\text{-}1,\ \hat{u},\ (\hat{u}\ \hat{u}_0\ \mathbf{...})]\!]$

$\text{arg}[\![\mathbf{val}]\!] = \mathbf{imm}$
$\text{arg}[\![\mathbf{ref}]\!] = \mathbf{box}$

Figure 24: The verification rules for applications

Figure 24 shows the clauses for verifying procedure applications. The last clause handles the general case, where temporary slots for argument results are created on the stack using abs-push, and the procedure and argument expressions are verified in order (so that abstract effects from earlier expressions are visible to later expressions). Temporary stack slots are set to the **not** state, because they are for internal use in the evaluator; forms like **loc** or **install-value** must never use or change the slots.

The first clause in figure 24 handles the case of self-application tail calls. As in the loader, the self-application rule is triggered by an operator expression that is **loc-noclr**, that uses the stack position indicated by the last parameter to verify, and that applies the expected number of arguments for a self-application. The use of **loc-noclr** for the operation position ensures that the self-reference slot access can be re-ordered with respect to the argument evaluation. In addition, the stack slots containing unpacked closure values must be intact at the point of the self call, so that the implementation of the self application can avoid unpacking the closure. That is, the MzScheme compiler generates a tail self-application using **loc-noclr** only when it also refrains from clearing stack slots that correspond to closure values, and the verifier ensures as much.

The second and third verify clauses in figure 24 cover the case where a procedure accepts boxed arguments. The compiler generates such procedures only when it can eliminate closure allocation by converting captured variables into arguments. In this case, “eliminate allocation” includes allocating the closure only once at the top level (to close over other top-level bindings); since our simplified language does not cover top-level bindings, we model uses of such bindings as an immediately applied **lam** bytecode, which is covered by the second application clause. The third clause shows an immediately applied **proc-const** form, which represents a procedure whose closure allocation is eliminated completely. In either case, argument expressions are checked with verify*-ref, which verifies each expression and checks that it has the type (immediate or boxed) that is expected by the procedure.

Figure 25 shows the verify clauses for a procedure in arbitrary expression positions, in which case the argument types must be immediate values (not boxed). The figure also shows the verify-lam function, which is used for checking all **lam** forms. The verify-lam function checks the body of a procedure in a fresh stack that starts with abstract values for the procedure arguments, and then contains abstract values for captured values. For slots to be captured in the closure, the abstract values must be **imm**, **imm-nc**, **box**, or **box-nc** (not **not** or **uninit**). Those abstract values are copied from the stack where they are captured, but “never cleared” annotations are stripped in the copy, because a application of a closure unpacks values into fresh stack slots that can be cleared independently.

The last argument to verify-lam provides the location in the current stack for the procedure. If the procedure captures that location and then applies the captured value in tail position, then the application counts as a self-application. Most uses of verify-lam supply *?*, which indicates that a self-application slot is not available. Verification of a **let-rec** form supplies a slot number, in which case extract-self extracts information to be used by self-applications within the procedure body.

mistakes in this approach to unwinding and merging stack information.

$\mathsf{verify}[[(\mathbf{lam}\ (\tau\ \mathbf{...})\ (n_0\ \mathbf{...})\ e), s, n_l, b, \gamma, \eta, f]] = (s\ \gamma\ \eta)$
where $\tau = \mathbf{val}$, $\mathsf{verify\text{-}lam}[[(\mathbf{lam}\ (\tau\ \mathbf{...})\ (n_0\ \mathbf{...})\ e), s, \mathbf{?}]]$
$\mathsf{verify}[[(\mathbf{proc\text{-}const}\ (\tau\ \mathbf{...})\ e), s, n_l, b, \gamma, \eta, f]] = \mathsf{verify}[[(\mathbf{lam}\ (\tau\ \mathbf{...})\ ()\ e), s, n_l, b, \gamma, \eta, f]]$
where $\tau = \mathbf{val}$
$\mathsf{verify}[[(\mathbf{case\text{-}lam}\ l\ \mathbf{...}), s, n_l, b, \gamma, \eta, f]] = (s\ \gamma\ \eta)$
where $\mathsf{verify\text{-}lam}[[l, s, \mathbf{?}]]\ , \mathbf{...}$

$\mathsf{verify\text{-}lam}[[(\mathbf{lam}\ (\tau_0\ \mathbf{...})\ (n_0\ \mathbf{...})\ e), s, m]] = s_1 \neq \mathbf{invalid}$
where $s = (\hat{u}_0\ \mathbf{...})$, $n_d = \#s$, $n_{d^*} = \#(\tau_0\ \mathbf{...}) + \#(n_0\ \mathbf{...})$,
$n_0 < n_d\ , \mathbf{...}$,
$\mathsf{stack\text{-}ref}[[n_0, s]] \notin \{\mathbf{uninit}, \mathbf{not}\}\ , \mathbf{...}$,
$(\hat{u}\ \mathbf{...}) = (\mathsf{drop\text{-}noclear}[[\mathsf{stack\text{-}ref}[[n_0, s]]]]\ \mathbf{...})$,
$f = \mathsf{extract\text{-}self}[[m, (n_0\ \mathbf{...}), (\tau_0\ \mathbf{...}), (\hat{u}\ \mathbf{...})]]$,
$(s_1\ \gamma_1\ \eta_1) = \mathsf{verify}[[e, (\hat{u}\ \mathbf{...}\ \mathsf{arg}[[\tau_0]]\ \mathbf{...}), n_{d^*}, \#\mathrm{f}, (), (), f]]$
$\mathsf{verify\text{-}lam}[[\mathbf{any}, s, m]] = \#\mathrm{f}$

$\mathsf{drop\text{-}noclear}[[\mathbf{imm\text{-}nc}]] = \mathbf{imm}$
$\mathsf{drop\text{-}noclear}[[\mathbf{box\text{-}nc}]] = \mathbf{box}$
$\mathsf{drop\text{-}noclear}[[\hat{u}]] = \hat{u}$

$\mathsf{extract\text{-}self}[[\mathbf{?}, (n_0\ \mathbf{...}), (\tau_0\ \mathbf{...}), (\hat{u}_0\ \mathbf{...})]] = \varnothing$
$\mathsf{extract\text{-}self}[[n_i, (n_0\ \mathbf{...}\ n_i\ n_{i+1}\ \mathbf{...}), (\tau_0\ \mathbf{...}), (\hat{u}_0\ \mathbf{...})]] = (\#(n_0\ \mathbf{...})\ \ \#(\tau_0\ \mathbf{...})\ \ (\hat{u}_0\ \mathbf{...}))$
where $n_i \notin \{n_{i+1}, \mathbf{...}\}$
$\mathsf{extract\text{-}self}[[n, (n_0\ \mathbf{...}), (\tau_0\ \mathbf{...}), (\hat{u}_0\ \mathbf{...})]] = \varnothing$

Figure 25: The verification rules for procedures

$\mathsf{verify}[[(\mathbf{let\text{-}one}\ e_r\ e_b), (\hat{u}_1\ \mathbf{...}), n_l, b, \gamma, \eta, f]] = \mathsf{verify}[[e_b, (\mathbf{imm}\ \hat{u}_{1^*}\ \mathbf{...}), n_l{+}1\ ,\ b, \gamma, \eta, \mathsf{shift}[[1, f]]]]$
where $(s_1\ \gamma_1\ \eta_1) = \mathsf{verify}[[e_r, (\mathbf{uninit}\ \hat{u}_1\ \mathbf{...}), n_l{+}1\ ,\ \#\mathrm{f}, \gamma, \eta, \varnothing]]$, $s_1 \neq \mathbf{invalid}$, $(\hat{u}_{1^*}\ \mathbf{...}) = \mathsf{trim}[[s_1, (\hat{u}_1\ \mathbf{...})]]$
$\mathsf{verify}[[(\mathbf{let\text{-}void}\ n\ e), s, n_l, b_i, \gamma, \eta, f]] = \mathsf{verify}[[e, \mathsf{abs\text{-}push}[[n, \mathbf{uninit}, s]], n + n_l\ ,\ b_i, \gamma, \eta, \mathsf{shift}[[n, f]]]]$
where $s = (\hat{u}_0\ \mathbf{...})$
$\mathsf{verify}[[(\mathbf{let\text{-}void\text{-}box}\ n\ e), s, n_l, b_i, \gamma, \eta, f]] = \mathsf{verify}[[e, \mathsf{abs\text{-}push}[[n, \mathbf{box}, s]], n + n_l\ ,\ b_i, \gamma, \eta, \mathsf{shift}[[n, f]]]]$
where $s = (\hat{u}_0\ \mathbf{...})$
$\mathsf{verify}[[(\mathbf{install\text{-}value}\ n\ e_r\ e_b), s, n_l, b, \gamma, \eta, f]] = \mathsf{verify}[[e_b, \mathsf{set}[[\mathbf{imm}, n, s_2]], n_l, b, \gamma, \eta, f]]$
where $n < n_l$, $(s_1\ \gamma_1\ \eta_1) = \mathsf{verify}[[e_r, s, n_l, \#\mathrm{f}, \gamma, \eta, \varnothing]]$, $s_2 = \mathsf{trim}[[s_1, s]]$, $s_2 \neq \mathbf{invalid}$,
$\mathbf{uninit} = \mathsf{stack\text{-}ref}[[n, s_2]]$
$\mathsf{verify}[[(\mathbf{install\text{-}value\text{-}box}\ n\ e_r\ e_b), s, n_l, b, \gamma, \eta, f]] = \mathsf{verify}[[e_b, s_2, n_l, b, \gamma_1, \eta_1, f]]$
where $s = (\hat{u}_0\ \mathbf{...})$, $n < \#s$, $(s_1\ \gamma_1\ \eta_1) = \mathsf{verify}[[e_r, s, n_l, \#\mathrm{f}, \gamma, \eta, \varnothing]]$, $s_2 = \mathsf{trim}[[s_1, s]]$, $s_2 \neq \mathbf{invalid}$,
$\mathsf{stack\text{-}ref}[[n, s_2]] \in \{\mathbf{box}, \mathbf{box\text{-}nc}\}$
$\mathsf{verify}[[(\mathbf{boxenv}\ n_p\ e), (\hat{u}_0\ \mathbf{...}\ \mathbf{imm}\ \hat{u}_{n+1}\ \mathbf{...}), n_l, b, \gamma, \eta, f]] = \mathsf{verify}[[e, (\hat{u}_0\ \mathbf{...}\ \mathbf{box}\ \hat{u}_{n+1}\ \mathbf{...}), n_l, b, \gamma, \eta, f]]$
where $\#(\hat{u}_0\ \mathbf{...}) = n_p$, $n_p < n_l$
$\mathsf{verify}[[(\mathbf{let\text{-}rec}\ (l\ \mathbf{...})\ e), (\hat{u}_0\ \mathbf{...}\ \hat{u}_n\ \mathbf{...}), n_l, b, \gamma, \eta, f]] = \mathsf{verify}[[e, s_1, n_l, b, \gamma, \eta, f]]$
where $l = (\mathbf{lam}\ (v\ \mathbf{...})\ (n_0\ \mathbf{...})\ e_0)$, $v = \mathbf{val}$, $n = \#(l\ \mathbf{...})$, $\#(\hat{u}_0\ \mathbf{...}) = n$, $\hat{u}_0 = \mathbf{uninit}\ , \mathbf{...}$, $n <= n_l$,
$s_1 = \mathsf{abs\text{-}push}[[n, \mathbf{imm}, (\hat{u}_n\ \mathbf{...})]]$, $(n_i\ \mathbf{...}) = (0\ ...\ \#(l\ \mathbf{...})\ {-1})$, $\mathsf{verify\text{-}lam}[[l, s_1, n_i]]\ , \mathbf{...}$

$\mathsf{shift}[[n, \varnothing]] = \varnothing$
$\mathsf{shift}[[n, (n_f\ n_s\ (\hat{u}\ \mathbf{...}))]] = (n + n_f\ \ n + n_s\ \ (\hat{u}\ \mathbf{...}))$

Figure 26: The verification rules for stack operations

$$
\begin{aligned}
&\mathsf{verify}[\![(\mathbf{seq}\ e_0\ \mathbf{...}\ e_n), s, n_l, b, \gamma, \eta, f]\!] &&= \mathsf{verify}[\![e_n, s_1, n_l, b, \gamma_1, \eta_1, f]\!] \\
&\ \text{where } (s_1\ \gamma_1\ \eta_1) = \mathsf{verify^*}[\![(e_0\ \mathbf{...}), s, n_l, \#\mathrm{t}, \gamma, \eta]\!] && \\
&\mathsf{verify}[\![\mathit{number}, s, n_l, b, \gamma, \eta, f]\!] &&= (s\ \gamma\ \eta) \\
&\mathsf{verify}[\![b, s, n_l, b_i, \gamma, \eta, f]\!] &&= (s\ \gamma\ \eta) \\
&\mathsf{verify}[\![\text{'}\mathit{variable}, s, n_l, b, \gamma, \eta, f]\!] &&= (s\ \gamma\ \eta) \\
&\mathsf{verify}[\![\mathbf{void}, s, n_l, b, \gamma, \eta, f]\!] &&= (s\ \gamma\ \eta) \\
&\mathsf{verify}[\![(\mathbf{indirect}\ x), s, n_l, b, \gamma, \eta, f]\!] &&= (s\ \gamma\ \eta) \\
&\mathsf{verify}[\![e, s, n_l, b, \gamma, \eta, f]\!] &&= (\mathbf{invalid}\ \gamma\ \eta)
\end{aligned}
$$

Figure 27: The verification rules for the remaining cases

Verification of **let-rec** and other stack-modifying forms is shown in figure 26. In each of these forms, the final sub-form is in tail position, so self-application information is propagated and updated as necessary using shift. The **let-void** clause simply pushes uninitialized slots into the stack, and **let-void-box** similarly pushes boxes onto the stack. The **install-value** form installs an immediate value into an uninitialized slot, but only if the slot is within the nearest enclosing branch. The **install-value-box** form is similar to **install-value**, but it requires the slot to contain a box already; it does not update the abstract state of the slot, since the run-time effect is just to change the value within the box. The slot does not have to be within the nearest enclosing branch. The **boxenv** form changes the abstract state of a stack slot from **imm** to **box**; again, the slot must be within the nearest enclosing branch. The **let-rec** form is verified in much the same way as **install-value**, but it handles multiple slots. It also calls verify-lam instead of the generic verify, and it supplies a self-application slot for each call to verify-lam.

Figure 26 completes the definition of verify. It covers the simple cases of sequencing and immediate values. An **indirect** form also needs no further work, since the procedure to which it refers is in the process of being verified. The final clause is a catch-all that reports an invalid form when the side conditions of other clauses are not met.

6.9 Verifier Bugs

To assess the bytecode verification algorithm and machine model, we applied PLT Redex's randomized testing framework [36] to check two properties. The first, a safety property, holds if the machine cannot get stuck stuck while evaluating valid bytecode. Formally, this property requires the following.

safety For a bytecode expression e containing the named cycles $((x_0\ e_0)\ \ldots)$, if the verifier accepts e and $(\mathit{load}\ e\ ((x_0\ e_0)\ \ldots)) \rightarrow^* (V\ S\ H\ T\ C)$, then either $C = ()$ (i.e., no instructions remain) or $(V\ S\ H\ T\ C) \rightarrow p$, for some machine state p.

In the machine's production implementation, a stuck state corresponds to a crash or undefined behavior.

The second property, an approximation of confluence, holds if the machine's evaluation rules define at most one result for a valid program. The formal statement follows.

confluence For a bytecode expression e containing the named cycles $((x_0\ e_0)\ \ldots)$, if

Bug #	Description	Discoveries	Attempts	Rate
1	**application** space	458	4 million	1/8,734
2	branch effects (**boxenv**)	2	4 million	1/2,000,000
3	branch effects (**let-rec**)	1	25 million	1/25,000,000
4	**case-lam** ignored	15074	4 million	1/265
5	closure capture	69391	4 million	1/58
6	buffer overflow	44930	4 million	1/89
7	unrestricted update	0	$>$ 45 million	—

Figure 28: The rates (discoveries per attempts) at which randomized testing finds the known verifier bugs.

> $(load\ e\ ((x_0\ e_0) \ldots)) \rightarrow^* (V\ S\ H\ T\ ())$ and $(load\ e\ ((x_0\ e_0) \ldots)) \rightarrow^* (V^{'}\ S^{'}\ H^{'}\ T^{'}\ ())$, then $V = V^{'}$.

This approximation admits evaluation rules that allow divergent computation when a result exists, but detecting non-termination is undecidable in general, preventing us from testing the stronger property.

Testing these properties revealed six bugs in MzScheme's production bytecode verifier but failed to discover a pervasive flaw in the verification algorithm. Figure 28 shows our detection rates for each bug, i.e., the ratio of tests attempted to revealing instances generated. The rates for bugs 1 – 4 correspond to a test case generator that ensures that no stack offset exceeds the depth of the stack but makes no other effort to produce bytecode that passes the verifier. For example, this generator does not ensure that the slot referenced by a **loc-box** expression will actually contain a box. The rates for bugs 5 and 6 correspond to an even simpler generator that completely ignores the verifier's invariants, allowing it to produce any syntactically valid bytecode expression.

Bug #1 The verification rules in figure 24 push the abstract value **not** to reserve slots for the results of the **application**'s sub-expressions. No expression may read or write these slots, preventing a program from observing or disrupting the implementation's placement of intermediate results. The original verification algorithm, however, did not distinguish slots reserved by **application** from any other uninitialized slots, for example allowing the following expression to borrow a reserved slot.

```
(application (install-value 0 (proc-const (val) (loc 0))
                            (loc 0))
             'x)
```

This expression violates neither safety nor confluence, but other expressions that reference **application**-reserved slots do. For example, the following expression produces 'y if the machine evaluates the **application**'s sub-expressions in-order but 'x if it chooses to delay the **loc-noclr** in function position.

```
(let-one (proc-const (val) (loc 0))
  (application
    (loc-noclr 1)
    (install-value 1 (proc-const (val) 'x)
      'y)))
```

Violating safety is no more difficult. For example, the following expression overwrites the result of the **proc-const** expression with a box, causing the application to get stuck at the **call** step.

```
(application (proc-const (val) (loc 0))
             (boxenv 0 'x))
```

Conversely, the machine's implicit stores to the slots it reserves may overwrite what the program explicitly placed in those slots, as in the following expression.

```
(application
  (proc-const (val val) (loc 0 #f))
  (install-value 0 'x (boxenv 0 'y))
  (loc-box 0 #f))
```

This expression gets stuck at the **loc-box** expression, when the machine finds 'y in the target slot. The verifier allowed this **loc-box** because its analysis ignores the machine's implicit store to offset 0, leaving the slot with the box installed by the first argument.

Bugs #2 & #3 The verification of **branch** expressions reverts the clears and no-clears applied in one branch before proceeding with the other, preventing these effects from restricting the code in the second branch. After completing the second branch, the verification algorithm re-applies the first branch's clears and no-clears, merging the branches' effects. The algorithm makes no effort to revert and re-apply the installation of immediate values or boxes because none should occur in the portion of the stack that survives the branch, identified by the verify function's n parameter. The original **boxenv** clause, however, ignored the restriction on slots beyond n, allowing expressions like the following, in which the second branch relies on the effects of the first branch.

```
(let-one 'x
  (branch #f (boxenv 0 'y) (loc-box 0 #f)))
```

Similarly, this bug admits the following expression, in which the expression that follows the **branch** relies an effect that occurs in only one of the branch paths.

```
(let-one 'x
  (seq (branch #f (boxenv 0 'y) 'z)
       (loc-box 0 #f)))
```

The original **let-rec** clause also failed to enforce the restriction on slots beyond n, allowing unsafe expressions like the following.

```
(let-void 1
  (branch #f
    (let-rec ((lam () (0) 'x)) 'y)
    (loc 0 #f)))
```

Bug #4 The original verifier neglected to check the bodies of **case-lam** expressions. Reassuringly, randomized testing discovered this omission immediately.

Bug #5 An off-by-one error in the original verifier allowed **lam** and **case-lam** expressions to capture the first slot beyond the bottom of the active frame, as in the following expression.

```
(proc-const ()
  (lam () (0) (loc 0 #f)))
```

This procedure's safety depends on the context in which it is applied. For example, the machine evaluates the following expression to completion.

```
(application
  (proc-const (val)
    (application (application (loc 0 #f))))
  (proc-const () (lam () (0) (loc 0 #f))))
```

On the other hand, the machine gets stuck on this expression, when the **loc** attempts to captured the uninitialized slot pushed by **let-one**.

```
(application
  (proc-const (val)
    (let-one (application (application (loc 0 #f)))
      'x))
  (proc-const () (lam () (0) (loc 0 #f))))
```

Besides threatening safety, this bug allows bytecode to distinguish expressions that should be equivalent, e.g.,

```
(proc-const (val)
  (application (application (loc 0 #f))))
```

and

```
(proc-const (val)
  (let-one 'q
    (application (application (loc 1 #f)))))
```

Bug #6 The grammar in figure 8 simplifies the syntax of **lam** and **proc-const** expression. In practice, these forms contain an upper bound on the number of stack slots pushed by an application of the procedure (not including pushes by the procedures it calls). The machine's implementation uses this bound to stop a program before its stack grows into the adjacent memory region. In addition to validating this bound, the production verifier uses it to allocate the entire abstract interpretation stack up-front (instead of shrinking and growing it incrementally, as in section 6.8). An early version of the verification algorithm we tested mimicked the production implementation in this regard, and our testing found an off-by-one error in the handling of this bound that could cause the verifier to accept invalid bytecode.

Bug #7 (Not Found) To provide optimization opportunities, the verification rules in section 6.8 restrict a program's ability to change the contents of a slot that already contains an immediate value or a box pointer. In particular, only the **loc-clr**, **loc-box-clr**, and **boxenv** forms change such slots. These forms do not prevent the JIT from delaying a **loc-noclr** because the verification rules reject them when the target slot holds **imm-nc**, as it does after a **loc-noclr**. Similarly, they do not prevent the JIT from reusing the closure-captured values already on the stack for a **self-app** because verify-self-app (figure 24) does not permit the program to clear or box these values.

In two respects, the original verification algorithm did not sufficiently restrict updates to initialized slots. First, it allowed the **install-value** and **let-rec** forms to overwrite initialized slots, permitting expressions like this one, which produces produces 'y if the **loc-noclr** is delayed but 'x if it is not.

```
(let-one 'x
  (application (proc-const (val val) (loc 0))
               (loc-noclr 2)
               (install-value 2 'y 'z)))
```

The failure to restrict these forms also breaks the JIT's optimization of **self-app** expressions. For example, the following expression produces 'b with the optimization but 'a without it, because only in the optimized execution does the second invocation sees the effect of the **install-value**.

```
(let-one 'a
  (let-void 1
    (let-rec ((lam (val) (0 1)
                   (branch (loc 2)
                   (loc 1)
                   (install-value 1 'b
                     (application (loc-noclr 1) #t)))))
      (application (loc 1) #f))))
```

Second, the original formulation of verify-self-app checked only that the procedure did not clear the slots containing the closure-captured values (and *not* that it did not box them). This allows expressions like the following, in which the procedure's first invocation replaces the contents of such a slot with a box pointer, causing the second invocation to get stuck at the **loc**.

```
(let-one (proc-const () void)
  (let-void 1
    (let-rec ((lam () (0 1)
                   (seq (application (loc 1))
                        (boxenv 1
                          (application (loc-noclr 0))))))
      (application (loc 0)))))
```

7 Related Work

7.1 Randomized Testing

Randomized testing goes back at least as far as 1970, to Hanford's *syntax machine*, a test case generator designed to exercise compilers [32]. Like Redex, the syntax machine generates terms from a BNF grammar, choosing productions at random and recurring on their non-terminals. Hanford does not consider the problem of controlling the size of generated terms (section 4.1), but the syntax machine does provide a mechanism called *dynamic grammar* for enforcing context-sensitive constraints, such as variable binding. Hanford applies the syntax machine only to test relatively shallow properties, e.g., that the parser does not reject syntactically valid programs and that the compiler terminates normally.

Our experience testing operational semantics corroborates Hanford's observation of the strength of randomized testing.

> Although as a writer of test cases, the syntax machine is certainly unintelligent, it is also uninhibited. It can test a [language] processor with many combinations that would not be thought of by a human test case writer.

Redex's discovery that the R^{6}RS grammar should not accept the name *make-cond* as a variable (section 5.3, page 20) provides one particularly striking example—few human testers would consider testing all of the grammar's 61 keywords as variable names.

Much of the early attention to randomized testing, however, does not emphasize this bug-finding ability as an end in itself; rather, its motivation tends to be the interpretion of random tests as a statistically significant random sample of a program's real world reliability [73, 24, 30]. This technique may justify statements of the form, "With confidence p, the program fails at most m times in n runs," *provided* that the random tests are selected according a distribution that models the way the program will be used in practice. Unfortunately, this distribution, known as the program's *operational profile*, may be difficult to predict. For example, using random sampling to measure the reliability of a programming language's type system, as one might attempt with Redex, requires a probability distribution that captures the way programmers use the language in practice. Even if this distribution were known, programmatically generating terms according to it may be difficult.

Investigations in the 1980s and 1990s resume the focus on randomized testing as an error-detection technique, comparing it to partition testing, a model intended to represent systematic techniques that divide the input into classes and force execution of at least one test from each class. Examples of partition testing include, for example, techniques based on notions of control or data flow. Early results show randomized testing to be competitive with partition testing [16, 31], but follow-up work shows that partition testing gains an advantage when the partition is chosen carefully [76, 6]. Recent empirical results are mixed.

Many studies show systematic techniques outperforming randomized ones, especially when the program under test exhibits interesting behavior for only a few inputs, e.g., ones satisfying some complex invariant. Marinov et al. find that a sophisticated implementation of bounded exhaustive testing outperforms randomized testing in mutation tests and with respect to simple coverage metrics, when considering the same number of tests from each method [46]. Cadar et al. find symbolic execution paired with a custom contraint-solver achieves vastly greater coverage than randomized testing over the same number of tests [5]. Ferguson and Korel find that chaining, an approach that uses data flow analysis to guide test generation, achieves greater coverage than randomized testing in runs bounded by execution time, arguably a more apt comparison than test case counts [18]. Visser et al. find that randomized testing leaves much to be desired when testing complex data structures like Fibonacci heaps, which have complex preconditions [74]. Randomized testing in Redex mitigates this somewhat, due to the way programs are typically 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 generate test cases that cover all of the interesting configurations using the #:source keyword. Of course, this does not work in general, due to arbitrary side-conditions on rules. For example, we were unable to automatically generate many tests for the the rule [6applyce][6] in the R^6RS formal semantics, due to its side-condition. Often, though, the rule's precondition can be expressed entirely in Redex's pattern language (without a Scheme-level side-condition), and in such cases, Redex can easily satisfy the rule.

Other studies are more favorable to randomized testing. Pacheco et al. compare randomized testing with model checking in two studies [54]. In the first, they repeat the experiment performed by Visser et al. [74], this time using a more sophisticated randomized technique that incrementally extends previously generated tests according to their results. For example, when this feedback-directed algorithm generates a test that appears heuristically to violate a precondition (e.g., by inducing an exception), the algorithm chooses to avoid generating subsequent tests as extensions of this apparently invalid one. Their study finds this technique to outperform systematic approaches, even for complex data structures. In the second study, they report industrial success in applying this technique to widely used Java and .NET libraries, finding many unknown bugs missed by model checking, which failed to scale to the libraries' size. A related study applies the same feedback-directed approach to another large .NET component, this time finding many bugs missed by tests based on symbolic execution [53]. Feedback-directed randomized generation, perhaps guided by Fischer and Kuchen's notion of declarative data flow [20], may be a valuable supplement to Redex's case-based generation, which can breakdown with Scheme-level side-conditions.

Groce et al. describe another industrial study in which engineers successfully applied randomized testing early in the development cycle, switching to heavyweight model-checking and theorem-proving tools only after requirements and prototype code stabilized [29]. Though the subject of their study is the development of robust file system for use on spacecrafts, their balanced approach to verification mirrors what we have begun to explore in the context of programming language metatheory.

Our work was inspired by QuickCheck [2, 9], 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 [8, 62], Clean [39], and even one for the PLT Scheme's ACL2 integration [55].

7.2 Mechanized Metatheory

Work on mechanized metatheory goes back almost as far as interest in randomized testing. Milner produced mechanically verified proofs of properties of programming

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

languages as early as 1972, using a proof-checker for a λ-calculus translation of Scott's logic for computable functions (LCF) [48]. This proof-checker lead to the Edinburgh LCF system [27], and from there, Cambridge LCF [56], NuPRL [14], and HOL [26], and more recently, Isabelle [57], Twelf [58], and Coq [58].

To date, these proof assistants have been applied to many substantial problems. For example, there are several programming languages with verified models that comprise more than an idealized core. Lee et al. [41] show type safety for a language similar to Standard ML [41], Frujia shows safety for a large subset of C# [22], Nipkow and van Oheimb [51] and Syme [72] show safety for fragments of Java, and Norrish establishes several properties of C [52].

Much other work focuses on the verification of implementations. Beginning with Moore's Pinton compiler [49], there have been several certified compilers, i.e., ones accompanied by formal proofs of some correctness property. These include Dold and Vialard's compiler for a subset of Common Lisp [15], Strecker's [69] and Klein and Nipkow's [38] compilers for subsets of Java, compilers for C-like languages by Strecker [70], Leinenbachk et al. [42], and Leroy [44]. As a more lightweight approach, others have explored proof-carying code [50] and translation validation [60], in which a separate program certifies the compiler's output, one program at a time.

Bytecode verification algorithms have themselves been the subject of formal verification. MzScheme's bytecode verification algorithm resembles the ones typically applied to JVM bytecode, originally due to Gosling and Yellin [28, 77, 45]. These algorithms involve an abstract interpreter that conservatively approximates a defensive VM [12], using forward data-flow analysis to resolve the uncertain control-flow of branches and exceptions. There have been many formalizations of this approach, many of which include machine-checked proofs of their soundness [37, 38, 59, 66, 11]. Other techniques approach verification with the ASM method [65] or cast it as a type inference problem [67, 21] or a model checking problem [3]. Leroy [43] and Hartel and Moreau [33] provide excellent surveys of these approaches and others.

There is some, though much less, work on verifying CIL [17] verification algorithms. Gordon and Syme [25] show type safety for a substantial fragment of CIL using Syme's DECLARE [71] system. Follow-up work focuses on CIL's generic types [78]. More recently, Fruja verified type safety for a nearly-complete CIL formalization [22].

7.3 Testing Metatheory

There are number of other tools designed to test programming language metatheory, but there has been little empirical validation of their techniques on large-scale models. The case studies in sections 5 and 6 explore models that are an order of magnitude larger than the subject of most prior studies.

Berghofer and Nipkow have integrated randomized testing into the Isabelle/HOL proof assistant, with the goal of providing a more cost-effective debugging tool than doomed proof attempts [4]. They validate their approach with two case studies. The first exercises a toy language only slightly more complex than the one in section 3. The second shows that their tool finds a known bug in a formalization of red-black trees [35] based on the implementation provided with SML/NJ.

In addition to the challenge of executing higher-order logic, Berghofer and Nipkow tackle one more problem not present in Redex. Consider the statement of the preservation property in section 5.2. A randomized testing tool that instantiates the free variables p and p' independently is unlikely to find a counterexample, since in addition to finding a state p for which the property fails to hold, it must *guess* the subsequent state p'. Berghofer and Nipkow's tool uses a mode analysis on the relations in the property's statement to identify which free variables to treat as independent inputs and which to treat as dependent outputs of these inputs. A Redex programmer would embed this input-output relationship directly into the test predicate, leaving only the inputs as free variables, as in the predicate corresponding to the preservation property (page 19).

Cheney and Momigliano present a similar tool for αProlog that uses bounded model checking [7]. Their motivation is similar.

> We argue that mechanically verified proof is neither the only, nor always the most appropriate, way of gaining condence in the correctness of a formal system; moreover, it is almost never the most appropriate way to debug such a system.

In addition to validating their approach on a toy language similar to the one in section 3, they report some success with examples that are more substantial but still smaller than the subjects of our studies. In particular, their tool automatically confirms a known limitation of λ^{zap} [75] and, with extra guidance, finds the well-known unsoundness in a model of core ML without the value restriction.

Roberson et al. describe another promising model checking approach to testing type soundness [61]. Their system performs several additional optimizations, but the basic idea is the following. The programmer formulates the type system in a declarative subset of Java, similar to first-order logic. Their tool constructs a propositional logic formula describing all well-typed terms within some size bound and initializes an incremental SAT solver with that formula. Next, the tool selects a satisfying assignment—corresponding to some well-typed term—and tests progress and preservation for that term, noting any parts of the term that the reduction step does not examine. The reduction relation behaves similarly on any term that differs only in these unexamined components, and the tool uses a SAT solver to prove that these similar terms also satisfy progress and preservation. Finally, the tool extends the incremental solver's formula to exclude these similar terms, selects a new satisfying term, and repeats the processes. This process continues until the incremental solver indicates that formula is unsatisfiable, meaning that type soundness holds for all terms within the size bound. Roberson et al. report impressive preliminary results for two extensions to Featherweight Java [34], finding many seeded errors. These models are larger than the ones tested by Berghofer and Nipkow and Cheney and Momigliano but still smaller than the models in our studies.

It would be interesting to apply their approach to the R^{6}RS model, in which the type system is much simpler (a term is well-typed if it is closed), but the grammar and reduction relation is larger, compared to Featherweight Java. It is not clear, however, how to test our MzScheme model using their approach. Because our notion of valid bytecode is defined at the level of expressions, not abstract machine states, we cannot frame safety in terms of the single-step reduction relation. Their technique does not appear suited to our multi-step safety formulation because constructing the initial machine state and iteratively applying the reduction relation leaves less of the input term structure untouched, reducing the applicability of their fundamental search space pruning technique. The finite confluence property poses a similar challenge.

8 Conclusion

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 easily able to find bugs in the R^{6}RS formal semantics, one of the biggest, most well-tested (even community-reviewed), mechanized models of a programming language in existence. With only slightly more work (namely, some configuration of the distribution of terms, to get a greater portion past the verifier), we were able to find several bugs in our formalization of the MzScheme machine, a model nearly as large. In this latter case, however, our technique missed one pervasive bug (and nearly another)—a reminder that testing inevitably leaves plenty for a theorem prover.

References

[1] A. W. Appel. *Compiling with Continuations*. Cambridge University Press, 1992.

[2] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with quviq quickcheck. In *Proceedings of the ACM SIGPLAN workshop on Erlang*, pages 2–10, 2006.

[3] D. Basin, S. Friedrich, and M. Gawkowski. Bytecode verification by model-checking. *Journal of Automated Reasoning*, 30(3–4):399–444, 2003.

[4] 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.

[5] C. Cadar, V. Ganesh, P. M. Pawlowski, D. L. Dill, and D. R. Engler. Exe: automatically generating inputs of death. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 322–335, 2006.

[6] T. Y. Chen and Y. T. Yu. On the relationship between partition and random testing. *IEEE Transactions on Software Engineering*, 20(12):977–980, 1994.

[7] 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.

[8] 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.

[9] 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.

[10] W. D. Clinger. Proper tail recursion and space efficiency. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 174–185, June 1998.

[11] A. Coglio, A. Goldberg, and Z. Qian. Toward a provably-correct implementation of the JVM bytecode verifier. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications*, pages 403–410, 1998.

[12] R. Cohen. The defensive Java virtual machine specification. Technical report, Computational Logic Inc., 1997.

[13] 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.

[14] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nurpl Proof Development System*. Prentice-Hall International, 1986.

[15] A. Dold and V. Vialard. A mechanically verified compiling specification for a Lisp compiler. In *Proceedings of the Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 144–155, 2001.

[16] J. W. Duran and S. C. Ntafos. An evaluation of random testing. *IEEE Transactions on Software Engineering*, 10(4):438–4444, 1984.

[17] ECMA. *Common Language Infrastructure (CLI), Standard ECMA–335*. European Association for Standardizing Information and Communication Systems, 4th edition, 2006.

[18] R. Ferguson and B. Korel. The chaining approach for software test data generation. *ACM Transactions on Software Engineering Methodology*, 5(1):63–86, 1996.

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

[20] S. Fischer and H. Kuchen. Data-flow testing of declarative programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 201–212, 2008.

[21] S. N. Freund and J. C. Mitchell. A type system for the Java bytecoe language and verifier. *Journal of Automated Reasoning*, 30(3–4):271–321, 2003.

[22] N. G. Fruja. *Type Safety of C# and .NET CLR*. PhD thesis, ETH Zürich, 2007.

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

[24] E. Girard and J.-C. Rault. A programming technique for software reliability. In *Proceedings of IEEE Symposium on Computer Software Reliability*, pages 44–50, 1973.

[25] A. D. Gordon and D. Syme. Typing a multi-language intermediate code. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 248–260, 2001.

[26] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[27] M. J. C. Gordon, R. Milner, and C. P. Wadsworth. *Edinburgh LCF: A Mechanized Logic of Computation*. Lecture Notes in Computer Science. Springer-Verlag, 1978.

[28] J. Gosling. Java intermediate bytecodes. In *Proceedings of ACM SIGPLAN Workshop on Intermediate Representations*, pages 111–118, 1995.

[29] 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.

[30] D. Hamlet. Random testing. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, pages 970–978, 1994.

[31] D. Hamlet and R. Taylor. Partition testing does not inspire confidence. *IEEE Transactions on Software Engineering*, 16(12):1402–1411, 1990.

[32] K. Hanford. Automatic generation of test cases. *IBM Systems Journal*, 9(4):244–257, 1970.

[33] P. H. Hartel and L. Moreau. Formalizing the safety of Java, the Java virtual machine, and Java card. *ACM Computing Surveys*, 33(4):517–558, 2001.

[34] A. Igarashi, B. C. Pierce, and P. Wadler. Featherweight java: a minimal core calculus for java and gj. *ACM Transactions on Programming Languages and Systems*, 23(3):396–450, 2001.

[35] A. Kimmig. *Red-black trees of smlnj*. Studienarbeit, Universität Freiburg, January 2004.

[36] C. Klein and R. B. Findler. Randomized testing in PLT Redex. In *Scheme and Functional Programming*, 2009. To appear.

[37] G. Klein. *Verified Java Bytecode Verification*. PhD thesis, Institut für Informatik, Technische Universität München, 2003.

[38] G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Transactions on Programming Languages and Systems*, 28(4):619–695, 2006.

[39] 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.

[40] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 6(4):308–320, 1963.

[41] D. K. Lee, K. Crary, and R. Harper. Toward a mechanized metatheory of standard ml. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 173–184, 2007.

[42] D. Leinenbachk, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler: Code generation and implementation correctnes. In *Proceedings of the International Conference on Software Engineering and Formal Methods*, pages 2–12, 2005.

[43] X. Leroy. Java bytecode verification:algorithms and formalizations. *Journal of Automated Reasoning*, 30(3–4):319–340, 2003.

[44] X. Leroy. Formal verification of a realistic compiler. *Communications of the ACM*, 52(7):107–115, 2009.

[45] T. Linholm and F. Yellin. *The Java Virtual Machine Specification*. The Java Series. Prentice Hall PTR, 2nd edition, 1999.

[46] D. Marinov, A. Andoni, D. Daniliuc, S. Kurshid, and M. Rinard. An evaluation of exhaustive testing for data structures. Technical Report 921, MIT Laboratory for Computer Science, 2003.

[47] 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.

[48] R. Milner. Implementation and application of scott's logic for computer functions. In *Proceedings of ACM conference on Proving assertions about programs*, pages 1–6, 1972.

[49] J. S. Moore. A mechanically verified language implementation. *Journal of Automated Reasoning*, 5(4):461–492, 1989.

[50] G. Necula. Proof-carrying code. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 106–119, 1997.

[51] T. Nipkow and D. von Oheimb. Java light is type-safe—definitely. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 161–170, 1998.

[52] M. Norrish. C formalized in HOL. Technical report, University of Cambridge, 1998.

[53] C. Pacheco, S. K. Lahiri, and T. Ball. Finding errors in .NET with feedback-directed random testing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 87–96, 2008.

[54] C. Pacheco, S. K. Lahiri, M. D. Ernst, and T. Ball. Feedback-directed random test generation. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 75–84, 2007.

[55] 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.

[56] L. C. Paulson. *Logic and Computation: Interactive Proof with Cambridge LCF*. Cambridge University Press, 1987.

[57] L. C. Paulson. *Isabelle: A Generic Theorem Prover*. Lecture Notes in Computer Science. Springer-Verlag, 1994.

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

[59] Z. Qian. A formal specification of Java virtual machine instructions for objects, methods and subrountines. In *Formal Syntax and Semantics of Java*, pages 271–312. Springer-Verlag, 1999.

[60] M. C. Rinard and D. Marinov. Credible compilation with pointers. In *Proceedings of the Workshop on Run-Time Result Verification*, 1999.

[61] 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.

[62] 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.

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

[64] 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.

[65] R. Stärk and J. Schmid. Completeness of a bytecode verifier and a certifying Java-to-JVM compiler. *Journal of Automated Reasoning*, 30(3–4):323–361, 2003.

[66] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.

[67] R. Stata and M. Abadi. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems*, 21(1):90–137, 1999.

[68] G. L. Steele Jr. Debunking the "expensive procedure call" myth; or, Procedure call implementations considered harmful; or, LAMBDA: The ultimate goto. Technical Report 443, MIT Artificial Intelligence Laboratory, 1977. First appeared in the Proceedings of the ACM National Conference (Seattle, October 1977), 153–162.

[69] M. Strecker. Formal verification of a Java compiler in Isabelle. In *Proceedings of the International Conference on Automated Deduction*, pages 63–77, 2002.

[70] M. Strecker. Compiler verification for C0. Technical report, Universite Paul Sabatier, 2005.

[71] D. Syme. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.

[72] D. Syme. Proving Java type soundness. Technical report, University of Cambridge, 2001.

[73] T. A. Thayer, M. Lipow, and E. C. Nelson. *Software Reliability*. North-Holland, 1978.

[74] 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.

[75] D. Walker, L. Mackey, J. Ligatti, G. Reis, and D. I. August. Static typing for a faulty lambda calculus. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming*, pages 38–49, 2006.

[76] E. Weyuker and B. Jeng. Analyzing partition testing strategies. *IEEE Transactions on Software Engineering*, 17(7):703–711, 1991.

[77] F. Yellin. Low level security in Java. In *Proceedings of the International World Wide Web Conference*, pages 369–379, 1995.

[78] D. Yu, A. Kennedy, and D. Syme. Formalization of generics for the .NET Common Language Runtime. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2004.