**Noname manuscript No.**
(will be inserted by the editor)

# The Racket Virtual Machine and Randomized Testing

**Casey Klein*** · **Matthew Flatt** ·
**Robert Bruce Findler**

Received: date / Accepted: date

**Abstract** We present a PLT Redex model of a substantial portion of the Racket virtual machine and bytecode verifier (formerly known as MzScheme), along with lessons learned in developing the model. Inspired by the "warts-and-all" approach of the VLISP project, in which Wand *et al.* produced a verified implementation of Scheme, our model reflects many of the realities of a production system. Our methodology departs from the VLISP project's in its approach to validation; instead of producing a proof of correctness, we explore the use of QuickCheck-style randomized testing, finding it a cheap and effective technique for discovering a variety of errors in the model—from simple typos to more fundamental design mistakes.

**CR Subject Classification** F.3.1 [*Logics and Meanings of Programs*]: Specifying and Verifying and Reasoning about Programs—assertions, invariants, mechanical verification · D.2.4 [*Software Engineering*]: Software / Program Verification—assertion checkers · D.3.1 [*Programming Languages*]: Formal Definitions and Theory

**Keywords** formal models · randomized testing · bytecode verification

* Work partly conducted at the University of Chicago, Chicago, IL

Casey Klein
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, IL 60208
E-mail: clklein@eecs.northwestern.edu

Matthew Flatt
School of Computing
University of Utah
Salt Lake City, UT 84112
E-mail: mflatt@cs.utah.edu

Robert Bruce Findler
Electrical Engineering and Computer Science Department
Northwestern University
Evanston, IL 60208
E-mail: robby@eecs.northwestern.edu

## 1 Introduction

The VLISP project [20], completed in 1992, produced a verified implementation of R$^4$RS Scheme [7]. The effort remains noteworthy today for tackling a real "warts-and-all" language capable of bootstrapping itself. Inspired by their landmark effort, we present a PLT Redex [13] model of a substantial portion of the Racket virtual machine and its bytecode verifier [14] (specifically, the bytecode forms with the most interesting verification issues). The model explains many of the irregularities and special cases that support a large-scale, production system.

The similarity to VLISP ends at our approach to validation; instead of proving the machine's correctness, we settle for testing it, using a combination of hand-written test cases and QuickCheck-style randomized testing [5]. In one sense, our work can be seen as a "B-movie" rendition of VLISP, but there are several reasons why semantics engineers might prefer, at least initially, to test their models. First, testing can be a useful prelude to proof. In all but the simplest of cases, building a model is an iterative process: repeatedly revise the definitions as simpler or more useful ones are found, as proof attempts stall, *etc.* Sometimes it is faster to reach the next iteration with testing (especially automated testing) than with proof. Second, testing a theorem usually requires less manual effort than proving it; consequently, we can afford to test more theorems than we could afford to prove. Third, in some cases, there are insufficient resources or motivations to develop proofs (*e.g.*, the Definition of Standard ML [28], the R$^6$RS formal semantics [33], *etc.*).

We make three contributions: an operational semantics for the Racket machine, an account of the machine's bytecode verification algorithm, and an experience report describing the lessons learned in testing the model. The paper proceeds as follows. Section 2 introduces the Racket bytecode language. Section 3 introduces the Racket machine and gives the mapping from bytecode programs to initial machine states. Sections 4 and 5 present the machine's transitions, and Section 6 addresses bytecode verification. Section 7 describes our experience with randomized testing, Section 8 highlights related work, and Section 9 concludes.

## 2 Bytecode Overview

The Racket bytecode language reflects, in contrast to a careful and informed design, the Racket implementation's history as a pure interpreter hastily converted to a bytecode interpreter. Even the term "bytecode" is somewhat misleading, because the representation is based on a tree of bytecode forms rather than a linear sequence of operations. A bytecode tree mimics the tree structure of the Racket surface language. For example, the surface language's **if** form with three sub-expressions is compiled to a **branch** bytecode form with references to three bytecode sub-forms. At the same time, the bytecode machine has neither explicit variables (*e.g.*, as in the SECD machine [23]) nor programmer-visible registers (*e.g.*, as in the JVM [26]); instead, it use a stack of values that is explicitly manipulated by bytecode forms that install and retrieve values on the stack.

The Racket run-time system includes a JIT compiler that converts bytecode forms to machine code. JIT compilation is optional but rarely disabled, and the JIT compiler is meant to run with minimal overhead. Non-trivial optimizations are implemented by the compiler from Racket source to bytecode, rather than in the JIT compiler. Although the JIT compiler rewrites many stack operations to register operations to improve performance, the generated machine code retains much of the structure of the bytecode. Bytecode also remains the

principal format for storing and loading Racket code, but the format is not inherently safe, so a bytecode verifier checks code as it is loaded into the run-time system to ensure that the bytecode obeys the constraints of the run-time system. Much of this bytecode verification amounts to checking that the stack is used properly, including checks on the consistency of hints that are left by the bytecode compiler for use by the JIT compiler.

Figure 1 gives the grammar for bytecode. 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 three after those update the value at a given offset. The remaining productions correspond to forms in Racket'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))
```

The **begin** form in its body executes a sequence of expressions for effect, returning the value of the final expression. This **begin** expression 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 [2, 8]. 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 the top of the stack, but in general, a variable's relative location may shift as the procedure executes. For example, consider the following Racket 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!** assignment expression, the Racket compiler represents that variable with a heap-allocated cell, called a box. Consider the body of the following procedure for example.

```
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
```

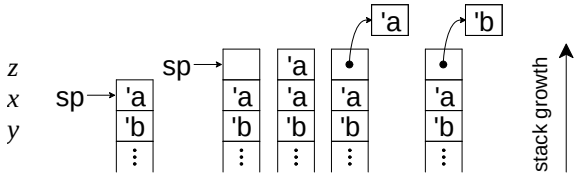
**Fig. 1** The grammar for bytecode expressions `e`.

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

With this representation, the procedure's body corresponds to the following bytecode.

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

The following illustrates the progression of the values stack as the machine evaluates the bytecode above, assuming the procedure's caller supplies 'a for *x* and 'b for *y*.



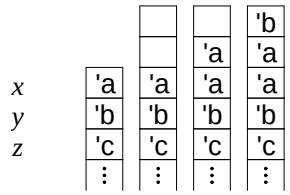

Initially, the active stack frame contains the arguments 'a and 'b. The **let-void** expression pushes one uninitialized slot on the stack. Second, the **install-value** expression initializes that slot with *x*'s value, which is now at offset 1. Third, a **boxenv** expression allocates a fresh box containing 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))

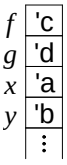
Assuming that the machine processes **application** sub-expressions from left to right, this expression produces two intermediate results, the values fetched by (**loc** 2) and (**loc** 3), and so the machine pushes two uninitialized slots when it begins evaluating the **application**. This push shifts *x*'s offset from 0 to 2, *y*'s offset from 1 to 3, and *z*'s offset from 2 to 4. For example, assuming *x*, *y*, and *z* are respectively 'a, 'b, and 'c, the following shows the stack progression leading to the (**loc** 4) expression.




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, 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. When control reaches the procedure's body, the top of the stack looks as follows, assuming *x*, *y*, *f*, and *g* are respectively 'a, 'b, 'c, and 'd.




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.

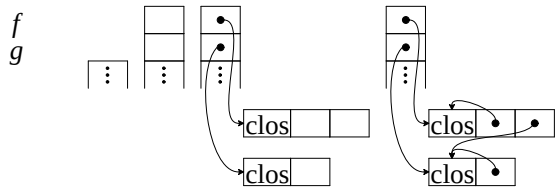
The bytecode **let-rec** form represents a surface-level **letrec** (used for recursive definitions) in which the right-hand side of each definition is a λ-expression. For example, consider the following recursive definition.

```
(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 the enclosing **let-void**. Next, the **let-rec** closes the **lam** expressions—the first, corresponding to surface procedure *f*, captures both closure pointers, while the second captures only itself. The body of the first refers to itself and to the second, respectively, via stack offsets 1 and 2 because their closures are unpacked above their arguments but the enclosing **application** expressions push temporary slots. 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-void** expression above.




When at least one of the surface-level right-hand sides is not a λ-expression[1] , the Racket 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)
```

[1] Such right-hand sides can be used to construct cyclic data structures. For example, the following defines an infinite stream of ones, abstracted over the implementation of *cons*.

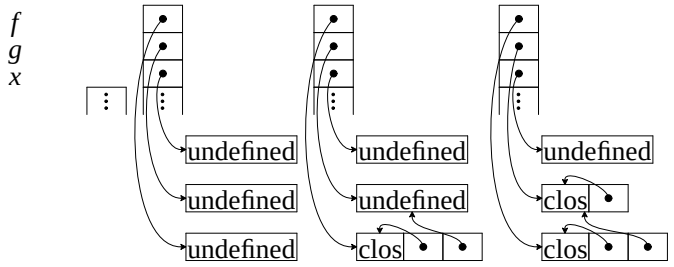
```
(λ (cons)
  (letrec ([ones (cons 1 (λ () ones))])
    ones))
```

This expression compiles to the following bytecode:

The corresponding bytecode then becomes the following.

```
(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**. The following shows the stack progression leading to the evaluation of the definition of *x*.




To improve performance, the bytecode language supports one other representation of recursive procedures: cyclic bytecode. Bytecode expressions need not be trees, and so self-references can be encoded as a cycles in the structure of the expression. For the virtual machine's benefit, such self-references are always wrapped with in the **indirect** form. For example, the procedure

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

can be compiled to bytecode containing the cycle shown below.


The grammar of Figure 1, however, restricts bytecode expressions to be trees, by replacing self-references with pointers into a separate table of named cycles, described in

```
(proc-const (val)
  (let-void-box 1 ; box for 'ones'
    (install-value 0
      (application (loc 3) ; 'cons'
                   1 ; literal 1
                   (lam () (2) ; thunk
                     (loc-box 0)))
      (loc-box 0))))
```

$p$ ::= ($V$ $S$ $H$ $T$ $C$) | **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** $x$ $e_0$ $e_1$ **...**) |
| $T$ ::= (($x$ $e$) **...**) | \| (**call** $n$) \| (**self-call** $x$) | $m$ ::= $n$ \| **?** |

**Fig. 2** The grammar for machine states.

Section 3. For the cyclic expression above, this table would contain an entry such as the following.

($x1$ (**proc-const** () 0 (**application** (**indirect** $x1$))))

The remaining bytecode forms are straightforward. The **case-lam** form represents surface-level **case-lambda** expressions, which denote procedures that dispatch on the number of arguments they receive. The **branch** form represents surface-level **if** expressions, **loc-box-clr** and **loc-box-noclr** are the box analogs of **loc-clr** and **loc-noclr**, and the non-terminal $v$ defines bytecode constants. Constants of form 'variable are symbols, and the constants #t and #f denote booleans.

## 3 Bytecode Loading

This section introduces an abstract machine for evaluating Racket bytecode and explains the loading process that maps a bytecode expression to an initial machine configuration. In addition to initializing the machine's registers, loading makes several small changes to the representation of bytecode expressions.

Figure 2 gives the grammar of machine states. This grammar extends the one in Figure 1, defining several new non-terminals, extending the existing $v$ and $e$ non-terminals, and *redefining* the existing $l$ non-terminal.

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 2. 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 partitioned into segments that simplify pushing and popping contiguous 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 2, 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. Sections 4 and 5 describe these tokens in more detail.

The final column of Figure 2 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. Racket's JIT compiler optimizes these applications, as described in Section 5. 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 $((x_0\ e_0)\ \ldots)$.

load : $e\ T \rightarrow (V\ S\ H\ T\ C)$
load⟦$e$, (($x_0$ (**proc-const** (τ **...**) $e_0$)) **...**)⟧ = (**uninit** ((ε))) $H$ concat⟦(($x_0$ $e_{0*}$) **...**), $T$⟧ ($e_*$))
where (($e_*$ $e_{0*}$ **...**) $H$ $T$ ($y$ **...**)) = load'*⟦(($e$ -) ((**proc-const** (τ **...**) $e_0$) -) **...**), ($x_0$ **...**)⟧

The value register begins uninitialized, and the values stack begins with three empty segments. From outermost to innermost, this stack configuration represents the evaluation of the body of a procedure which has pushed no local variables, which has no values in its closure, and which received no explicit arguments. The side-condition on the initial text segment insists that each **indirect** target be a **proc-const** expression. This restriction simplifies the task of the bytecode verifier (Section 6). The mutually recursive functions load' and load'* (shown in Figure 3) compute the initial values of the machine's other three registers.

In addition to a bytecode expression $e$, the load' function accepts an accumulator $\phi$ that controls when **application** expressions are transformed into **self-app** expressions and a list of the heap and text segment addresses $(x\ \ldots)$ already allocated. This function produces a new bytecode expression suitable for evaluation, as well as initial values for the machine's heap and text segment registers and an updated list of allocated addresses. 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 cycle-breaking entries provided to the initial call to load.

The first two cases deal with **self-app** expressions. When $e$ is in tail position with respect to a recursive procedure, the $\phi$ 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 $\phi$ parameter is **-** when the loader is not transforming a recursive procedure or when $e$ is not in tail position.

Using $\phi$, the first case of load' transforms an **application** expression into a **self-app** expression when (i) the arity of the application matches the arity recorded in $\phi$ and (ii) when the function being called is located at the proper position on the stack. The second case of load' calls load-lam-rec, which constructs appropriate $\phi$ values and recurs on the **lam** expressions.

The load-lam-rec function accepts an expression from the right-hand side of a **let-rec**, a number, $n_i$, indicating the position where the function occurs in the **let-rec**, and a list of already allocated addresses. 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 $\phi$ as a triple; otherwise, load-lam-rec just calls load', with an empty $\phi$.

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

$\phi$+ : $\phi\ n \rightarrow \phi$
$\phi$+⟦-, $n$⟧ = -
$\phi$+⟦($n_p$ $n_a$ $x$), $n$⟧ = ($n + n_p\ n_a\ x$)

load' : $e\ \phi\ (x \ldots) \rightarrow e\ H\ T\ (x \ldots)$ $\qquad \phi ::= \text{-} \mid (n\ n\ x)$

load'⟦(**application** (**loc-noclr** $n$) $e_1$ **...**), $(n_p\ n_a\ x)$, $(y$ **...**)⟧ = ((**self-app** $x$ (**loc-noclr** $n$) $e_{1*}$ **...**) $H\ T$ $(y_*$ **...**))
where $n = n_p + |(e_1$ **...**$)|$, $n_a = |(e_1$ **...**$)|$, $((e_{1*}$ **...**$)\ H\ T\ (y_*$ **...**$))$ = load'*⟦$((e_1$ -**)** ...**)**, $(y$ **...**)⟧

load'⟦(**let-rec** $(l_0$ **...**$)\ e)$, $\phi$, $(y$ **...**)⟧ = ((**let-rec** $(l_{0*}$ **...**$)\ e_*)$ concat⟦$H_0, H_1$⟧ concat⟦$T_0, T_1$⟧ $(y_{**}$ **...**))
where $(e_*\ H_0\ T_0\ (y_*$ **...**$))$ = load'⟦$e, \phi, (y$ **...**)⟧, $(n_0$ **...**$) = (0 \ldots |(l_0$ **...**$)| - 1)$,
$((l_{0*}$ **...**$)\ H_1\ T_1\ (y_{**}$ **...**$))$ = load-lam-rec*⟦$((l_0\ n_0)$ **...**$), (y_*$ **...**)⟧

load'⟦(**application** $e_0\ e_1$ **...**), $\phi$, $(y$ **...**)⟧ = ((**application** $e_{0*}\ e_{1*}$ **...**$)\ H\ T\ (y_*$ **...**$)$)
where $((e_{0*}\ e_{1*}$ **...**$)\ H\ T\ (y_*$ **...**$))$ = load'⟦$((e_0$ -**)** $(e_1$ -**)** ...**)**, $(y$ **...**)⟧

load'⟦(**let-one** $e_r\ e_b$), $\phi$, $(y$ **...**)⟧ = ((**let-one** $e_{r*}\ e_{b*}$) concat⟦$H_r, H_b$⟧ concat⟦$T_r, T_b$⟧ $(y_{**}$ **...**))
where $(e_{r*}\ H_r\ T_r\ (y_*$ **...**$))$ = load'⟦$e_r$, -, $(y$ **...**)⟧, $(e_{b*}\ H_b\ T_b\ (y_{**}$ **...**$))$ = load'⟦$e_b$, $\phi$+⟦$\phi$, 1⟧, $(y_*$ **...**)⟧

load'⟦(**let-void** $n\ e$), $\phi$, $(y$ **...**)⟧ = ((**let-void** $n\ e_*)\ H\ T\ (y_*$ **...**$)$)
where $(e_*\ H\ T\ (y_*$ **...**$))$ = load'⟦$e$, $\phi$+⟦$\phi$, $n$⟧, $(y$ **...**)⟧

load'⟦(**let-void-box** $n\ e$), $\phi$, $(y$ **...**)⟧ = ((**let-void-box** $n\ e_*)\ H\ T\ (y_*$ **...**$)$)
where $(e_*\ H\ T\ (y_*$ **...**$))$ = load'⟦$e$, $\phi$+⟦$\phi$, $n$⟧, $(y$ **...**)⟧

load'⟦(**boxenv** $n\ e$), $\phi$, $(y$ **...**)⟧ = ((**boxenv** $n\ e_*)\ H\ T\ (y_*$ **...**$)$)
where $(e_*\ H\ T\ (y_*$ **...**$))$ = load'⟦$e$, $\phi$, $(y$ **...**)⟧

load'⟦(**install-value** $n\ e_r\ e_b$), $\phi$, $(y$ **...**)⟧ = ((**install-value** $n\ e_{r*}\ e_{b*}$) concat⟦$H_r, H_b$⟧ concat⟦$T_r, T_b$⟧ $(y_{**}$ **...**))
where $(e_{r*}\ H_r\ T_r\ (y_*$ **...**$))$ = load'⟦$e_r$, -, $(y$ **...**)⟧, $(e_{b*}\ H_b\ T_b\ (y_{**}$ **...**$))$ = load'⟦$e_b$, $\phi$, $(y_*$ **...**)⟧

load'⟦(**install-value-box** $n\ e_r\ e_b$), $\phi$, $(y$ **...**)⟧ = ((**install-value-box** $n\ e_{r*}\ e_{b*}$) concat⟦$H_r, H_b$⟧ concat⟦$T_r, T_b$⟧ $(y_{**}$ **...**))
where $(e_{r*}\ H_r\ T_r\ (y_*$ **...**$))$ = load'⟦$e_r$, -, $(y$ **...**)⟧, $(e_{b*}\ H_b\ T_b\ (y_{**}$ **...**$))$ = load'⟦$e_b$, $\phi$, $(y_*$ **...**)⟧

load'⟦(**seq** $e_0$ **...** $e_n$), $\phi$, $(y$ **...**)⟧ = ((**seq** $e_{0*}$ **...** $e_{n*}$) concat⟦$H_0, H_1$⟧ concat⟦$T_0, T_1$⟧ $(y_{**}$ **...**))
where $((e_{0*}$ **...**$)\ H_0\ T_0\ (y_*$ **...**$))$ = load'*⟦$((e_0$ -**)** ...**)**, $(y$ **...**)⟧, $(e_{n*}\ H_1\ T_1\ (y_{**}$ **...**$))$ = load'⟦$e_n$, $\phi$, $(y_*$ **...**)⟧

load'⟦(**branch** $e_c\ e_t\ e_f$), $\phi$, $(y$ **...**)⟧ = ((**branch** $e_{c*}\ e_{t*}\ e_{f*}$) concat⟦$H_c, H_t, H_f$⟧ concat⟦$T_c, T_t, T_f$⟧ $(y_{***}$ **...**))
where $(e_{c*}\ H_c\ T_c\ (y_*$ **...**$))$ = load'⟦$e_c$, -, $(y$ **...**)⟧, $(e_{t*}\ H_t\ T_t\ (y_{**}$ **...**$))$ = load'⟦$e_t$, $\phi$, $(y_*$ **...**)⟧,
$(e_{f*}\ H_f\ T_f\ (y_{***}$ **...**$))$ = load'⟦$e_f$, $\phi$, $(y_{**}$ **...**)⟧

load'⟦(**lam** $(\tau_0$ **...**$)\ (n_0$ **...**$)\ e)$, $\phi$, $(y$ **...**)⟧ = ((**lam** $n\ (n_0$ **...**$)\ x)\ H$ concat⟦$((x\ e_*))$, $T$⟧ $(y_*$ **...**$)$)
where $x =$ a variable not in $(y$ **...**$)$, $n = |(\tau_0$ **...**$)|$, $(e_*\ H\ T\ (y_*$ **...**$))$ = load'⟦$e$, -, $(x\ y$ **...**)⟧

load'⟦(**proc-const** $(\tau_0$ **...**$)\ e)$, $\phi$, $(y$ **...**)⟧ = ((**clos** $x$) concat⟦$((x\ ((\textbf{clos}\ n\ ()\ x_*))), H$⟧ concat⟦$((x_*\ e_*)), T$⟧ $(y_*$ **...**$)$)
where $x =$ a variable not in $(y$ **...**$)$, $x_* =$ a variable not in $(x\ y$ **...**$)$, $n = |(\tau_0$ **...**$)|$,
$(e_*\ H\ T\ (y_*$ **...**$))$ = load'⟦$e$, -, $(x\ x_*\ y$ **...**)⟧

load'⟦(**case-lam** $l_0$ **...**), $\phi$, $(y$ **...**)⟧ = ((**case-lam** $l_{0*}$ **...**$)\ H\ T\ (y_*$ **...**$)$)
where $((l_{0*}$ **...**$)\ H\ T\ (y_*$ **...**$))$ = load'*⟦$((l_0\ \phi)$ **...**$), (y$ **...**)⟧

load'⟦$e$, $\phi$, $(y$ **...**)⟧ = $(e\ ()\ ()\ (y$ **...**$))$

load'* : $((e\ \phi)\ \ldots)\ (x\ \ldots) \rightarrow (e\ \ldots)\ H\ T\ (x\ \ldots)$

load'*⟦(), $(y$ **...**)⟧ = $(()\ ()\ ()\ (y$ **...**$))$

load'*⟦$((e_0\ \phi_0)\ (e_1\ \phi_1)$ **...**$), (y$ **...**)⟧ = ((**...**) concat⟦$H_0, H_1$⟧ concat⟦$T_0, T_1$⟧ $(y_{**}$ **...**$)$)
where $(e_{0*}\ H_0\ T_0\ (y_*$ **...**$))$ = load'⟦$e_0$, $\phi_0$, $(y$ **...**)⟧,
$((e_{1*}$ **...**$)\ H_1\ T_1\ (y_{**}$ **...**$))$ = load'*⟦$((e_1\ \phi_1)$ **...**$), (y_*$ **...**)⟧

**Fig. 3** Construction of the initial machine state.

load-lam-rec : $e\ n\ (x\ \ldots) \to e\ H\ T\ (x\ \ldots)$
load-lam-rec⟦(**lam** ($\tau_0$ **...**) ($n_0$ **...** $n_i$ $n_{i+1}$ **...**) $e$), $n_i$, ($y$ **...**)⟧ =
((**lam** $n$ ($n_0$ **...** $n_i$ $n_{i+1}$ **...**) $x$) $H$ concat⟦(($x$ $e_*$)), $T$⟧ ($y_*$ **...**))
where $n$ = |($\tau_0$ **...**)|, $x$ = a variable not in ($y$ **...**),
($e_*$ $H$ $T$ ($y_*$ **...**)) = load'⟦$e$, (|($n_0$ **...**)| $n$ $x$), ($x$ $y$ **...**)⟧,
$n_i \notin \{n_{i+1}, \ldots\}$
load-lam-rec⟦$l$, $n_j$, ($y$ **...**)⟧ =
load'⟦$l$, -, ($y$ **...**)⟧

load-lam-rec* : $(e\ \ldots)\ n\ (x\ \ldots) \to (e\ \ldots)\ H\ T\ (x\ \ldots)$
load-lam-rec*⟦(), ($y$ **...**)⟧ =
(() () () ($y$ **...**))
load-lam-rec*⟦(($l_0$ $n_0$) ($l_1$ $n_1$) **...**), ($y$ **...**)⟧ =
(($l_{0*}$ $l_{1*}$ **...**) concat⟦$H_0$, $H_1$⟧ concat⟦$T_0$, $T_1$⟧ ($y_{**}$ **...**))
where ($l_{0*}$ $H_0$ $T_0$ ($y_*$ **...**)) = load-lam-rec⟦$l_0$, $n_0$, ($y$ **...**)⟧,
(($l_{1*}$ **...**) $H_1$ $T_1$ ($y_{**}$ **...**)) = load-lam-rec*⟦(($l_1$ $n_1$) **...**), ($y_*$ **...**)⟧

**Fig. 4** The metafunction responsible for loading recursive procedures.

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.

## 4 Bytecode Evaluation

We model the Racket machine as series of transition rules that dispatch on the first element in the *C* register. Figure 5 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 5. 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 6 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 high-level instructions reduce to uses of these lower-level ones. 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 6 push and pop a a stack frame. In support of the optimization described in Section 5.2, a frame is partitioned into three segments: one for the temporary values it pushes, one for the values captured in its closure, and one for the active

$(V\ S\ H\ T\ ((\textbf{loc}\ n)\ i\ \textbf{...})) \longrightarrow (\text{stack-ref}[\![n, S]\!]\ S\ H\ T\ (i\ \textbf{...}))$ [loc]

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

$(V\ S\ H\ T\ ((\textbf{loc-clr}\ n)\ i\ \textbf{...}))$ [loc-clr]
$\longrightarrow (\text{stack-ref}[\![n, S]\!]\ \text{stack-set}[\![\textbf{uninit}, n, S]\!]\ H\ T\ (i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{loc-box}\ n)\ i\ \textbf{...}))$ [loc-box]
$\longrightarrow (\text{heap-ref}[\![\text{stack-ref}[\![n, S]\!], H]\!]\ S\ H\ T\ (i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{loc-box-noclr}\ n)\ i\ \textbf{...}))$ [loc-box-noclr]
$\longrightarrow (\text{heap-ref}[\![\text{stack-ref}[\![n, S]\!], H]\!]\ S\ H\ T\ (i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{loc-box-clr}\ n)\ i\ \textbf{...}))$ [loc-box-clr]
$\longrightarrow (\text{heap-ref}[\![\text{stack-ref}[\![n, S]\!], H]\!]\ \text{stack-set}[\![\textbf{uninit}, n, S]\!]\ H\ T\ (i\ \textbf{...}))$

$\text{stack-ref} : n\ S \to s$
$\text{stack-ref}[\![0, (v\ u\ \textbf{...}\ s)]\!] = v$
$\text{stack-ref}[\![0, ((\textbf{box}\ x)\ u\ \textbf{...}\ s)]\!] = (\textbf{box}\ x)$
$\text{stack-ref}[\![n, (u_0\ u_1\ \textbf{...}\ s)]\!] = \text{stack-ref}[\![n - 1,\ (u_1\ \textbf{...}\ s)]\!]$ where $n > 0$
$\text{stack-ref}[\![n, ((u\ \textbf{...}\ s))]\!] = \text{stack-ref}[\![n, (u\ \textbf{...}\ s)]\!]$

$\text{stack-set} : u\ n\ S \to S$
$\text{stack-set}[\![u, n, (u_0\ \textbf{...}\ u_n\ u_{n+1}\ \textbf{...}\ s)]\!] = (u_0\ \textbf{...}\ u\ u_{n+1}\ \textbf{...}\ s)$ where $n = |(u_0\ \textbf{...})|$
$\text{stack-set}[\![u, n, (u_0\ \textbf{...}\ s)]\!] = (u_0\ \textbf{...}\ \text{stack-set}[\![u, n - |(u_0\ \textbf{...})|,\ s]\!])$

$\text{heap-ref} : (box\ x)\ H \to h$
$\text{heap-ref}[\![(\textbf{box}\ x_i), ((x_0\ h_0)\ \textbf{...}\ (x_i\ h_i)\ (x_{i+1}\ h_{i+1})\ \textbf{...})]\!] = h_i$

$\text{heap-set} : h\ (box\ x)\ H \to H$
$\text{heap-set}[\![h, (\textbf{box}\ x_i), ((x_0\ h_0)\ \textbf{...}\ (x_i\ h_i)\ (x_{i+1}\ h_{i+1})\ \textbf{...})]\!] = ((x_0\ h_0)\ \textbf{...}\ (x_i\ h)\ (x_{i+1}\ h_{i+1})\ \textbf{...})$

**Fig. 5** Machine transitions related to looking at the stack.

procedure's explicit arguments. As Steele advocates [36], the [framepush] and [framepop] 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 7 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 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 8 shows the rules that allocate more space on the stack. The [let-one] rule pushes an uninitialized slot, evaluates its right-hand side, stores 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 does the almost the same, instead filling them with boxes initialized to the **undefined** value.

Figure 9 covers 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.

$$
\begin{array}{ll}
\quad (V\ S\ H\ T\ ((\textbf{set}\ n)\ i\ \textbf{...})) & \text{[set]} \\
\longrightarrow (V\ \mathsf{stack\text{-}set}[\![V, n, S]\!]\ H\ T\ (i\ \textbf{...})) & \\
\\
\quad (v\ S\ H\ T\ ((\textbf{set-box}\ n)\ i\ \textbf{...})) & \text{[set-box]} \\
\longrightarrow (v\ S\ \mathsf{heap\text{-}set}[\![v, \mathsf{stack\text{-}ref}[\![n, S]\!], H]\!]\ T\ (i\ \textbf{...})) & \\
\\
\quad (V\ S\ H\ T\ ((\textbf{swap}\ n)\ i\ \textbf{...})) & \text{[swap]} \\
\longrightarrow (\mathsf{stack\text{-}ref}[\![n, S]\!]\ \mathsf{stack\text{-}set}[\![V, n, S]\!]\ H\ T\ (i\ \textbf{...})) & \\
\\
\quad (V\ (u_0\ \textbf{...}\ (u_i\ \textbf{...}\ (u_j\ \textbf{...}\ s)))\ H\ T\ (\textbf{framepop}\ i\ \textbf{...})) & \text{[framepop]} \\
\longrightarrow (V\ s\ H\ T\ (i\ \textbf{...})) & \\
\\
\quad (V\ S\ H\ T\ (\textbf{framepush}\ i\ \textbf{...})) & \text{[framepush]} \\
\longrightarrow (V\ (((S)))\ H\ T\ (i\ \textbf{...})) &
\end{array}
$$

**Fig. 6** Rules for implicit stack manipulation.

$$
\begin{array}{ll}
\quad (V\ S\ H\ T\ ((\textbf{install-value}\ n\ e_r\ e_b)\ i\ \textbf{...})) & \text{[install-value]} \\
\longrightarrow (V\ S\ H\ T\ (\textbf{framepush}\ e_r\ \textbf{framepop}\ (\textbf{set}\ n)\ e_b\ i\ \textbf{...})) & \\
\\
\quad (V\ S\ H\ T\ ((\textbf{install-value-box}\ n\ e_r\ e_b)\ i\ \textbf{...})) & \text{[install-value-box]} \\
\longrightarrow (V\ S\ H\ T\ (\textbf{framepush}\ e_r\ \textbf{framepop}\ (\textbf{set-box}\ n)\ e_b\ i\ \textbf{...})) & \\
\\
\quad (V\ S\ ((x_0\ h_0)\ \textbf{...})\ T\ ((\textbf{boxenv}\ n\ e)\ i\ \textbf{...})) & \text{[boxenv]} \\
\longrightarrow (V\ \mathsf{stack\text{-}set}[\![(\textbf{box}\ x), n, S]\!]\ ((x\ v)\ (x_0\ h_0)\ \textbf{...})\ T\ (e\ i\ \textbf{...})) & \\
\quad \text{where } v = \mathsf{stack\text{-}ref}[\![n, S]\!],\ x \text{ fresh} &
\end{array}
$$

**Fig. 7** Machine transitions related to changing the contents of the stack.

Figure 10 gives the rules for immediate values, branches, sequences and indirect expressions. Values are moved in the value register. A **branch** expression pushes its test subexpression onto the control stack, followed by a two-position **branch** instruction containing its "then" and "else" sub-expressions. Once the test positions has been evaluated and its result stored in the value register, either the [branch-true] or [branch-false] rule applies, dispatching to the appropriate expression in the **branch** token. The [seq-two] and [seq-many] rules handle sequences and the [indirect] rule extracts an expression from the text segment to continue evaluation.

## 5 Application Expressions

The rules for application expressions are more complex than the virtual machine's other rules in order to model two of the optimizations in the Racket 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 Racket behaves when the interpreter runs) and those that do (modeling how Racket 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.

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

$\text{push-uninit} : n\ S \rightarrow S$
$\text{push-uninit}[\![0, S]\!] = S$
$\text{push-uninit}[\![n, (u\ \textbf{...}\ s)]\!] = \text{push-uninit}[\![n - 1,\ (\textbf{uninit}\ u\ \textbf{...}\ s)]\!]$

$\text{push} : (s\ldots)\ S \rightarrow S$
$\text{push}[\![(u_0\ \textbf{...}), (u_i\ \textbf{...}\ s)]\!] = (u_0\ \textbf{...}\ u_i\ \textbf{...}\ s)$

**Fig. 8** 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 $(n\ \textbf{...}) = (0\ \ldots\ |(l_0\ \textbf{...})|\ \text{-}1)$, $S_* = \text{stack-set*}[\![((\textbf{clos}\ x)\ n), \textbf{...}, S]\!]$,
$l_0 = (\textbf{lam}\ n_0\ (n_{00}\ \textbf{...})\ y_0)$, $(x\textbf{...})$ fresh

$\text{stack-set*} : (u\ n)\ \ldots\ S \rightarrow S$
$\text{stack-set*}[\![S]\!] = S$
$\text{stack-set*}[\![(u_0\ n_0), (u_1\ n_1), \textbf{...}, S]\!] = \text{stack-set*}[\![(u_1\ n_1), \textbf{...}, \text{stack-set}[\![u_0, n_0, S]\!]]\!]$

**Fig. 9** Machine transitions for procedure definition.

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

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

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

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

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

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

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

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

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

**Fig. 10** Machine transitions for values, branches, sequences, and indirect expressions

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)))
```

As shown above, 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.

The argument of 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.

A stack frame comprises three segments. The innermost segment contains the arguments to the current procedure. The next segment 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 segments, the call replaces those frames with 22 33 for the arguments, 11 for the unpacked closure, and an empty scratch space, represented as extra pair of parentheses.

### 5.1 The Reordering Optimization: An Overview

The reordering optimization sometimes moves **loc-noclr** references to the end of an application to avoid extra stack operations. Figure 11 illustrates this possibility with a reduction graph for an **application** expression where all the sub-expressions are **loc-noclr** expressions. To save space, the figure shows only the name of the first instruction in the control register (except in the case of **loc-noclr** instructions, where the index is also 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, the possible 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.

The reorderings shown in Figure 11 can avoid a **swap** operation. They also simulate how Racket's JIT can achieve similar improvements for known primitives, such as addition. For example, if the function position of the example beginning Section 5 had been (**loc-noclr** 2), then the **reorder** instruction above could also reduce as follows.

```
(uninit                          (uninit                          (uninit
 (uninit uninit (clos f) 22 33 ((ε)))  (uninit uninit (clos f) 22 33 ((ε)))  (uninit uninit (clos f) 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)))
 ((reorder                       ((reorder                       (framepush (loc 3) framepop
   (call 2)                        (call 2)                       (set 0)
   ((loc-noclr 2) ?)               ((loc 3) 0)                    framepush (loc 4) framepop
   ((loc 3) 0)                     ((loc 4) 1)                    (set 1)
   ((loc 4) 1))))                  ((loc-noclr 2) ?))))           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** instruction.

### 5.2 The Self-Application Optimization: An Overview

As discussed in Section 3, recursive tail-calls are transformed into **self-app** expressions by the loader. Evaluation of such calls can then assume that the closure record is already unpacked on the stack, allowing them to skip this step of procedure call setup.

For example, the Racket 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                          (uninit
          (f))))             (let-rec ((lam () (0)                (((ε)))
   (f))                                (application (loc-noclr 0))))  ()
                                 (application (loc-noclr 0))))   ((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,

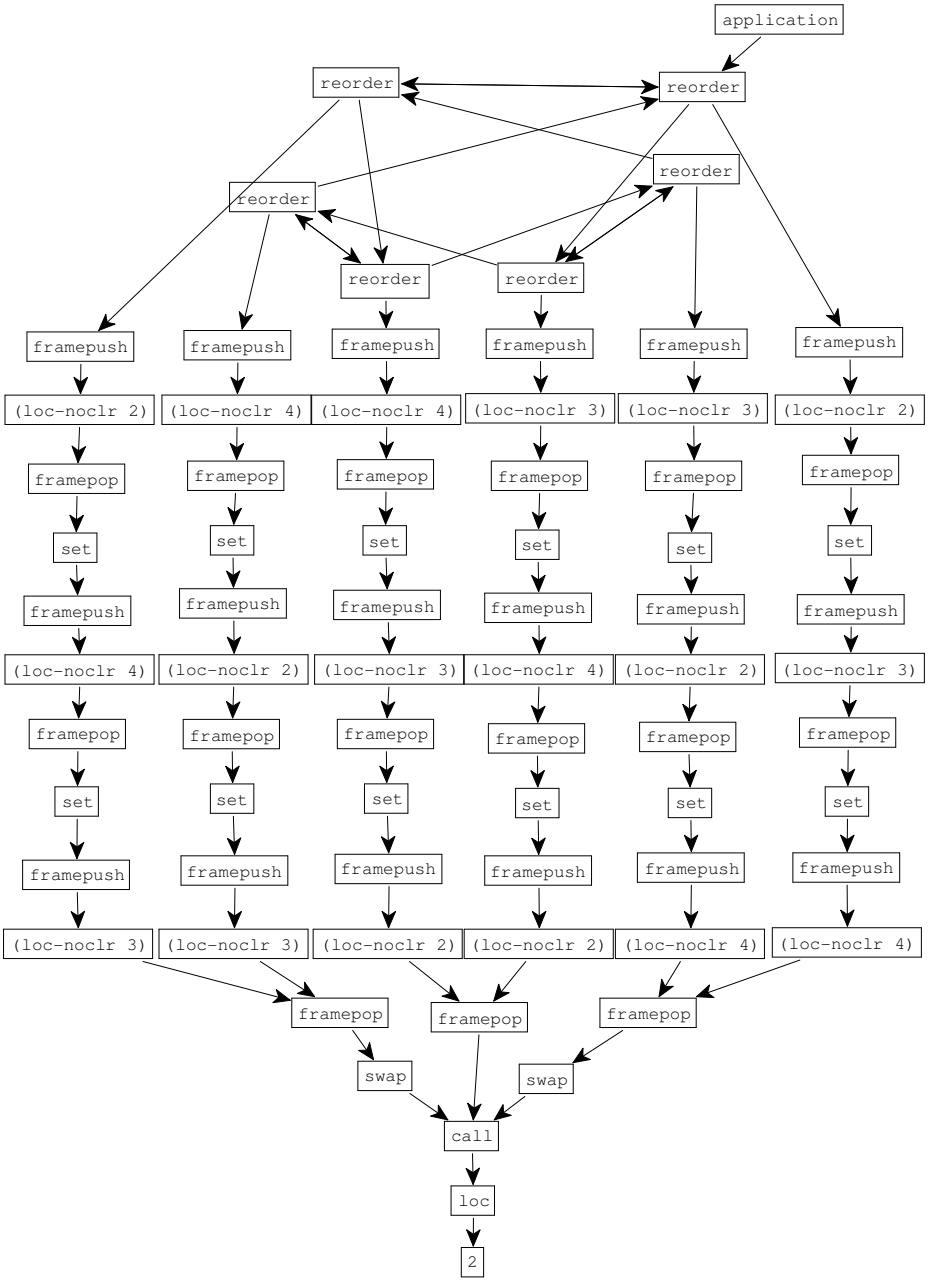



**Fig. 11** Reordering optimization reduction graph

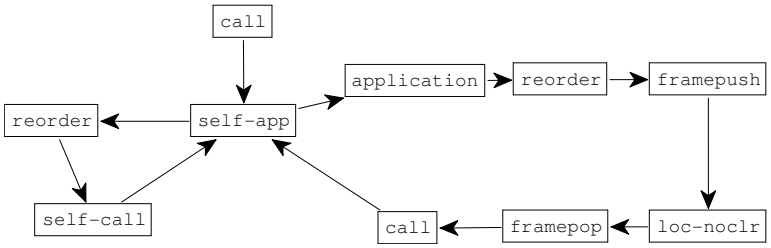



**Fig. 12** Self-app optimization reduction graph

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 without reinstalling the closure values.

Figure 12 shows 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.

## 5.3 The Complete Application Rules

Figure 13 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.

$(V\ S\ H\ T\ ((\textbf{application}\ e_0\ e_1\ \textbf{...})\ i\ \textbf{...}))$ [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]
$\longrightarrow (V\ S\ H\ T\ (\textsf{flatten}[\![((\textbf{framepush}\ e_0\ \textbf{framepop}\ (\textbf{set}\ n_0))\ \textbf{...})]\!]$
$\textbf{framepush}\ e_i\ \textbf{framepop}\ (\textbf{set}\ n_j)$
$\textsf{flatten}[\![((\textbf{framepush}\ e_{i+1}\ \textbf{framepop}\ (\textbf{set}\ n_{i+1}))\ \textbf{...})]\!]$
$\textbf{framepush}\ e_j\ \textbf{framepop}$
$(\textbf{swap}\ n_j)\ (\textbf{call}\ n)\ i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{reorder}\ (\textbf{call}\ n)\ (e_0\ n_0)\ \textbf{...}\ (e_n\ \textbf{?}))\ i\ \textbf{...}))$ [finalize-app-is-last]
$\longrightarrow (V\ S\ H\ T\ (\textsf{flatten}[\![((\textbf{framepush}\ e_0\ \textbf{framepop}\ (\textbf{set}\ n_0))\ \textbf{...})]\!]$
$\textbf{framepush}\ e_n\ \textbf{framepop}\ (\textbf{call}\ n)\ i\ \textbf{...}))$

$(V\ S\ H\ T\ ((\textbf{reorder}\ (\textbf{self-call}\ x)\ (e_0\ n_0)\ \textbf{...})\ i\ \textbf{...}))$ [finalize-self-app]
$\longrightarrow (V\ S\ H\ T\ (\textsf{flatten}[\![((\textbf{framepush}\ e_0\ \textbf{framepop}\ (\textbf{set}\ n_0))\ \textbf{...})]\!]$
$(\textbf{self-call}\ x)\ i\ \textbf{...}))$

$(V\ (u_0\ \textbf{...}\ u_i\ \textbf{...}\ (u_j\ \textbf{...}\ (u_k\ \textbf{...}\ s)))\ H\ T\ ((\textbf{self-call}\ x_i)\ i\ \textbf{...}))$ [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 $n_i \notin \{n_0, \textbf{...}\}$, $n_i = |(u_1\ \textbf{...})|$, $H = ((x_0\ h_0)\ \textbf{...}$ ,
$(x_i\ ((\textbf{clos}\ n_0\ (u_0\ \textbf{...})\ y_0)\ \textbf{...}$
$(\textbf{clos}\ n_i\ (u_i\ \textbf{...})\ y_i)$
$(\textbf{clos}\ n_{i+1}\ (u_{i+1}\ \textbf{...})\ y_{i+1})\ \textbf{...}))$
$(x_{i+1}\ h_{i+1})\ \textbf{...})$
$T = ((y_j\ e_j)\ \textbf{...}\ (y_i\ e_i)\ (y_k\ e_k)\ \textbf{...})$

$(v\ S\ H\ T\ ((\textbf{call}\ n)\ i\ \textbf{...})) \longrightarrow \textbf{error}$ [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 $n \notin \{n_0, \textbf{...}\}$

**Fig. 13** Machine transitions for procedure application.

```
(λ (car cdr null?)
  (letrec ([find (λ (it? xs)
                   (if (null? xs)
                       #f
                       (let ([x (car xs)])
                         (if (it? x)
                             x
                             (find it? (cdr xs))))))])
    find))
```

**Fig. 14** An implementation of *find*, abstracted over the list-accessor functions.

```
(proc-const (val val val)
  (let-void 1
    (let-rec ((lam (val val) (0 1 2 3) ; S0
                (branch (application (loc 4) (loc 6)) ; S1
                        #f
                        (let-one (application (loc 3) (loc 7)) ; S2
                          (branch (application (loc 6) (loc 1)) ; S3, S4
                                  (loc-noclr 0)
                                  (seq (loc-clr 0) ; S5
                                       (application ; S6
                                         (loc-noclr 3)
                                         (loc-clr 7) ; S7
                                         (application
                                           (loc-noclr 6)
                                           (loc-clr 9)))))))) ; S8
             (loc-noclr 0))))
```

**Fig. 15** The (abstracted) *find* function in Racket bytecode.

### 5.4 A Concluding Example

We conclude this section with a realistic example, intended to summarize everything covered up to this point. Figure 14 shows an implementation of the R$^6$RS library function *find*, abstracted over the list-accessor functions *car*, *cdr*, and *null?*. The *find* function takes a predicate, *it?*, and a list, *xs*, and returns the first element of *xs* satisfying *it?* (or false, if no such element exists). Compiling this function produces the bytecode in Figure 15.

The outer λ-expression in Figure 14 compiles to a **proc-const** because it has no free variables. The **let-void** in its body pushes a slot for use by the **let-rec** that precedes it. This **let-rec** ties the knot for the inner λ-expression in Figure 14, which compiles to a **lam** because it captures the list-accessor functions (in addition to itself).

Applying *find* (after instantiating it to a particular set of list-accessors) produces the first stack in Figure 16 (S0). This stack begins with the four values unpacked from the procedure's closure, followed by the two values supplied as arguments. The (*null? xs*) call beginning the procedure's body pushes one temporary slot and consequently uses the stack offsets 4 and 6 (S1). Assuming this call returns #f, control turns to the (*car xs*) call inside the **let**. The **let** and the call itself each push one temporary slot, and so this **application** uses offsets 3 and 7 (S2). When this call returns, the **let-one** places the result, *x*, on top of the stack (S3). Next, the body performs the call (*it? x*) (S4) using the offsets 6 and 1, again due to the the temporary slot pushed by **application**. Assuming the predicate returns #f, control

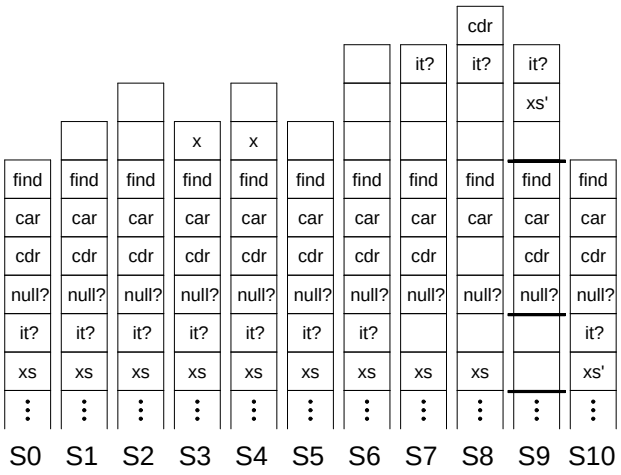



**Fig. 16** The stack progression for an application of the (instantiated) *find* function in which the supplied list is non-empty but its first element does not satisfy the given predicate.

turns to the **loc-clr** instruction, which clears the slot containing *x* because no uses remain (S5).

Next, the body begins the recursive call (S6). The compiler emits (**loc-noclr** 3) for the *find* reference, to mark the call as a self-application, and (**loc-clr** 7) for the body's final *it?* reference. Marking the call enables the [self-app-opt] rule which skips directly to the (**loc-clr** 7) instruction (S7) because the callee's address is known. The call extracting the rest of the list, *xs*', uses (**loc-noclr** 6) for the *cdr* reference, to allow the instruction to be delayed, and (**loc-clr** 9) for the body's final *xs* reference. Loading *xs* (S8) before loading *cdr* allows the machine to avoid the **swap** instruction that would otherwise be necessary to place the callee in the value register and the argument on the stack. After that call completes (S9), the machine continues the recursive call using the [self-call] rule. This rule leaves the closure segment of the active frame (between the first and second horizontal bars) in place, moves the new arguments (in the topmost two slots) to their expected positions (between the second and third horizontal bars), and jumps to the procedure's entry point (S10).

## 6 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 using 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 verifier's analysis is not especially general; it handles only the kind of bytecode expected from the Racket compiler. 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 rules out certain patterns that might be valid otherwise.

### 6.1 Abstract states

For each stack slot, the verifier's approximation records whether that slot's contents are defined, and if so, whether it holds an immediate value or a boxed value. To allow the JIT to reorder stack accesses (Section 5.1) while maintaining space-safety, the approximation further records which defined slots the program promises not to clear. The verifier rejects programs like this one, which violates its promise.

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

On the other hand, the verifier accepts program like this one, where the promise and the clear occur in separate branches.

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

In general, each slot is in one of the following states:

- **not**: not directly readable, and the slot cannot change state. This state is used for a temporary slot that the evaluator uses to store an evaluated **application** sub-expression, and it is also used for a slot that is cleared to enable space safety. The evaluation rules use the concrete value **uninit** for such slots, as well as for the ones pushed by **let-one** and **let-void**. The verifier, on the other hand, must distinguish cleared and **application**-reserved slots from these other undefined slots, necessitating a distinct abstract state.
- **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.

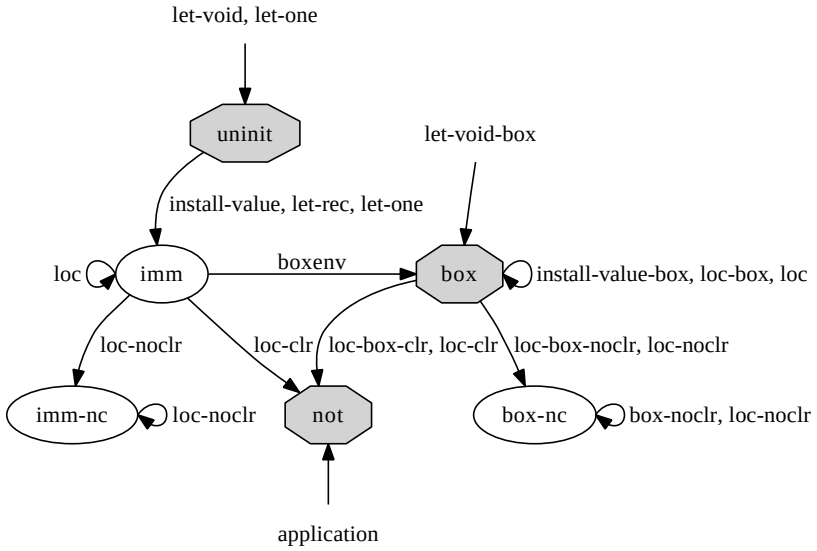



**Fig. 17** Abstract slot states and transitions.

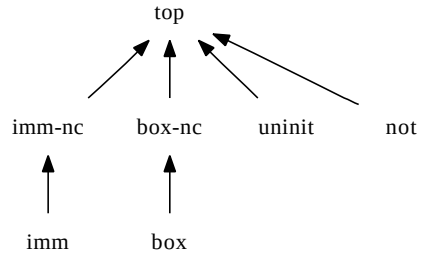



**Fig. 18** A conceivable ordering on abstract states.

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

$s$ ::= ($\hat{u}$ **...**) | **invalid**
$\hat{u}$ ::= **uninit** | **imm** | **box** | **imm-nc** | **box-nc** | **not**
$\gamma$ ::= (($n$ $\hat{u}$) **...**)
$\eta$ ::= ($n$ **...**)
$f$ ::= ($n$ $n$ ($\hat{u}$ **...**)) | $\varnothing$
$m$ ::= $n$ | **?**

**Fig. 19** The verifier's abstract interpretation language.

Figure 17 summarizes the legal transitions between these states. The shaded, octagonal states are possible initial states, and the labels on transitions indicate the bytecode forms that can trigger the transition.

When abstract interpretation joins after a branch, the stack effects of the two possible paths must be merged and checked for consistency. In many abstract interpreters,

this operation amounts to selecting the least upper bound in an ordering like the one in Figure 18. This ordering provides a starting point for understanding the verifier's treatment of branches, but the actual behavior differs in two ways. First, the verifier allows code in which one path clears a slot but the other does not, as long as the code that follows the join treats that slot as cleared. Capturing this behavior requires, for example, **imm** $\sqcup$ **not** $=$ **not** and **box** $\sqcup$ **not** $=$ **not**, which implies **imm** $\sqcup$ **box** $\sqsubseteq$ **not**. This would mean that **imm** $\sqcup$ **box** $\neq$ **top**, but the verifier considers **imm** and **box** to be inconsistent. Second, the verifier does not treat a branch's arms symmetrically. When slots differ only in the presence of no-clear annotations, the verifier arbitrarily selects the slot produced by the "else" branch. This loss of precision is harmless because the JIT never delays **branch** expressions. Section 6.2.2 discusses branch verification in more detail.

## 6.2 Verification Algorithm

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 20 through 26, while Figure 19 gives the function's full type:

- The input $e$ is a bytecode expression (Figure 1) 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.
- The input $n$ indicates how much of the current stack is local to the nearest enclosing branch. This depth is used to track abstract-state changes that must be unwound and merged with the other branch. It 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 in a context such that its value is thrown away by an enclosing **seq** expression. For expressions in such contexts, there is no danger in allowing the expression direct access to a box pointer by loading its slot using, for example, **loc** instead of **loc-box**. The compiler pass responsible for clearing dead slots for space-safety exploits this leeway: when space-safety requires a slot to be cleared but the context does not need the result, the compiler always inserts a **seq**-ignored **loc-clr** instruction—whether or not the slot actually contains a box pointer.
- The input $\gamma$ 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 $\eta$ accumulates information about no-clear 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 $\phi$ 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.

The machine accepts an acyclic bytecode expression $e$ if $\mathsf{verify}[\![e,(),0,\#f,(),(),\emptyset]\!] = (s,\gamma,\eta)$ for some $s$, $\gamma$, and $\eta$ where $s \neq \mathit{invalid}$. The machine accepts a cyclic expression if it accepts each of its named cycles. Verifying each component independently suffices because the expressions referenced by **indirect** expressions are **proc-const** expressions, which

verify⟦(**loc** *n*), ($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**), $n_l$, #f, γ, η, *f*⟧ = (($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**) γ η)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $\hat{u}_n \in$ {**imm**, **imm-nc**}
verify⟦(**loc** *n*), ($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**), $n_l$, #t, γ, η, *f*⟧ = (($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**) γ η)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $\hat{u}_n \in$ {**imm**, **imm-nc**, **box**, **box-nc**}
verify⟦(**loc-box** *n*), ($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**), $n_l$, *b*, γ, η, *f*⟧ = (($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**) γ η)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $\hat{u}_n \in$ {**box**, **box-nc**}
verify⟦(**loc-noclr** *n*), ($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**), $n_l$, #f, γ, η, *f*⟧ = (($\hat{u}_0$ **...** nc⟦$\hat{u}_n$⟧ $\hat{u}_{n+1}$ **...**) γ log-noclr⟦*n*, $n_l$, $\hat{u}_n$, η⟧)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $\hat{u}_n \in$ {**imm**, **imm-nc**}
verify⟦(**loc-noclr** *n*), ($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**), $n_l$, #t, γ, η, *f*⟧ = (($\hat{u}_0$ **...** nc⟦$\hat{u}_n$⟧ $\hat{u}_{n+1}$ **...**) γ log-noclr⟦*n*, $n_l$, $\hat{u}_n$, η⟧)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $\hat{u}_n \in$ {**imm**, **imm-nc**, **box**, **box-nc**}
verify⟦(**loc-box-noclr** *n*), ($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**), $n_l$, *b*, γ, η, *f*⟧ = (($\hat{u}_0$ **...** **box-nc** $\hat{u}_{n+1}$ **...**) γ log-noclr⟦*n*, $n_l$, $\hat{u}_n$, η⟧)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $\hat{u}_n \in$ {**box**, **box-nc**}
verify⟦(**loc-clr** *n*), *s*, $n_l$, #f, γ, η, *f*⟧ = (($\hat{u}_0$ **...** **not** $\hat{u}_{n+1}$ **...**) log-clr⟦*n*, *s*, $n_l$, γ⟧ η)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $s = (\hat{u}_0$ **...** **imm** $\hat{u}_{n+1}$ **...**)
verify⟦(**loc-clr** *n*), *s*, $n_l$, #t, γ, η, *f*⟧ = (($\hat{u}_0$ **...** **not** $\hat{u}_{n+1}$ **...**) log-clr⟦*n*, *s*, $n_l$, γ⟧ η)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $\hat{u}_n \in$ {**imm**, **box**}, $s = (\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**)
verify⟦(**loc-box-clr** *n*), *s*, $n_l$, *b*, γ, η, *f*⟧ = (($\hat{u}_0$ **...** **not** $\hat{u}_{n+1}$ **...**) log-clr⟦*n*, *s*, $n_l$, γ⟧ η)
 where $|(\hat{u}_0 \textbf{ ...})| = n$, $s = (\hat{u}_0$ **...** **box** $\hat{u}_{n+1}$ **...**)

nc⟦**imm**⟧ = **imm-nc**
nc⟦**imm-nc**⟧ = **imm-nc**
nc⟦**box**⟧ = **box-nc**
nc⟦**box-nc**⟧ = **box-nc**

log-noclr⟦$n_p$, $n_l$, $\hat{u}_p$, ($n_0$ **...**)⟧ = ($n_p$ - $n_l$ $n_0$ **...**) where $n_p >= n_l$, $\hat{u}_p \in$ {**imm**, **box**}
log-noclr⟦$n_p$, $n_l$, $\hat{u}_p$, η⟧ = η

log-clr⟦$n_p$, *s*, $n_l$, (($n_i$ $\hat{u}_i$) **...**)⟧ = (($n_h$ - $n_p$ - 1 $\hat{u}_p$) ($n_i$ $\hat{u}_i$) **...**)
 where $n_h = |s|$, $n_p >= n_l$, $|(\hat{u}_0 \textbf{ ...})| = n_p$, $s = (\hat{u}_0$ **...** $\hat{u}_p$ $\hat{u}_{p+1}$ **...**)
log-clr⟦$n_p$, *s*, $n_l$, γ⟧ = γ

**Fig. 20** The verification rules for variable references

cannot read or write slots beyond their own stack frame. This restriction makes irrelevant the contexts in which an **indirect** target appears.

### *6.2.1 Stack References*

Figure 20 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 25) produces **invalid**, causing the verifier to reject the program containing this expression. The second clause, which matches #t for the fourth argument, accommodates a direct reference to a box when the result is ignored by an enclosing **seq** expression. 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**. When the target slot is

verify⟦(**branch** $e_c$ $e_t$ $e_e$), $s$, $n_l$, $b$, γ, η, $f$⟧ = (redo-clrs⟦$\gamma_2$, trim⟦$s_3$, $s$⟧⟧ concat⟦$\gamma_2$, $\gamma_3$⟧ $\eta_3$)
where ($s_1$ $\gamma_1$ $\eta_1$) = verify⟦$e_c$, $s$, $n_l$, #f, γ, η, ∅⟧,
($s_2$ $\gamma_2$ $\eta_2$) = verify⟦$e_t$, trim⟦$s_1$, $s$⟧, 0, $b$, (), (), $f$⟧,
($s_3$ $\gamma_3$ $\eta_3$) = verify⟦$e_e$, undo-noclrs⟦$\eta_2$, undo-clrs⟦$\gamma_2$, trim⟦$s_2$, $s$⟧⟧⟧, 0, $b$, $\gamma_1$, $\eta_1$, $f$⟧

undo-clrs⟦γ, **invalid**⟧ = **invalid**
undo-clrs⟦(), $s$⟧ = $s$
undo-clrs⟦(($n_0$ $\hat{u}_0$) ($n_1$ $\hat{u}_1$) **...**), $s$⟧ = undo-clrs⟦(($n_1$ $\hat{u}_1$) **...**), set⟦$\hat{u}_0$, $n_h$ - $n_0$ - 1, $s$⟧⟧
where $n_h$ = |$s$|, $n_0$ < $n_h$
undo-clrs⟦(($n_0$ $\hat{u}_0$) ($n_1$ $\hat{u}_1$) **...**), $s$⟧ = undo-clrs⟦(($n_1$ $\hat{u}_1$) **...**), $s$⟧

undo-noclrs⟦η, **invalid**⟧ = **invalid**
undo-noclrs⟦(), $s$⟧ = $s$
undo-noclrs⟦($n_0$ $n_1$ **...**), ($\hat{u}_0$ **... imm-nc** $\hat{u}_i$ **...**)⟧ = undo-noclrs⟦($n_1$ **...**), ($\hat{u}_0$ **... imm** $\hat{u}_i$ **...**)⟧
where |($\hat{u}_0$ **...**)| = $n_0$
undo-noclrs⟦($n_0$ $n_1$ **...**), ($\hat{u}_0$ **... box-nc** $\hat{u}_i$ **...**)⟧ = undo-noclrs⟦($n_1$ **...**), ($\hat{u}_0$ **... box** $\hat{u}_i$ **...**)⟧
where |($\hat{u}_0$ **...**)| = $n_0$
undo-noclrs⟦($n_0$ $n_1$ **...**), $s$⟧ = undo-noclrs⟦($n_1$ **...**), $s$⟧

redo-clrs⟦γ, **invalid**⟧ = **invalid**
redo-clrs⟦(), $s$⟧ = $s$
redo-clrs⟦(($n_0$ $\hat{u}_0$) ($n_1$ $\hat{u}_1$) **...**), $s$⟧ = redo-clrs⟦(($n_1$ $\hat{u}_1$) **...**), set⟦**not**, $n_h$ - $n_0$ - 1, $s$⟧⟧
where $n_h$ = |$s$|, $n_0$ < $n_h$
redo-clrs⟦(($n_0$ $\hat{u}_0$) ($n_1$ $\hat{u}_1$) **...**), $s$⟧ = redo-clrs⟦(($n_1$ $\hat{u}_1$) **...**), $s$⟧

set⟦$\hat{u}$, $n$, ($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**)⟧ = ($\hat{u}_0$ **...** $\hat{u}$ $\hat{u}_{n+1}$ **...**) where |($\hat{u}_0$ **...**)| = $n$

**Fig. 21** The verification rules for branches

not local to the nearest enclosing branch and does not already have a no-clear annotation, the log-noclr function records the action in the verify function's $\eta$ result.

The last three clauses of verify in Figure 20 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 $\gamma$ result using the log-clr function. Like log-noclr, log-clr only records slots that are not local to the enclosing branch, but unlike log-noclr, it records the slots' locations as offsets from the *bottom* of the active procedure's stack frame. This representation supports the verification rule for **branch** expressions.

### *6.2.2 Branches*

Figure 21 shows the verification rule for **branch** expressions. The verify function takes an unconventional approach to branches; instead of verifying the contingent expressions independently and computing the least upper bound of the resulting stacks, verify threads one stack through both recursive calls, reverting and reapplying stack effects along the way. Though much more complicated, we adopt this approach because it mirrors the behavior the production implementation, which is not obviously equivalent to the conventional approach.

The **branch** verification clause begins by verifying the test expression, augmenting the input clears and no-clears with those performed by the test expression. For example, con-

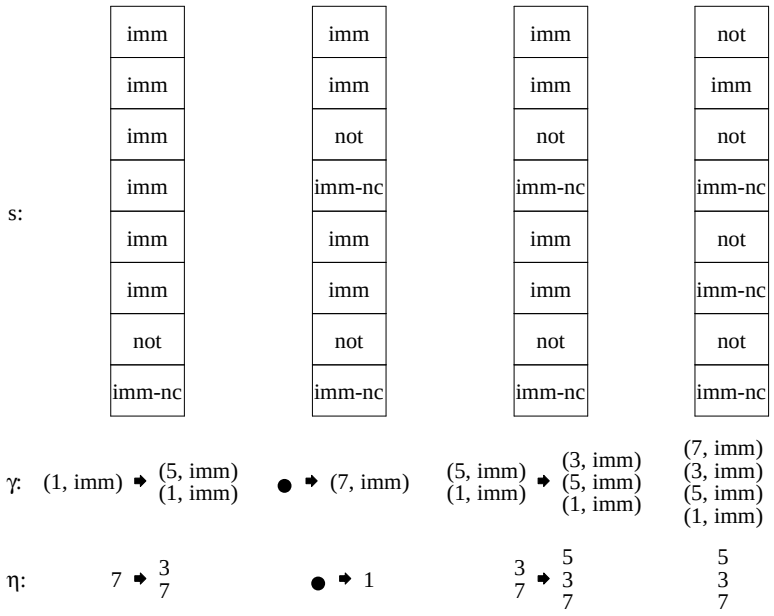



**Fig. 22** The progression of the stack, clear log, and no-clear log during verification of an example branch verification.

sider the following **branch** expression and suppose it appears in a context that has pushed eight immediate values, cleared offset 6, and promised not to clear offset 7.

```
(branch (seq (loc-clr 2)
             (loc-noclr 3))
        (seq (loc-clr 0)
             (loc-noclr 1))
        (seq (loc-clr 4)
             (loc-noclr 5)))
```

The first three columns of Figure 22 show the stacks flowing into each recursive call in **branch** clause of verify, as well as the clears and no-clears produced and consumed by those calls, when verifying the expression above. The first recursive call, which verifies the test expression, extends the initial clear log with $(5,$ **imm**$)$ (since cleared locations are recorded as offsets from the bottom of the stack), extends the initial no-clear log with 3, and returns a stack where offset 2 contains **not** and offset 3 contains **imm-nc**.

This stack, shown in the figure's second column, flows directly into the second recursive call, which verifies the "then" expression. The **branch** clause stores separately the clears and no-clears performed by the "then" expression so that it can undo them in the call's stack result before feeding that stack into the recursive call for the "else" expression. The figure's second column shows the clear log and no-clear log flowing in and out of the second recursive call.

Undoing the stack effects recorded in the output logs produces the stack shown in the figure's third column. This stack is used to verify the "else" expression, so it should be identical to the one used in verifying the "then" expression. The third recursive call stores the clears and no-clears of the "else" expression along with those performed by the test

expression and the context of the **branch**. The third column of Figure 22 shows the clear and no-clear logs flowing in and out of this recursive call.

Finally, the verify clause replays the clears performed by the "then" expression and combines those clears with the ones saved in the previous step. The no-clears performed by the "then" expression are *not* replayed or combined with those from the previous step, as explained in Section 6.1. The fourth column of Figure 22 shows the stack and clear and no-clear logs produced by this final step.

Notice that the verify function does not revert and re-apply the stack effects of **boxenv** and **install-value** expressions. This is unnecessary because the clauses for these forms disallow their use when the target slot survives the nearest enclosing branch, as indicated by the verify function's $n_l$ parameter.

### *6.2.3 Applications*

Figure 23 shows the clauses for verifying procedure applications. The last clause of verify 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** cannot use or change the slots.

The first verify clause in Figure 23 handles the case of self-application tail calls. As in the loader, the self-application rule applies when the function position is a **loc-noclr** expression that uses the stack position indicated by the last parameter to verify and 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 Racket 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 23 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.

### *6.2.4 Procedures*

Figure 24 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 lam-verified? function, which is used for checking all **lam** forms. The lam-verified? 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**

verify⟦(**application** $e_0$ $e_1$ **...**), $s$, $n_l$, $b_i$, γ, η, ($n_f$ $n_s$ ($\hat{u}$ **...**))⟧ =
verify-self-app⟦(**application** $e_0$ $e_1$ **...**), $s$, $n_l$, γ, η, ($n_f$ $n_s$ ($\hat{u}$ **...**))⟧
where $n = n_f + |(e_1 \text{ ...})|$, $e_0 = (\textbf{loc-noclr}\ n)$

verify⟦(**application** (**lam** ($\tau_0$ **...**) ($n_0$ **...**) $e$) $e_0$ **...**), $s$, $n_l$, $b$, γ, η, $f$⟧ =
verify*-ref⟦($e_0$ **...**), ($\tau_0$ **...**), $s_1$, $n_{l*}$, γ, η⟧
where $n = |(e_0 \text{ ...})|$, $n_{l*} = n + n_l$, $s_1$ = abs-push⟦$n$, **not**, $s$⟧,
lam-verified?⟦(**lam** ($\tau_0$ **...**) ($n_0$ **...**) $e$), $s_1$, **?**⟧, $s = (\hat{u}_0 \text{ ...})$

verify⟦(**application** (**proc-const** ($\tau_0$ **...**) $e$) $e_0$ **...**), $s$, $n_l$, $b$, γ, η, $f$⟧ =
verify⟦(**application** (**lam** ($\tau_0$ **...**) () $e$) $e_0$ **...**), $s$, $n_l$, $b$, γ, η, $f$⟧

verify⟦(**application** $e_0$ $e_1$ **...**), $s$, $n_l$, $b$, γ, η, $f$⟧ =
verify*⟦($e_0$ $e_1$ **...**), abs-push⟦$n$, **not**, $s$⟧, $n_{l*}$, #f, γ, η⟧
where $n = |(e_1 \text{ ...})|$, $n_{l*} = n + n_l$, $s = (\hat{u}_0 \text{ ...})$

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

verify*⟦(), $s$, $n_l$, $b$, γ, η⟧ = ($s$ γ η)
verify*⟦($e_0$ $e_1$ **...**), $s$, $n_l$, $b$, γ, η⟧ = verify*⟦($e_1$ **...**), trim⟦$s_1$, $s$⟧, $n_l$, $b$, $\gamma_1$, $\eta_1$⟧
where ($s_1$ $\gamma_1$ $\eta_1$) = verify⟦$e_0$, $s$, $n_l$, $b$, γ, η, ∅⟧

verify*-ref⟦(), (), $s$, $n_l$, γ, η⟧ = ($s$ γ η)
verify*-ref⟦($e_0$ $e_1$ **...**), (**val** $\tau_1$ **...**), $s$, $n_l$, γ, η⟧ = verify*-ref⟦($e_1$ **...**), ($\tau_1$ **...**), trim⟦$s_1$, $s$⟧, $n_l$, $\gamma_1$, $\eta_1$⟧
where ($s_1$ $\gamma_1$ $\eta_1$) = verify⟦$e_0$, $s$, $n_l$, #f, γ, η, ∅⟧
verify*-ref⟦($e_0$ $e_1$ **...**), (), $s$, $n_l$, γ, η⟧ = verify*⟦($e_0$ $e_1$ **...**), $s$, $n_l$, #f, γ, η⟧
verify*-ref⟦(), ($\tau_0$ $\tau_1$ **...**), $s$, $n_l$, γ, η⟧ = ($s$ γ η)
verify*-ref⟦((**loc** $n$) $e_1$ **...**), (**ref** $\tau_1$ **...**), $s$, $n_l$, γ, η⟧ = verify*-ref⟦($e_1$ **...**), ($\tau_1$ **...**), $s_1$, $n_l$, $\gamma_1$, $\eta_1$⟧
where ($s_1$ $\gamma_1$ $\eta_1$) = verify⟦(**loc-box** $n$), $s$, $n_l$, #f, γ, η, ∅⟧
verify*-ref⟦((**loc-noclr** $n$) $e_1$ **...**), (**ref** $\tau_1$ **...**), $s$, $n_l$, γ, η⟧ = verify*-ref⟦($e_1$ **...**), ($\tau_1$ **...**), $s_1$, $n_l$, $\gamma_1$, $\eta_1$⟧
where ($s_1$ $\gamma_1$ $\eta_1$) = verify⟦(**loc-box-noclr** $n$), $s$, $n_l$, #f, γ, η, ∅⟧
verify*-ref⟦((**loc-clr** $n$) $e_1$ **...**), (**ref** $\tau_1$ **...**), $s$, $n_l$, γ, η⟧ = verify*-ref⟦($e_1$ **...**), ($\tau_1$ **...**), $s_1$, $n_l$, $\gamma_1$, $\eta_1$⟧
where ($s_1$ $\gamma_1$ $\eta_1$) = verify⟦(**loc-box-clr** $n$), $s$, $n_l$, #f, γ, η, ∅⟧
verify*-ref⟦($e$ **...**), ($\tau$ **...**), $s$, $n_l$, γ, η⟧ = (**invalid** γ η)

stack-ref⟦$n$, ($\hat{u}_0$ **...** $\hat{u}_n$ $\hat{u}_{n+1}$ **...**)⟧ = $\hat{u}_n$
where $|(\hat{u}_0 \text{ ...})| = n$

abs-push⟦0, $\hat{u}$, ($\hat{u}_0$ **...**)⟧ = ($\hat{u}_0$ **...**)
abs-push⟦$n$, $\hat{u}$, ($\hat{u}_0$ **...**)⟧ = abs-push⟦$n$-1, $\hat{u}$, ($\hat{u}$ $\hat{u}_0$ **...**)⟧

arg⟦**val**⟧ = **imm**
arg⟦**ref**⟧ = **box**

**Fig. 23** The verification rules for applications

verify⟦(**lam** (τ **...**) ($n_0$ **...**) $e$), $s$, $n_l$, $b$, γ, η, $f$⟧ = ($s$ γ η)
 where lam-verified?⟦(**lam** (τ **...**) ($n_0$ **...**) $e$), $s$, **?**⟧, τ = **val**
verify⟦(**proc-const** (τ **...**) $e$), $s$, $n_l$, $b$, γ, η, $f$⟧ = verify⟦(**lam** (τ **...**) () $e$), $s$, $n_l$, $b$, γ, η, $f$⟧
 where τ = **val**
verify⟦(**case-lam** $l$ **...**), $s$, $n_l$, $b$, γ, η, $f$⟧ = ($s$ γ η)
 where lam-verified?⟦$l$, $s$, **?**⟧ , **...**, $l$ = (**lam** (**val ...**) ($n$ **...**) $e$)

lam-verified?⟦(**lam** ($\tau_0$ **...**) ($n_0$ **...**) $e$), $s$, $m$⟧ = $s_1 \neq$ **invalid**
 where $n_d = |s|$, $n_{d^*} = |(\tau_0 \text{ ...})| + |(n_0 \text{ ...})|$, $n_0 < n_d$ , **...**,
 stack-ref⟦$n_0$, $s$⟧ ∉ {**uninit**, **not**} , **...**,
 ($\hat{u}$ **...**) = (drop-noclr⟦stack-ref⟦$n_0$, $s$⟧⟧ **...**),
 $f$ = extract-self⟦$m$, ($n_0$ **...**), ($\tau_0$ **...**), ($\hat{u}$ **...**)⟧,
 ($s_1$ $\gamma_1$ $\eta_1$) = verify⟦$e$, ($\hat{u}$ **...** arg⟦$\tau_0$⟧ **...**), $n_{d^*}$, #f, (), (), $f$⟧,
 $s$ = ($\hat{u}_0$ **...**)
lam-verified?⟦**any**, $s$, $m$⟧ = #f

drop-noclr⟦**imm-nc**⟧ = **imm**
drop-noclr⟦**box-nc**⟧ = **box**
drop-noclr⟦$\hat{u}$⟧ = $\hat{u}$

extract-self⟦**?**, ($n_0$ **...**), ($\tau_0$ **...**), ($\hat{u}_0$ **...**)⟧ = ∅
extract-self⟦$n_i$, ($n_0$ **...** $n_i$ $n_{i+1}$ **...**), ($\tau_0$ **...**), ($\hat{u}_0$ **...**)⟧ = (|($n_0$ **...**)| |($\tau_0$ **...**)| ($\hat{u}_0$ **...**))
 where $n_i \notin \{n_{i+1}, \text{...}\}$
extract-self⟦$n$, ($n_0$ **...**), ($\tau_0$ **...**), ($\hat{u}_0$ **...**)⟧ = ∅

**Fig. 24** The verification rules for procedures

or **uninit**). Those abstract values are copied from the stack where they are captured, but no-clear annotations are stripped in the copy, because an application of a closure unpacks values into fresh stack slots that can be cleared independently.

The last argument to lam-verified? 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 lam-verified? 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.

### *6.2.5 Other Forms*

Verification of **let-rec** and other stack-modifying forms is shown in Figure 25. 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

$$
\begin{array}{l}
\mathsf{verify}[\![(\textbf{let-one}\ e_r\ e_b), (\hat{u}_1\ \textbf{...}), n_l, b, \gamma, \eta, f]\!] \\
\quad = \mathsf{verify}[\![e_b, (\textbf{imm}\ \hat{u}_{1^*}\ \textbf{...}), n_l + 1,\ b, \gamma_1, \eta_1, \\
\qquad \mathsf{shift}[\![1, f]\!]]\!] \\
\quad \text{where } s_0 = (\textbf{uninit}\ \hat{u}_1\ \textbf{...}), (s_1\ \gamma_1\ \eta_1) = \mathsf{verify}[\![e_r, s_0, n_l{+}1,\ \#\mathrm{f}, \gamma, \eta, \varnothing]\!], (\textbf{uninit}\ \hat{u}_{1^*}\ \textbf{...}) = \mathsf{trim}[\![s_1, s_0]\!] \\
\mathsf{verify}[\![(\textbf{let-void}\ n\ e), s, n_l, b_i, \gamma, \eta, f]\!] \\
\quad = \mathsf{verify}[\![e, \mathsf{abs\text{-}push}[\![n, \textbf{uninit}, s]\!], n + n_l, \\
\qquad b_i, \gamma, \eta, \mathsf{shift}[\![n, f]\!]]\!] \\
\quad \text{where } s = (\hat{u}_0\ \textbf{...}) \\
\mathsf{verify}[\![(\textbf{let-void-box}\ n\ e), s, n_l, b_i, \gamma, \eta, f]\!] \\
\quad = \mathsf{verify}[\![e, \mathsf{abs\text{-}push}[\![n, \textbf{box}, s]\!], n + n_l, \\
\qquad b_i, \gamma, \eta, \mathsf{shift}[\![n, f]\!]]\!] \\
\quad \text{where } s = (\hat{u}_0\ \textbf{...}) \\
\mathsf{verify}[\![(\textbf{install-value}\ n\ e_r\ e_b), s, n_l, b, \gamma, \eta, f]\!] \\
\quad = \mathsf{verify}[\![e_b, \mathsf{set}[\![\textbf{imm}, n, s_2]\!], n_l, b, \gamma, \eta, f]\!] \\
\quad \text{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 \textbf{invalid}, \\
\qquad \textbf{uninit} = \mathsf{stack\text{-}ref}[\![n, s_2]\!] \\
\mathsf{verify}[\![(\textbf{install-value-box}\ n\ e_r\ e_b), s, n_l, b, \gamma, \eta, f]\!] \\
\quad = \mathsf{verify}[\![e_b, s_2, n_l, b, \gamma_1, \eta_1, f]\!] \\
\quad \text{where } 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 \textbf{invalid}, \\
\qquad \mathsf{stack\text{-}ref}[\![n, s_2]\!] \in \{\textbf{box}, \textbf{box-nc}\}, s = (\hat{u}_0\ \textbf{...}) \\
\mathsf{verify}[\![(\textbf{boxenv}\ n_p\ e), (\hat{u}_0\ \textbf{...}\ \textbf{imm}\ \hat{u}_{n+1}\ \textbf{...}), n_l, b, \gamma, \eta, f]\!] \\
\quad = \mathsf{verify}[\![e, (\hat{u}_0\ \textbf{...}\ \textbf{box}\ \hat{u}_{n+1}\ \textbf{...}), n_l, b, \gamma, \eta, f]\!] \\
\quad \text{where } |(\hat{u}_0\ \textbf{...})| = n_p,\ n_p < n_l \\
\mathsf{verify}[\![(\textbf{let-rec}\ (l\ \textbf{...})\ e), (\hat{u}_0\ \textbf{...}\ \hat{u}_n\ \textbf{...}), n_l, b, \gamma, \eta, f]\!] \\
\quad = \mathsf{verify}[\![e, s_1, n_l, b, \gamma, \eta, f]\!] \\
\quad \text{where } n = |(l\ \textbf{...})|,\ |(\hat{u}_0\ \textbf{...})| = n, \hat{u}_0 = \textbf{uninit}\ , \textbf{...},\ n <= n_l, s_1 = \mathsf{abs\text{-}push}[\![n, \textbf{imm}, (\hat{u}_n\ \textbf{...})]\!], \\
\qquad (n_i\ \textbf{...}) = (0\ \ldots\ |(l\ \textbf{...})|\ \text{-}1), \mathsf{lam\text{-}verified?}[\![l, s_1, n_l]\!]\ , \textbf{...}, v = \textbf{val}, l = (\textbf{lam}\ (v\ \textbf{...})\ (n_0\ n_1\ \textbf{...})\ e_0)
\end{array}
$$

$$
\begin{array}{l}
\mathsf{shift}[\![n, \varnothing]\!] = \varnothing \\
\mathsf{shift}[\![n, (n_f\ n_s\ (\hat{u}\ \textbf{...}))]\!] = (n + n_f n + n_s(\hat{u}\ \textbf{...}))
\end{array}
$$

**Fig. 25** The verification rules for stack operations

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

**Fig. 26** The verification rules for the remaining cases

**let-rec** form is verified in much the same way as **install-value**, but it handles multiple slots. It also calls lam-verified? instead of the generic verify, and it supplies a self-application slot for each call to lam-verified?.

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 verified separately. The final clause is a catch-all that reports an invalid expression when the side conditions of other clauses are not met.

## 7 Randomized Testing

To assess the correctness of the bytecode verification algorithm and machine model, we applied PLT Redex's QuickCheck-inspired [5] randomized testing framework [22]. This

| Test subject | Tested properties | Bugs found | Bugs known *a priori* |
|---|---|---|---|
| Implementation | Internal | 5 | 5 |
| | External | 3 | 0 |
| Model | Internal | 24 | 0 |
| | External | 4 | 1 |

**Fig. 27** Summary of bugs found in the model and implementation

framework attempts to automatically falsify claims encoded as Racket predicates by testing them on many random examples. For example, to test the claim that the machine does not get stuck on bytecode accepted by the verifier, we write a predicate that takes a bytecode expression, applies the verification algorithm, and if the bytecode passes, repeatedly applies the machine transitions in search of stuck states. There is no need to craft a random example generator as in QuickCheck; Redex derives one for free from the grammar on which the model's reduction relations and meta-functions are defined.

Sometimes, this free generator produces effective examples, but other times, only a tiny portion of them induces interesting behavior in the predicate under test. For example, bytecode expressions chosen freely from the grammar in Figure 1 are unlikely to pass the verifier; moreover, those that do pass tend to be small and collectively similar. In one sample of 25,000 expressions, only about 12% pass the verifier. A cursory examination of the sample suggests that many of expressions are rejected for just a few common violations, such as out-of-bounds stack indices and procedures that appear in non-function positions but nevertheless demand **ref**-arguments. Fortunately, those particular violations are easy to remedy in a post-generation pass that replaces out-of-bounds indices with randomly chosen in-bounds indices and illegal **ref** annotations with **val** annotations. With these corrections, the portion that passes the verifier increases to about 58%. It is not difficult to improve these figures further, by incorporating corrections for other common violations, but doing so is time consuming and, taken to the extreme, produces an independent specification of the verifier, which may or may not coincide with the one defined in Section 6. For this reason, we restrict our corrections to the aforementioned two and run some (though fewer) tests without applying any corrections at all.

We tested five properties, which we classify in two broad categories: *external* and *internal*. External properties relate the virtual machine model discussed here to its production implementation in C. Internal properties, on the other hand, are typical meta-theoretic propositions, such as safety. Testing these five properties repeatedly over the course of the model's development revealed 36 bugs in all. Of these bugs, 30 were previously unknown; the remaining were known deficiencies which we intended to address later or intentionally seeded errors (see Section 7.1). These 30 genuine bugs eluded diligent manual testing which preceded each round of random tests—by the model's completion, our hand-written test suite grew to comprise over 200 individual test cases, collectively spanning more than 1,200 LOC. Figure 27 breaks down the bugs according to the property that found them and the location of the error.

The remainder of this section states the particular properties, the bugs found by testing them (as well as a few *not* found), and concludes with more general lessons learned.

7.1 Internal Properties

The first internal property states that the verify function defined in Section 6 produces an answer for every bytecode expression.

**totality** For a bytecode expression $e$, $\mathsf{verify}[\![e, (), 0, \#f, (), (), \emptyset]\!] = (s, \gamma, \eta)$ for some $s$, $\gamma$, and $\eta$.

The second internal property relates the verifier to the evaluator. It holds if the machine cannot get stuck while evaluating verified bytecode.

**safety** For a bytecode expression, $e$, containing the named cycles $((x_0\ e_0)\ ...)$, if the verifier accepts $e$ and $\mathsf{load}[\![e, ((x_0\ e_0)\ ...)]\!] \rightarrow^* (V\ S\ H\ T\ C)$, 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 third and final internal property, an approximation of confluence, holds if the machine's evaluation rules define at most one result for a valid program.

**confluence** For a bytecode expression $e$ containing the named cycles $((x_0\ e_0)\ ...)$, if $\mathsf{load}[\![e, ((x_0\ e_0)\ ...)]\!] \rightarrow^* (V\ S\ H\ T\ ())$ and $\mathsf{load}[\![e, ((x_0\ e_0)\ ...)]\!] \rightarrow^* (V'\ S'\ H'\ T'\ ())$, 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 24 violations. In all cases, the problem was one we introduced while constructing the model, not a latent flaw inherited from the years-old implementation.

*Bug #1* Some of these bugs were simple typos. For example, the left-hand side of the model's original [app-arity] rule (Section 5.3) matched heaps with the following shape.

$$((x_0\ u_0)\ ...\ (x_i\ ((\mathbf{clos}\ n_0\ (u'_0\ ...)\ y_0)\ ...))\ (x_i + 1\ u_i + 1)...)$$

This rule provides an error transition for this program

```
(let-one 'x
  (boxenv 0
    (application (lam (val) (0) (loc-box 0)))))
```

but not this program

```
(let-one (lam () () 'x)
  (boxenv 0
    (application (lam (val) (0) (loc-box 0)))))
```

because it does not apply when, for any $j \neq i$, the value at heap address $x_j$ is a closure, since the $u$ non-terminal does not include closures.

*Bug #2* Other bugs were cases where we failed to grasp some subtlety of the implementation and consequently produced an oversimplified model. For example, when the implementation verifies a **loc-clr** or **loc-box-clr** expression and the target slot is not popped before control returns to the nearest enclosing **branch** expression, it saves a pointer to the abstract stack location representing the target slot. Our model used a stack offset instead of a raw pointer, leading to two mistakes in our original definitions of log-clear (Figure 20) and redo-clrs (Figure 21).

First, it necessitates the guard on the third clause of redo-clrs. This guard prevents the model from attempting to re-clear the slot if it has been popped, as would otherwise happen just after verifying the outer **branch** in the following expression.

```
(proc-const (val)
  (branch (loc 0)
          (let-one 'x
            (branch (loc 1)
                    void
                    (loc-clr 1)))
          void))
```

Our original definition lacked the guard, and for this expression, it results in an undefined use of **set** (Figure 21).The pointer-based implementation, on the other hand, has no trouble without the guard; it safely writes the **not** value to a location that it did not deallocate but will not read again until the stack regrows to its former depth.

Second, implementing this guard precludes our original strategy of storing the locations of cleared slots as offsets from the top of the stack. The guard cannot compare such an offset to the current stack depth since slots may have been popped since storing the offset. For example, if the previous expression were a procedure of two arguments, then the location of the cleared slot (offset 1) would appear to be in bounds after the outer **branch**.

The manual process of translating a production C implementation into a relatively simple model did, however, reveal six previously unknown verification bugs which could be exploited to crash the virtual machine, violate the source-level invariants it promises, or induce undefined behavior. To see whether randomized testing would have discovered these bugs had we missed them, we intentionally introduced them into the model. Our randomized testing procedure finds five of the six, which we summarize here; Section 7.3 explains the bug it missed.

*Bug #3* The verification rules in Figure 23 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, allowing, for example, the following expression to borrow a reserved slot.

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

Although this expression violates neither safety nor confluence, 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))
             (install-value 0 'x
               (boxenv 0 'y)))
```

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))
  (install-value 0 'x (boxenv 0 'y))
  (loc-box 0))
```

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 #4 & #5* 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, 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)))
```

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

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

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)))
```

*Bug #6* The original verifier neglected to check the bodies of **case-lam** expressions at all. Reassuringly, randomized testing discovered this omission immediately.

*Bug #7* 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)))
```

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

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

```
(application
  (proc-const (val)
    (let-one (application (application (loc 1)))
      'x))
  (proc-const () (lam () (0) (loc 0))))
```

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

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

and

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

## 7.2 External Properties

Two external properties relate the machine model to its production implementation. The first says that the two verifiers either both accept an expression or both reject it. The second says that in the case where they accept it, the two evaluators produce the same answer. Testing these properties revealed seven bugs—three in the model and four in the implementation—of which only Bug #8 was previously known.

*Bug #8* The **let-one** verification rule in Figure 25 does not allow that form's right-hand side to use as temporary storage the uninitialized slot pushed in anticipation of its result. For example, the rule rejects the following program for temporarily storing 'x in the slot that eventually holds 'y.

```
(let-one (install-value 0 'x
                        'y)
  (loc 0))
```

No harm comes from allowing such programs, since the **let-one**'s implicit update merely replaces one **imm** value with another. Earlier versions of our model allowed this pattern, but the production implementation rejects it, since the compiler does not intentionally emit it. In Figure 27, we consider this discrepancy to be a model bug because we did not intend for it to deviate from the implementation's behavior.

*Bug #9* The **let-rec** verification rule in Figure 25 insists that each **lam** expression capture at least one slot. For example, the rule rejects the following program.

```
(let-void 1
  (let-rec ((lam () () 'x))
    (loc 0)))
```

This program evaluates safely according to the transition rules in Section 4, and earlier versions of our verifier model allowed it, but the situation in the production implementation is more subtle than the model suggests. In the implementation, the **proc-const** and **lam** forms produce values with different runtime representations, and the **let-rec** form is prepared to deal only with the latter. This appears to be no problem, since the bytecode grammar (Figure 1) restricts procedure positions to the **lam** form, but an optimization in the implementation's bytecode parser indiscriminately replaces closed **lam** expressions with **proc-const** expressions. Again, we count the discrepancy as a bug in the model because the implementation intentionally rejects programs that would be broken by the parser's optimization.

*Bug #10* The optimizations performed in the implementation's bytecode parser have another surprising implication: they do not preserve the verifier's answer (though they seem to preserve the evaluator's answer). For example, the parser simplifies the expression (**seq** (**loc** 42) 'x), which the verifier rejects, into the expression 'x which the verifier accepts.

The parser's optimizations can also turn a valid expression into an invalid one. Take for example the following expression, adapted for readability from one actually produced by the test case generator.

```
(let-void-box 1
  (let-one (branch #t (loc-box-noclr 1) 'x)
    (loc-box-clr 1)))
```

The verifier accepts this expression because the no-clear promise in the first branch is rewound before treating the second branch but never replayed, thus allowing the subsequent clear. The parser's optimizations simplify this expression to the following one, which the verifier rejects for the subsequent clear.

```
(let-void-box 1
  (let-one (loc-box-noclr 1)
    (loc-box-clr 1)))
```

This time, we count the discrepancy as an implementation bug because the surprising behavior was unintentional (and arguably undesirable).

*Bug #11* We successfully addressed bug #14 (see Section 7.3) in the model, but a mistake in the fix to the implementation continued to allow **install-value** expressions to overwrite initialized slots.

*Bug #4 again* Bug #4 was present not only in the model, but also in the implementation. We failed to fix it in the implementation when we found it in the model, and forgot about it. Without testing the external properties, we would not have realized our mistake.

*Bug #12* The **boxenv** rule in Figure 25 propagates to the recursive call the flag that indicates whether the expression's result is ignored by an enclosing **seq** expression, thus allowing expressions like this one.

```
(let-one 'x
  (seq (boxenv 0
         (loc-clr 0))
       'y))
```

The implementation, on the other hand, conservatively supplies #f to the recursive call, which we count as a bug in the implementation because it unintentionally weakens the flag's intended meaning.

*Bug #13* The first clause in the definition of log-noclr in Figure 20 checks whether the cleared slot already has a no-clear annotation; if so, the no-clear action is not logged because doing so could cause the subsequent undo-noclrs operation to strip annotations applied *outside* the branch. For example, omitting this side-condition (as we originally did in the model) accepts expressions like the following, in which rewinding the no-clear applied in first branch undoes the one applied in the function position, allowing the clear in the second branch.

```
(let-one (proc-const (val) (loc 0))
  (application
    (loc-noclr 1)
    (branch #f
            (loc-noclr 1)
            (loc-clr 1))))
```

If the machine chooses to delay the function position, then evaluation gets stuck because the taken branch clears the slot it reads.

## 7.3 Undiscovered Bugs

In the course of developing the machine model, we discovered three bugs which we have not managed to find with random testing. To put the testing successes in Sections 7.1 and 7.2 in perspective (and as fodder for papers on other testing techniques), we describe these three bugs here.

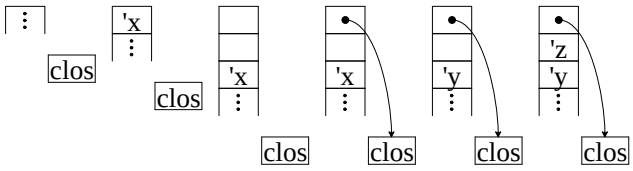
*Bug #14* The first is one of the six latent bugs intentionally inherited from the implementation. To provide optimization opportunities, the verification rules in Section 6 restrict a program's ability to change the contents of a slot that already contains an immediate value or a box pointer. In particular, 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 23) does not permit the program to clear or box these values.

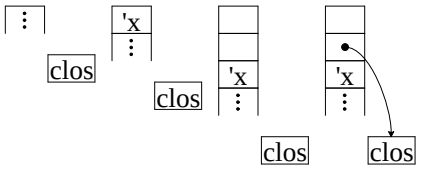
In two respects, the original verification algorithm did not sufficiently restrict updates to initialized slots. First, it allowed the **install-value** and **let-rec** forms to overwrite initialized slots, permitting expressions like the following.

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

This expression produces 'y if the JIT chooses to delay the **loc-noclr** instruction. The following stack progression, ending at the execution of the **loc-noclr** instruction, shows why.



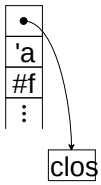

If, on the other hand, the JIT does not delay the **loc-noclr**, then the expression evaluates to 'x, this time via the following stack progression (again, ending the **loc-noclr**).




The failure to restrict these forms also breaks the JIT's optimization of **self-app** expressions. For example, consider the following expression.

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

When control first reaches the procedure's body, the stack looks as follows.




Consequently, the **install-value** in the body writes 'b to the slot containing 'a, and the **branch** makes a recursive call instead of returning the written value. What happens next

depends on the optimization level. The optimized calling convention leaves untouched the stack slots corresponding to the closure, causing the second invocation to begin with 'b in the slot second from the top and thus return 'b. The unoptimized calling convention, on the other hand, reinstalls the values in the procedure's closure, causing the second invocation to begin with 'a in the second slot and thus return 'a.

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. Just as overwriting these values breaks the JIT's **self-app** optimization, boxing them causes a problem too. Take the following expression for example.

```
(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)))))
```

The first invocation of the procedure replaces the procedure value captured in slot 1 with a box pointer, causing the second invocation to get stuck at the (**loc** 1) instruction.

*Bug #15* Our original formulation of the **case-lam** rule in Figure 24 failed to restrict the constituent **lam** expressions to **val** arguments. This omission allows expressions like the following, where a procedure expecting a boxed argument may be applied to an immediate value.

```
(application (case-lam (lam (ref) () (loc-box 0)))
             'x)
```

*Bug #16* The cleared slots reported by the $\gamma$ and $\eta$ results of the **branch** rule in Figure 21 include the slots cleared by the contingent expressions $e_t$ and $e_f$, even though these slots are also marked as cleared in the rule's stack result. This apparent redundancy is in fact necessary when the verified **branch** appears within the "then" expression of an enclosing **branch**, as in the following example.

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

In such cases, all slots cleared by the inner **branch** must be un-cleared before proceeding with the "else" expression of the outer **branch**. Our original formulation of the **branch** rule, however, failed to include the slots cleared by $e_t$ and $e_f$ in the second position of its result, thus causing the verifier to reject the above safe expression.

## 7.4 Discussion

Figure 28 shows the rates at which our random testing procedure finds the bugs discussed above, *i.e.* the ratio of examples checked to error-revealing instances. The rate for bug #7

| Bug # | Description | Discoveries | Attempts | Rate |
|---|---|---|---|---|
| 1 | [app-arity] typo | 9,004 | 25,000,000 | 1/2,777 |
| 2 | popped slots re-cleared | 364 | 25,000,000 | 1/68,681 |
| 3 | **application** space | 320 | 25,000,000 | 1/78,125 |
| 4 | **boxenv** branch effects | 4 | 25,000,000 | 1/6,250,000 |
| 5 | **let-rec** branch effects | 0 | 25,000,000 | – |
| 6 | **case-lam** ignored | 23,217 | 25,000,000 | 1/1,077 |
| 7 | closure capture | 28,096 | 25,000,000 | 1/890 |
| 8 | **let-one** right-hand side | 30 | 100,000 | 1/3,333 |
| 9 | closed **lam** in **let-rec** | 42 | 100,000 | 1/2,381 |
| 10 | unsound optimization | 572 | 100,000 | 1/175 |
| 11 | unrestricted update | 320 | 100,000 | 1/312 |
| 4, again | **boxenv** branch effects | 5 | 100,000 | 1/20,000 |
| 12 | ignored **boxenv** | 7 | 100,000 | 1/14,286 |
| 13 | no-clear rewind | 3 | 2,500,000 | 1/833,333 |
| 14 | unrestricted update | 0 | $>$ 25,000,000 | — |
| 15 | **case-lam** parameters | 0 | $>$ 25,000,000 | — |
| 16 | **branch**-cleared slots | 0 | $>$ 25,000,000 | — |
| 5+ | Bug #5 in the presence of #6, #8, and #9 | 5 | 25,000,000 | 1/5,000,000 |
| 13+ | Bug #13 in the presence of #6 and #8 | 13 | 2,500,000 | 1/192,308 |

**Fig. 28** The rates (discoveries per attempts) at which randomized testing finds the bugs described in this section.

reflects a bytecode generator that does *not* correct for out of bounds stack indices or illegal **ref**-arguments, since the former hides the bug.

As the figure shows, the difficulty of finding a particular bug varies dramatically, with the easiest requiring only a few hundred random examples and the hardest requiring several million. A modern desktop checks the internal properties on approximately 3,000 examples per minute per core and the external properties on approximately 100 examples per minute per core. In each case, nearly all the time is spent checking the example, not constructing it. Checking the internal properties is much faster than checking the external properties because the latter creates a fresh VM process for each check, in case the example crashes or corrupts the VM.

At these rates, checking several million examples can take days. Fortunately, this is computer-time, not human-time, and so one is free to continue developing the model while Redex searches for a counterexample. Furthermore, most of the 26 bugs we introduced while creating the model are quickly found by random testing—fortunate, since these simple mistakes, more-so than years-old fundamental errors in the verification algorithm, are the ones for which rapid feedback is most appreciated. While developing the model, we never felt hampered by the latency of test feedback; in fact, we did not even require days of computing time until we reproduced the results to compile Figure 28.

All but the figure's final two rows show the detection rate for a bug in the absence of all other known bugs. This setup is a bit unrealistic, since real bugs are typically discovered in the presence of other bugs, and bug interactions sometimes affects detection rates. For example, bugs #5 is much easier to discover before fixing bugs #6, #8, and #9. Similarly, bug #13 is easier to discover in the presence of bugs #6 and #8. We generally report rates for bugs in isolation to avoid considering too many bug pairings, but the last two lines in the figure show how these rates can change.

The rates for bugs #11 and #4 (the second time), the bugs unexpectedly discovered with external properties, demonstrate the relative power of differential testing [27]. Redex has

trouble finding these bugs by testing the internal properties but easily finds the lingering problems in the implementation when testing it against a corrected model. Differential testing finds these bugs more easily because it does not need to construct examples that exploit the verifier's vulnerability to cause a bad outcome; it suffices to tickle the vulnerability in any observable way.

## 8 Related Work

We are not the first to use automated testing to validate a programming language's semantics. Berghofer and Nipkow use a similar randomized testing technique in Isabelle/HOL [3]. They demonstrate its utility by finding counterexamples to false claims about the semantics of a tiny concurrent language. Cheney and Momigliano use logic programming to perform bounded exhaustive testing in $\alpha$Prolog [4]. With it, they find seeded bugs in a semantics for a simply typed $\lambda$-calculus with pairs and a known limitation of $\lambda^{zap}$ [39]. Roberson *et al.* use a SAT solver guided by dynamic analysis to perform bounded exhaustive testing for a declarative subset of Java [32]. With it, they find seeded bugs in an extension of Featherweight Java [21] with ownership types [6]. In all these cases, the semantics tested is essentially an idealized core model. We believe that our experience with the model in this paper, along with an earlier case study [22] of the R$^6$RS formal semantics [33], provides valuable empirical evidence that automated testing scales to large models.

Of course, large, realistic languages have been formalized and verified in proof assistants. Lee *et al.* formalize an elaboration of Standard ML and prove its type safety in Twelf [24]. Norrish formalizes most of C in HOL and proves several properties, including type safety (for certain well-behaved expressions) [30]. Nipkow and van Oheimb [29] and Syme [38] show type for fragments of Java. In most of these cases, the verification effort establishes only type safety (and subsidiary lemmas), presumably because such a result is a tremendous feat in itself. With testing, it is relatively easy to check additional properties, such as soundness of optimizations and correspondence with an independent implementation.

Racket's bytecode verification algorithm resembles the ones typically applied to JVM bytecode, originally due to Gosling and Yellin [19,26,40]. These algorithms involve an abstract interpreter that conservatively approximates a defensive VM [10], using forward dataflow analysis to resolve the uncertain control-flow. There have been many formalizations of this approach, including several with machine-checked proofs of their soundness [9,15, 17,31,34,35]. The two primary challenges in JVM bytecode verification do not arise in our context. The first, enforcing object initialization, requires a form of alias analysis to ensure that bytecode does not interact with any pre-object before executing one of its (imperative) constructors. Racket objects are created with a single procedure call, avoiding this issue entirely. The second challenge concerns subroutines, *i.e.* shared code blocks that execute in the stack frame of their caller. Expressive support for subroutines requires a flow analysis that does conflate the constraints of too many control contexts. The Racket VM's cyclic bytecode nearly poses a similar challenge, but because the target of an **indirect** is always a **proc-const**, and procedures cannot read or write beyond their own stack frame, no sophisticated analysis is necessary. Leroy provides an excellent survey of object initialization, subroutines, and other issues in JVM bytecode verification [25].

Other work investigates bytecode verification for CIL [12], the usual target for compilers of C#, Visual Basic, and other .NET languages. Gordon and Syme [18] present a type system and operational semantics for the core of CIL using DECLARE [37], and later work

by Yu *et al.* [41] and Fruja [16] extends their work to larger fragments of CIL. In addition to proving a type safety result, Gordon and Syme use their executable specification in differential testing with a production implementation. They do not describe their test generation technique or any of the bugs it found (or missed), though they do suggest that such testing effort is worthwhile. Our experience with the Racket machine model supports this conclusion.

Dockins and Guyer present a verification system for the bytecode emitted by the YHC Haskell compiler [11], a target language more similar to Racket's than JVM bytecode or CIL. YHC bytecode executes on a stack machine with most instructions naming operands by their stack offsets. This machine provides explicit support for safe-for-space compilers, with its verifier rejecting references to cleared slots, but there is no analog of our no-clear annotations to enable optimizations.

Finally, Amadio *et al.* [1] extend the usual safety verification algorithm to also verify resource boundedness. Using techniques from term rewriting, they verify bytecode annotations to ensure termination and polynomial space usage for programs in a first-order functional language.

## 9 Conclusion

The randomized testing procedure described in Section 7 was a worthwhile tool. Applying the procedure was easy—after stating the desired properties using a mixture of Racket and Redex, we made just two simple changes to the test generator produced automatically from the bytecode grammar. Nevertheless, the procedure found more than more than two dozen bugs in the machine model, as well as a few in the machine implementation. We conjecture that a more sophisticated automated test generation technique would find the three known bugs that eluded our random tests, but even our naive technique easily paid for itself over the course of development.

More generally, our experience illustrates three ways in which testing, automated or otherwise, is a valuable technique in semantics engineering. First, mistakes abound while a semantics is in motion, and testing is a often low-cost way to find them. For example, at one point in the model's development, we made a global change to the structure of the verify function. By this point, we had already accumulated a large hand-written test suite, and so we felt confident in the change's correctness when the tests passed. To our surprise, though, randomized testing quickly found a case mishandled by our change and uncovered by our hand-written tests. After adding the case to our test suite and fixing the bug, we ran more random tests only to discover yet another mishandled case. After further augmenting the test suite and fixing this second bug, we ran more random tests, now certain that the change was finally correct. We were wrong for a third time. Second, it can be easy—and productive—to test the correspondence between an executable model and its implementation. Establishing such a correspondence by proof is almost always intractable when the implementation is the product of over 15 years of development. Third, while no replacement for formal proof, concerted testing does provide useful validation in the unfortunate situation where one's semantics engineering budget does not allow proof. For example, our random tests found 5 of 6 long-standing bugs in the bytecode verifier, some of them subtle.

**Acknowledgements** We thank John Reppy and the anonymous HOSC reviewers for their helpful comments.

The models used in this paper are available online:
`http://plt.eecs.northwestern.edu/racket-machine/`

## References


1. AMADIO, R. M., COUPET-GRIMAL, S., ZILIO, S. D., AND JAKUBIEC, L. A functional scenario for bytecode verification of resource bounds. In *International Workshop on Computer Science Logic* (2004), pp. 265–279.
2. APPEL, A. W. *Compiling with Continuations*. Cambridge University Press, 1992.
3. BERGHOFER, S., AND NIPKOW, T. Random testing in Isabelle/HOL. In *Proceedings of the International Conference on Software Engineering and Formal Methods* (2004), pp. 230–239.
4. CHENEY, J., AND MOMIGLIANO, A. Mechanized metatheory model-checking. In *Proceedings of the ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming* (2007), pp. 75–86.
5. CLAESSEN, K., AND HUGHES, J. QuickCheck: A lightweight tool for random testing of Haskell programs. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* (2000), pp. 268–279.
6. CLARKE, D. G., POTTER, J. M., AND NOBLE, J. Ownership types for flexible alias protection. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (1998), pp. 48–64.
7. CLINGER, W., AND REES, J. Revised report on the algorithmic language Scheme. *ACM SIGPLAN Lisp Pointers IV*, 3 (1991), 1–55.
8. CLINGER, W. D. Proper tail recursion and space efficiency. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation* (June 1998), pp. 174–185.
9. COGLIO, A., GOLDBERG, A., AND QIAN, Z. Toward a provably-correct implementation of the JVM bytecode verifier. In *Proceedings of the Workshop on Formal Underpinnings of Java* (1998), pp. 403–410.
10. COHEN, R. The defensive Java virtual machine specification. Tech. rep., Computational Logic Inc., 1997.
11. DOCKINS, R., AND GUYER, S. Bytecode verification for Haskell. Tech. rep., Tufts University Department of Computer Science, 2007.
12. ECMA. *Common Language Infrastructure (CLI), Standard ECMA–335*, 4th ed. European Association for Standardizing Information and Communication Systems, 2006.
13. FELLEISEN, M., FINDLER, R. B., AND FLATT, M. *Semantics Engineering with PLT Redex*. MIT Press, 2009.
14. FLATT, M., AND PLT. Reference: Racket. Tech. Rep. PLT-TR-2010-1, PLT Inc., 2010. `http://racket-lang.org/tr1/`.
15. FREUND, S. N., AND MITCHELL, J. C. A type system for the Java bytecode language and verifier. *Journal of Automated Reasoning 30*, 3–4 (2003), 271–321.
16. FRUJA, N. G. *Type Safety of C# and .NET CLR*. PhD thesis, ETH Zürich, 2007.
17. GOLDBERG, A. A specification of Java loading and bytecode verification. In *Proceedings of the ACM Conference on Computer and Communications Security* (1998), pp. 49–58.
18. GORDON, A. D., AND SYME, D. Typing a multi-language intermediate code. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2001), pp. 248–260.
19. GOSLING, J. Java intermediate bytecodes. In *Proceedings of ACM SIGPLAN Workshop on Intermediate Representations* (1995), pp. 111–118.
20. GUTTMAN, J. D., AND WAND, M., Eds. *VLISP: A Verified Implementation of Scheme*. Kluwer, Boston, 1995. Originally published as a special double issue of the journal *Lisp and Symbolic Computation* (Volume 8, Issue 1/2).
21. IGARASHI, A., PIERCE, B. C., AND WADLER, P. Featherweight Java: A minimal core calculus for Java and GJ. *ACM Transactions on Programming Languages and Systems 23*, 3 (2001), 396–450.
22. KLEIN, C., AND FINDLER, R. B. Randomized testing in PLT Redex. In *Proceedings of the Workshop on Scheme and Functional Programming* (2009).
23. LANDIN, P. J. The mechanical evaluation of expressions. *The Computer Journal 6*, 4 (1963), 308–320.
24. LEE, D. K., CRARY, K., AND HARPER, R. Toward a mechanized metatheory of Standard ML. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2007), pp. 173–184.
25. LEROY, X. Java bytecode verification: algorithms and formalizations. *Journal of Automated Reasoning 30*, 3–4 (2003), 319–340.

26. LINHOLM, T., AND YELLIN, F. *The Java Virtual Machine Specification*, 2nd ed. The Java Series. Prentice Hall PTR, 1999.
27. MCKEEMAN, W. M. Differential testing for software. *Digital Technical Journal of Digital Equipment Corporation 10*, 1 (1998), 100–107.
28. MILNER, R., TOFTE, M., HARPER, R., AND MACQUEEN, D. *The Definition of Standard ML (Revised)*. MIT Press, 1997.
29. NIPKOW, T., AND VON OHEIMB, D. Java light is type-safe—definitely. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (1998), pp. 161–170.
30. NORRISH, M. C formalized in HOL. Tech. rep., University of Cambridge, 1998.
31. QIAN, Z. A formal specification of Java virtual machine instructions for objects, methods and subroutines. In *Formal Syntax and Semantics of Java* (1999), Springer-Verlag, pp. 271–312.
32. ROBERSON, M., HARRIES, M., DARGA, P. T., AND BOYAPATI, C. Efficient software model checking of soundness of type systems. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages, and Applications* (2008), pp. 493–504.
33. SPERBER, M., DYBVIG, R. K., FLATT, M., VAN STRAATEN, A., FINDLER, R. B., AND MATTHEWS, J. *Revised [6] Report on the Algorithmic Language Scheme*. Cambridge University Press, 2010.
34. STÄRK, R., SCHMID, J., AND BÖRGER, E. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.
35. STATA, R., AND ABADI, M. A type system for Java bytecode subroutines. *ACM Transactions on Programming Languages and Systems 21*, 1 (1999), 90–137.
36. STEELE JR., G. L. Debunking the "expensive procedure call" myth; or, Procedure call implementations considered harmful; or, LAMBDA: The ultimate goto. Tech. Rep. 443, MIT Artificial Intelligence Laboratory, 1977. First appeared in the Proceedings of the ACM National Conference (Seattle, October 1977), 153–162.
37. SYME, D. *Declarative Theorem Proving for Operational Semantics*. PhD thesis, University of Cambridge, 1998.
38. SYME, D. Proving Java type soundness. Tech. rep., University of Cambridge, 2001.
39. WALKER, D., MACKEY, L., LIGATTI, J., REIS, G. A., AND AUGUST, D. I. Static typing for a faulty lambda calculus. In *Proceedings of the ACM SIGPLAN International Conference on Functional Programming* (2006), pp. 38–49.
40. YELLIN, F. Low level security in Java. In *Proceedings of the International World Wide Web Conference* (1995), pp. 369–379.
41. YU, D., KENNEDY, A., AND SYME, D. Formalization of generics for the .NET Common Language Runtime. In *Proceedings of ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (2004).