THE UNIVERSITY OF CHICAGO

# OPERATIONAL SEMANTICS FOR SCHEME VIA TERM REWRITING

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCES
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
JACOB MATTHEWS

CHICAGO, ILLINOIS
DECEMBER 2004

# ABSTRACT

Felleisen's notion of context-sensitive term rewriting systems a flexible tool for specifying operational semantics, but one that requires great care to use properly. We introduce PLT Redex, a program for exploring context-sensitive rewriting systems as executable specifications that makes using Felleisen's systems easier and less error-prone and allows researchers to apply software engineering principles to formal semantics systems. We also present an operational semantics for R5RS Scheme encoded in PLT Redex that is to our knowledge the first operational semantics for R5RS Scheme that covers the entire report and the most complete formal encoding of the language's semantics given in any style.

# TABLE OF CONTENTS

# LIST OF FIGURES

# CHAPTER 1
# PLT REDEX

[1] Since the late 1980s researchers have used context-sensitive term-rewriting systems as one of their primary tools for specifying the semantics of programming languages. They do so with good reason. Syntactic rewriting systems have proved to be simple, flexible, composable, and expressive abstractions for describing programming language behaviors. In particular, the rewriting approach to operational semantics described by Felleisen and Hieb [9] and popularized by Wright and Felleisen [25] is widely referenced and used.

Unfortunately, designing context-sensitive rewriting systems is subtle and error-prone. People often make mistakes in their rewriting rules that can be difficult to detect, much less correct. In this paper, we show how we have addressed this problem: after a brief tutorial on context-sensitive reduction semantics (section 1.1) and a history of its development (section 1.2), we present PLT Redex, a domain-specific language for context-sensitive reduction systems embedded in PLT Scheme (section 1.3). We then present a new model of the semantics of R$^5$RS Scheme developed with and encoded in PLT Redex (chapter 2). This semantics is to our knowledge the first operational semantics for R$^5$RS Scheme that models all the language features the R$^5$RS Scheme document's formal semantics models, and the most accurate formal semantics for R$^5$RS Scheme with respect to its informal specification given in any style. We will assume that readers of this paper know some Scheme, and some knowledge of R$^5$RS Scheme will be helpful but is not essential.

Before we discuss our results, it is useful to review Felleisen-style reduction semantics, and so we turn to that now.

---

1. An earlier version of this chapter appeared at RTA 2004 [17].

$$
\begin{array}{rcl}
e & ::= & e_1\ e_2 \mid v \mid x \\
v & ::= & \lambda\, x.\, e
\end{array}
$$

$$
(\lambda\, x.\, e)\, v \quad \rightarrow \quad e[x := v]
$$

Figure 1.1: The untyped $\lambda_v$-calculus as a rewriting system

## 1.1 Context-Sensitive Rewriting

The goal of any formal semantic system for programming languages is to present the rules of a programming language mathematically so that interesting properties can be proved about it. That is a broad goal, and there are many strategies for accomplishing it. One slight refinement of the goal, called small-step operational semantics, is to define a mathematical machine that describes what an idealized interpreter would produce when given any particular program in the language. Context-sensitive rewriting for operational semantics, usually called reduction semantics, is a way of realizing that goal.

Reduction semantics give meanings to programs by specifying rules for rewriting a program into an equivalent program and applying those rules successively until the program is in an irreducible form, which should be a final answer. For instance, a simple arithmetic language might have two rules, one saying that a program that multiplies two numbers is rewritten into the same program with that multiplication replaced by the numbers' product, and another specifying the same rule for addition. Then the term "3+4*7" would be rewritten to "3+28" by the first rule and that term would be rewritten to "31" by the second. No rewriting rule applies to the term "31," but it is a number so it can be considered the program's final answer.

A programming language can be modeled the same way. Figure 1.1 shows a specification of the untyped $\lambda_v$-calculus as a reduction system. The first two lines define the language's syntax in the usual way: an expression is either an application, an abstraction, or a variable reference. The last line is a specification of the $\beta_v$ relation, the rule that drives computation in the $\lambda_v$ calculus. It states that wherever $((\lambda\ x.\ E)\ v)$ appears as a subterm

$$e ::= e_1 e_2 \mid v \mid x \tag{1.1}$$
$$v ::= \lambda x.\, e \tag{1.2}$$
$$C ::= [\,] \mid C\ e \mid v\ C \tag{1.3}$$
$$C[(\lambda\ x.\ e)\ v] \rightarrow C[e[x := v]] \tag{1.4}$$

Figure 1.2: The $\lambda_v$-calculus with left-to-right evaluation

in any context $C$, the entire term can be rewritten as the same context with the application replaced by the abstraction's body where all instances of $x$ have been replaced by the argument.

That specification models the pure $\lambda_v$-calculus, but richer programming languages typically guarantee that terms are actually evaluated in a particular order. For instance, the $\lambda$-calculus-based languages Haskell, Standard ML, and Scheme all specify that the bodies of functions that are never called never get evaluated, which makes some programs yield values that otherwise would not. For that reason, systems for specifying programming language semantics have some mechanism for codifying the order in which evaluations take place. Context-sensitive reduction semantics gets its name for the way it handles this: in addition to reduction rules, a context-sensitive reduction system can specify partially-redundant parsing rules that decompose a term into a context and a subterm within it (called the contractum) where a reduction can occur. Reduction rules can be specified to apply only to terms that are divided this way, and can explicitly use the context and contractum to determine how that term should reduce.

For example, to more accurately reflect the semantics of modern $\lambda$-calculus-based languages we should restrict the untyped $\lambda_v$-calculus so that applications must be evaluated left-to-right and computation never takes place within the body of an abstraction. To do that, we can specify context-parsing rules that do not allow contexts to contain a hole within a $\lambda$-expression or an application in which a non-value expression appears to the left of an expression with a hole in it.

The system in figure 1.2 shows an encoding of those changes. It is identical to the system in figure 1.1 except for the addition of a new grammar production, $C$, for contexts. Now instead of $C[(\lambda x.e)v]$ matching an application anywhere in a term, the context in which it appears must match that production with the contractum appearing in place of $[\,]$.

With these changes, a term like $(\lambda x.xx)(\lambda x.xx)$ still fails to terminate: by matching it to $C$, it can be parsed into the empty context and a contractum that matches reduction rule 1.4, which will act on it to produce $(\lambda x.xx)(\lambda x.xx)$ again. However, consider the term $\lambda y.((\lambda x.xx)(\lambda x.xx))$, which also fails to terminate under the pure $\lambda$-calculus. It cannot be parsed using $C$ into anything but the empty context and the entire term, and by inspecting all the reduction rules it's easy to see that none of them apply to that decomposition. On the other hand, the term $(\lambda y.(\lambda x.xx)(\lambda x.xx))((\lambda q.q)(\lambda z.z))$ could be parsed into a context consisting of $(\lambda y.((\lambda x.xx)(\lambda x.xx)))[\,]$ and a contractum consisting of $(\lambda q.q)(\lambda z.z)$, which would reduce by rule 1.4 to $(\lambda y.(\lambda x.xx)(\lambda x.xx))(\lambda z.z)$ — that is, the same context with $(\lambda z.z)$ in place of the hole.

## 1.2 History

In his seminal paper [20] on the relationship among abstract machines, interpreters and the $\lambda$-calculus, Plotkin shows that an evaluator specified via an abstract machine defines the same function as an evaluator specified via a recursive interpreter. Furthermore, the standard reduction theorem for a $\lambda$-calculus generated from properly restricted reduction relations is also equivalent to this function. As Plotkin indicated, the latter definition is by far the most concise and the easiest to use for many proofs.

Figure 1.3 presents Plotkin's $\lambda_v$-calculus. The top portion defines expressions, values, and the two basic relations ($\beta_v$ and $\delta_v$). The rules below on the left are his specification of the strategy for applying those two basic rules in a leftmost-outermost manner.

In a 1989 paper, Felleisen and Hieb [9] develop an alternate presentation of Plotkin's $\lambda$-calculi. Like Plotkin, they use $\beta_v$ and $\delta_v$ as primitive rewriting rules. Instead of inference rules, however, they specify a set of evaluation contexts that correspond to the ones we introduced in figure 1.2 with the notion that placing a term in the hole is equivalent to

$$
\begin{array}{lcll}
e & = & (e\ e) \mid x \mid v & ((\lambda\ (x)\ e)\ v) \mapsto e[x := v] \quad (\beta_v) \\
v & = & (\lambda\ (x)\ e) \mid f & (f\ v) \mapsto \delta(f,v) \quad (\delta_v)
\end{array}
$$

**Plotkin**

$$\frac{e \mapsto e'}{e \to e'}$$

$$\frac{e \to e'}{(e\ e'') \to (e'\ e'')}$$

$$\frac{e \to e'}{(v\ e) \to (v\ e')}$$

**Felleisen/Hieb**

$C = [] \mid (v\ C) \mid (C\ e)$

if $e \mapsto e'$, then $C[e] \to C[e']$

Figure 1.3: Specifying an evaluator for $\lambda_v$

the textual substitution of the hole by the term [1]. The right side of the bottom half of figure 1.3 shows how they specify Plotkin's evaluator function with evaluation contexts.

While the two specifications of a call-by-value evaluator are similar at first glance, Felleisen and Hieb showed that their specification was more suitable for extensions with non-functional constructs (assignment, exceptions, control, threads, etc). Figure 1.4 shows how to extend the system of figure 1.3 (right) with assignable variables. Each program is now a pair of a store and an expression. The bindings in the store introduce the mutable variables and bind free variables in the expression. When a dereference expression for a store variable appears in the hole of an evaluation context, it is replaced with its value. When an assignment with a value on the right-hand side appears in the hole, the let-bindings are modified to capture the effect of the assignment statement. The entire extension consists of three rules, with the original two rules included verbatim. Felleisen and Hieb also showed that this system can be turned into a conventional context-free calculus like the $\lambda$-calculus.

Context-sensitive term-rewriting systems are well-suited for proving the type soundness of programming languages. Wright and Felleisen [25] showed how this works for imperative extensions of the $\lambda$-calculus and a large number of people have adapted the technique to other languages since.

$p = ((\mathit{store}\ (x\ v)\ \ldots)\ e)$
$e = \cdots \mathit{as\ before} \cdots \mid (\textbf{let}\ ((x\ e))\ e) \mid (\textbf{set!}\ x\ e)$
$PC = ((\mathit{store}\ (x\ v)\ \ldots)\ C)$
$C = \cdots \mathit{as\ before} \cdots \mid (\textbf{let}\ ((x\ C))\ e) \mid (\textbf{set!}\ x\ C)$

$((\textbf{letrec}\ (x_1\ v_1)\ \ldots\ (x_2\ v_2)\ (x_3\ v_3)\ \ldots)\ C[x_2]) \rightarrow ((\textbf{letrec}\ (x_1\ v_1)\ \ldots\ (x_2\ v_2)\ (x_3\ v_3)\ \ldots)\ C[v_2])$

$((\textbf{letrec}\ (x_1\ v_1)\ \ldots\ (x_2\ v_2)\ (x_3\ v_3)\ \ldots)\ C[(\textbf{set!}\ x_2\ v_4)]) \rightarrow ((\textbf{letrec}\ (x_1\ v_1)\ \ldots\ (x_2\ v_4)\ (x_3\ v_3)\ \ldots)\ C[v_4])$

$((\textbf{letrec}\ (x_1\ v_1)\ \ldots)\ C[(\textbf{let}\ ((x_2\ v_2))\ e)]) \rightarrow ((\textbf{letrec}\ (x_1\ v_1)\ \ldots\ (x_3\ v_2))\ C[e[x_2 := x_3]])$ where $x_3$ is fresh

if $e \mapsto e'$, then $PC[e] \rightarrow PC[e']$

Figure 1.4: Specifying an evaluator for $\lambda_S$

## 1.3 A Language for Specifying Context-Sensitive Rewriting

Context-sensitive rewriting systems are good for building complex systems, but large systems are still difficult to work with simply because it is difficult to remember every detail of a complicated structure or to identify every rewrite rule that could apply to a term as it becomes large. To manage this complexity, we have developed PLT Redex, a declarative domain-specific language for specifying context-sensitive rewriting systems. The language is embedded in MzScheme [11], an extension of $R^5RS$ Scheme. MzScheme is particularly suitable for our purposes for two reasons. First, as an extension of Scheme, its basic form of data includes S-expressions and primitives for manipulating S-expressions as patterns. Roughly speaking, an S-expression is an abstract syntax tree representation of a syntactic term, making it a natural choice for manipulating program trees. Second, embedding PLT Redex in MzScheme gives PLT Redex programmers a program development environment and extensive libraries for free.

The three key forms PLT Redex introduces are **language**, **reduction**, and *traces* (we typeset syntactic forms in bold and functions in italics). The first,

(**language** (<*non-terminal-name*> <*rhs-pattern*> ...) ...)

specifies a BNF grammar for a regular tree language. Each right-hand side is written in

PLT Redex's pattern language (consisting of a mixture of concrete syntax elements and non-terminals, roughly speaking). With a language definition in place, the **reduction** form is used to define the reduction relation:

(**reduction** <*language*> <*lhs-pattern*> <*consequence*>)

Syntactically, it consists of three sub-expressions: a language to which the reduction applies, a source pattern specifying which terms the rule matches, and Scheme code that, when evaluated, produces the resulting term as an S-expression. Finally, the function *traces* accepts a language, a list of reductions, and a term (in Scheme terms, an arbitrary S-expression). When invoked, it opens a window that shows the reduction graph of terms reachable from the initial term. All screenshots in this paper show the output of *traces*. The remainder of this section presents PLT Redex via a series of examples.

### *1.3.1 Example: $\lambda_v$*

Our first example is Plotkin's call-by-value $\lambda$-calculus augmented with numbers and addition. In figure 1.5 we give its definition in Felleisen and Hieb's notation on the left and in PLT Redex on the right.

The $\lambda_v$ language consists of abstractions, numbers, applications, sums, and variable references, and has only two rewriting rules. As figure 1.5 shows, the traditional mathematical notation translates directly into PLT Redex: each line in the BNF description of $\lambda_v$'s grammar becomes one line in **language**. The pattern (*variable-except* **lambda** $+$) matches any symbol except **lambda** and $+$. The $\alpha$-consistency of $\lambda_v$ is maintained by the rewriting rules, not the pattern matcher.

The reduction rules also translate literally into uses of the **reduction** form. The first reduction rule defines the semantics of addition. The pattern in the first argument to **reduction** matches expressions where a syntactic term of the form (**+** *number number*) is the next step to be performed. It also binds the pattern metavariables *c_1*, *number_1*, and *number_2* (identified by their textual representation as a nonterminal name followed by an underscore) to the context and $+$'s operands, respectively.

The second subexpression of **reduction** constructs a new S-expression where the addition operation is replaced by the sum of the operands, using Scheme's $+$ operator (numeric

$e = v \mid (e\ e) \mid (+\ e\ e) \mid x$
$v = (\lambda\ (x)\ e) \mid number$
$C = [\,]$
$\quad \mid (v\ C) \mid (C\ e)$
$\quad \mid (+\ v\ C) \mid (+\ C\ e)$
$x \in variables$

```
(define λv
  (language (e v (e e) (+ e e) x)
            (v (lambda (x) e) number)
            (C hole
               (v C) (C e)
               (+ v C) (+ C e))
            (x (variable-except lambda +)))

(reduction λv
  (in-hole C_1 (+ number_1 number_2))
  (replace (term C_1) (term hole)
           (+ (term number_1) (term number_2))))
```

$c[(+\ n_1\ n_2)]$
$\rightarrow c[n_1 + n_2]$

```
(reduction λv
  (in-hole C_1 ((lambda (x_1) e_1) v_1))
  (replace (term C_1) (term hole)
           (substitute (term x_1)
                       (term v_1)
                       (term e_1)))))
```

$C[(\lambda\ (x)\ e)\ v)]$
$\rightarrow C[e[x := v]]$

Figure 1.5: $\lambda_v$ semantics

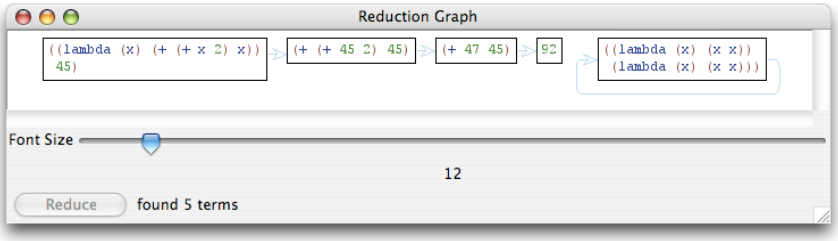



Figure 1.6: Reduction of a simple $\lambda_v$ term and of $\Omega$

constants in S-expressions are identical to the numbers they represent). The *in-hole* pattern is dual to the *replace* function. The former decomposes an expression into a context and a hole, and the latter composes an expression from a context and its hole's new content. The **term** form is PLT Redex's general-purpose tool for building S-expressions. Here we use it only to dereference pattern variables. The second reduction rule, $\beta_v$, uses the function

*substitute* to perform capture-avoiding variable substitution.[2] See figure 1.6 for an example term that reduces to 92 (left) and a term that diverges (right). Arrows are drawn from each term to the terms it can directly reduce to; the circular arrow attached to the $\Omega$ term indicates that it reduces to itself. By default, the *traces* function evaluates 10 reductions at a time and then pauses until the user presses the "Reduce" button, shown on the bottom left of the window.

### *1.3.2 Example: $\lambda_S$*

Figure 1.7 contains PLT Redex definitions for $\lambda_S$. The first reduction introduces a new notational convenience: it uses an ellipses pattern to match the subpatterns (*x_a v_a*) and (*x_b v_b*) any number of times including zero[3]. We use this feature here to select a distinguished element of the store: the matcher attempts to match *x_i* to every variable in the store and *v_i* to the corresponding value, but can only succeed in that match if the pattern variable *x_i*, which appears both in the store and in the expression, is identical in both places. Using the same subscript on two occurences of a pattern variable constrains S-expressions matched in the two places to be structurally identical, so the variable in the store and the variable in the term must be the same.

In figure 1.7 we also see **term** used to construct construct large S-expressions rather than just to get the values of pattern metavariables. In addition to treating pattern variables specially, **term** also has special rules for commas and ellipses. The expression following a comma is evaluated as Scheme code and its result is placed into the S-expression at that point. Ellipses in a **term** expression are duals to the pattern ellipses. The pattern before an ellipsis in a pattern is matched against a sequence of S-expressions and the S-expression before an ellipsis in a **term** expression is expanded into a sequence of S-expressions and spliced into its context.[4] Accordingly, the first rule produces a term whose store is identical

2. Currently, *substitute* must be defined by the user using a more primitive built-in form called **subst**. We intend to eliminate this requirement in a future version of PLT Redex.

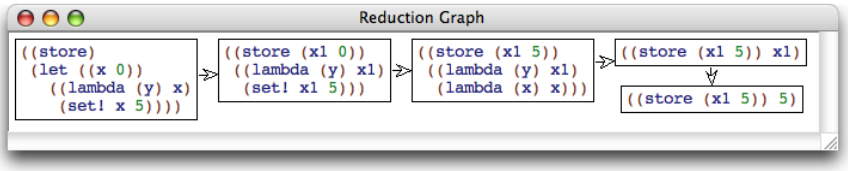
3. We use baseline-level ellipses (. . .) to indicate ellipses that literally appear in the program, and raised ellipses ($\cdots$) to indicate elided code.

4. With the exception of ellipsis and pattern variables, **term** is identical to Scheme's **quasiquote**.

```
(define λS
  (language
    (p ((store (x v) ...) e))
    (e ··· as before ···
       (let ((x e)) e)
       (set! x e))
    (x (variable-except
       lambda set! let))

    (PC ((store (x v) ...) EC))
    (EC ··· as before ···
        (set! x EC)
        (let ((x EC)) e)))
```

```
(reduction λS
    ((store (x_a v_a) ... (x_i v_i) (x_b v_b) ...)
     (in-hole EC_1 x_i))
    (term ((store (x_a v_a) ... (x_i v_i) (x_b v_b) ...)
           ,(replace (term EC_1) (term hole) (term v_i)))))
(reduction λS
    ((store (x_a v_a) ... (x_i v_old) (x_b v_b) ...)
     (in-hole EC_1 (set! x_i v_new)))
    (term ((store (x_a v_a) ... (x_i v_new) (x_b v_b) ...)
           ,(replace (term EC_1) (term hole) (term v_new)))))
(reduction λS
    ((store (x_a v_a) ...)
     (in-hole EC_1 (let ((x_i v_i)) e_1)))
    (let ((new-x (variable-not-in (term (x_a ...)) (term x_i))))
      (term ((store (x_a v_a) ... (,new-x v_i))
             ,(replace (term EC_1) (term hole)
                       (substitute (term x_i) new-x (term e_1)))))))
```

Figure 1.7: $\lambda_S$ semantics




acro

Figure 1.8: Reduction of a simple $\lambda_S$ term

to the store in the term it consumed.

The final rule also introduces another PLT Redex function, *variable-not-in*, which takes an arbitrary syntactic term and a variable name and produces a new variable whose name is similar to the input variable's name and that does not occur in the given term.

Figure 1.8 shows a sample reduction sequence in $\lambda_S$ using, in order, a **let** reduction, a **set!** reduction, a $\beta_v$ reduction, and a dereference reduction.

### *1.3.3 Example: Threaded $\lambda_S$*

We can add concurrency to $\lambda_S$ with surprisingly few modifications. The language changes as shown in figure 1.9. A program still consists of a single store, but instead of just one

```
(define t-λS
  (language
    (p (letrec ((x v) ...) (threads e ...)
    (PC (letrec ((x v) ...) TC))
    (TC (threads e ... EC e ...))
    ... as before ...)
```

Figure 1.9: Threaded $\lambda_S$

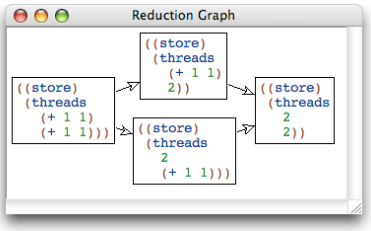



Figure 1.10: Multiple reductions

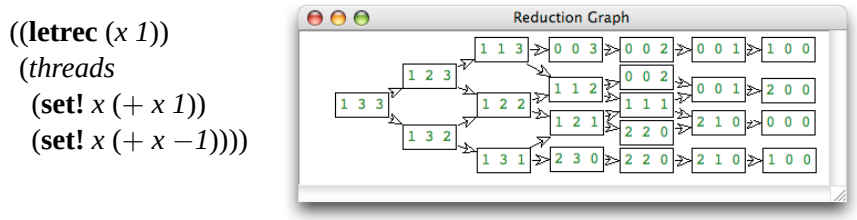
```
(letrec ((x 1))
 (threads
  (set! x (+ x 1))
  (set! x (+ x -1))))

(letrec ((x 1))
 (threads
  (set! x (+ 1 1))
  (set! x (+ x -1))))

(letrec ((x 0))
 (threads
  (set! x (+ 1 1))
  0))

(letrec ((x 2))
 (threads
  2
  0))
```

Figure 1.11: Interleaved evaluation

expression it now contains one expression per thread. In addition, each reference to *EC* in the $\lambda_S$ reductions becomes *TC*. No other changes need to be made and in particular no reduction rules need modification.

To express non-determinism, PLT Redex's pattern language supports ambiguous patterns by finding *all possible* ways a pattern might match a term. Consider the *TC* evaluation context in figure 1.9, which uses the selection idiom described in section 1.3.2. Unlike that example, nothing restricts this selection to a particular thread, so PLT Redex produces multiple matches, one for each reducible thread. The *traces* window reflects this by displaying all of the reductions that apply to each term when constructing the reduction graph, as shown in figure 1.10.

Owing to the possible interleaving of multiple threads, even simple expressions like the initial term in figure 1.11 reduce many different ways and gaining insight from a thicket of terms can be difficult. Accordingly, *traces* has an optional extra argument that allows the user to provide an alternative view of the term that can express a summary or just the salient part of a term without affecting the underlying reduction sequence. Figure 1.12 shows a summarized version of the reduction sequence of the initial term from figure 1.11. Rather than the entire term, each box only shows the number value of $x$, the number of steps left to take in the first thread, and the number of steps left to take in the second thread. With this




On the left, a threaded $\lambda_S$ term and on the right, boxes containing *x*'s value and the number of subexpressions remaining in each thread

Figure 1.12: Reduction summary using *traces*

summary, it is easy to see the possible reduction sequences at a high level without getting swamped with the details of each step.

## 1.4 Experience

PLT Redex has been useful for research and for pedagogy. On the pedagogical side, the University of Utah's graduate-level course on programming languages introduces students to the formal specification of languages through context-sensitive rewriting. Students model a toy arithmetic language, the pure $\lambda$-calculus, the call-by-value $\lambda$-calculus (including extensions for state and exceptions), typed $\lambda$-calculi, and a model of Java. In the most recent offering of the course, Matthew Flatt implemented many of the course's reduction systems using PLT Redex, and students used PLT Redex to explore specific evaluations. Naturally, concepts such as confluence and determinism stood out particularly well in the graphical presentation of reduction sequences. In the part of the course where an interpreter for the $\lambda$-calculus was derived through a series of "machines," PLT Redex was helpful in exposing the usefulness of each machine change.

As a final project, students implemented context-sensitive rewriting models from recent conference papers as PLT Redex programs. This exercise provided students with a much deeper understanding of the models than they would have gained from merely reading the paper. For a typical paper, students had to fill significant gaps in the formal content (*e.g.*, the figures with grammars and reduction rules). This experience suggests that paper authors

could benefit from creating a machine-checked version of a model, which would help to ensure that all relevant details are included in a paper's formalism.

In research, we found PLT Redex's visualization features useful for understanding the workings of language models and in helping us quickly discover bugs in them. We used it to develop the $R^5RS$ Scheme operational semantics we present in chapter 2 and found it invaluable: on numerous occasions a few minutes with the visualization tool exposed problems that might have taken us hours to find otherwise, if we found them at all. For instance, we took our semantics for $\lambda_{vs}$ through several revisions before settling on the form we present in section 2.4 and were able to quickly test and find flaws, make simplifying changes, and verify our intuitions by visualizing the reduction chains our draft systems induced on several example terms. While of course we do not know definitively how much time we saved and how many bugs we avoided by using PLT Redex instead of doing without tools, we suspect we saved considerable time with this approach due to the fact that many of our drafts contained subtle bugs that even knowledgable readers did not catch.

Another significant advantage of developing our reduction systems in PLT Redex was that since PLT Redex is a language embedded in PLT Scheme we could script PLT Redex execution and write test suites for our language models. This ability proved invaluable to our development effort. It seemed that every time we changed anything, everything broke — when we made changes to the model, we frequently made mistakes ranging from misspellings to misunderstandings of the ways features could interact that would break seemingly unrelated parts of the model. Having a quick way to see whether the system as a whole still worked when we modified or added a feature was a tremendous help. Furthermore, since the testing harness was just Scheme code we wrote we were able to modify it to check for custom error conditions that a prebuilt toolkit might not have been able to detect. For instance, we quickly discovered a problem with some of our models in which our specifications of grammar rules could accidentally find multiple ways to parse a term into the same context and contractum; when that happened it would slow the tool down because it had to consider the same decompositions multiple times, and in some cases this led to a substantial speed and memory penalty. We rectified the situation by extending our test harness to signal an error when a term reduced to the same result twice.

Of course PLT Redex is not an automated theorem prover and it does not prove anything

about the systems it works on. But in our experience having the ability to normalize terms in a language combined with visualization and scriptable testing dramatically improved our confidence in the systems we developed.

# CHAPTER 2
# A REDUCTION SEMANTICS FOR SCHEME

In this chapter we present an operational semantics for Scheme. We model the R$^5$RS Scheme programming language [15] at the same level as the denotational model presented in the Report. We will describe several of its most interesting features by showing smaller calculi that isolate those features within a simpler context.

We model R$^5$RS Scheme using a modified version of our $\lambda_S$ system (we could also say we start with Felleisen's $\lambda_v - CS$ calculus [8], which is roughly $\lambda_S$ plus *call/cc*). Every term in our model carries at its outermost level both a program and a store that maps names to values. Owing to R$^5$RS Scheme's relatively rich set of features compared to the $\lambda$-calculus, the R$^5$RS Scheme model uses more rules for decomposing a term into a context and a contractum than the $\lambda_S$ model. Fortunately, though, those rules are straightforward extensions with the exception of our extensions to model unspecified application order (section 2.1), equality-testing (section 2.3) and multiple-valued expressions (section 2.4). We leave presentation of our entire R$^5$RS Scheme semantics to appendix A.

## 2.1 Unspecified application order

In evaluating a procedure call, the R$^5$RS document deliberately leaves unspecified the order in which arguments are evaluated, but specifies that [15, section 4.1.3]

> *the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with* some *sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.*

In the formal semantics section, the authors explain how they model this ambiguity:

> *[w]e mimic [the order of evaluation] by applying arbitrary permutations* permute *and* unpermute *. . . to the arguments in a call before and after they are evaluated. This is not quite right since it suggests, incorrectly, that the order of evaluation is constant throughout a program . . .*. [15, Section 7.2].

Our rules, in contrast, capture the intended semantics using nondeterminism to select the argument to reduce.

In our first attempt to model this feature, we altered the $\lambda_S$ language from section 1.3.2 so that evaluation contexts could flow into any subexpression of an application using the rule $EC = (E \cdots EC\ E \cdots)$. This strategy does not work, as we saw when we modeled the resulting system in PLT Redex: we quickly discovered that terms like

(**letrec** ((*b2 1*))
((**set!** *b2* (− *b2*))
(**set!** *b2* (− *b2*))))

reduced in surprising ways, as figure 2.1 shows. This program should always reduce to the application of some unspecified value to some other unspecified value with the value of *b2* set to *1* in the store: no matter which subterm of the application ((**set!** *b2* (− *b2*)) (**set!** *b2* (− *b2*))) is reduced first, the result should be that *b2* is negated twice. But our model allows other interleavings. The problem is that the evaluator could interleave steps in multiple subexpressions to produce an outcome that could not be reached by any sequential ordering.

With that in mind, we developed a more sophisticated strategy that captures unspecified evaluation order without allowing nonsequential orderings. Figure 2.2 shows the necessary revisions to $\lambda_S$ to support R$^5$RS-style procedure applications (the full rule for R$^5$RS Scheme appears in the appendix). The basic idea is to use non-deterministic choice to pick a sub-expression to reduce only when we have not already committed to reducing some other subexpression. To achieve that effect, we introduce the non-terminal $inert$ and the notion of a marked expression, denoted with the $\bullet$ superscript. Marks identify a chosen expression: only marked expressions may be reduced, and only one reducible marked expression may appear in any application at one time. The $inert$ production stands for terms in which evaluation may not occur, *i.e.* unmarked expressions (those expressions we have not tried to evaluate yet) and marked values (those expressions we have already reduced all

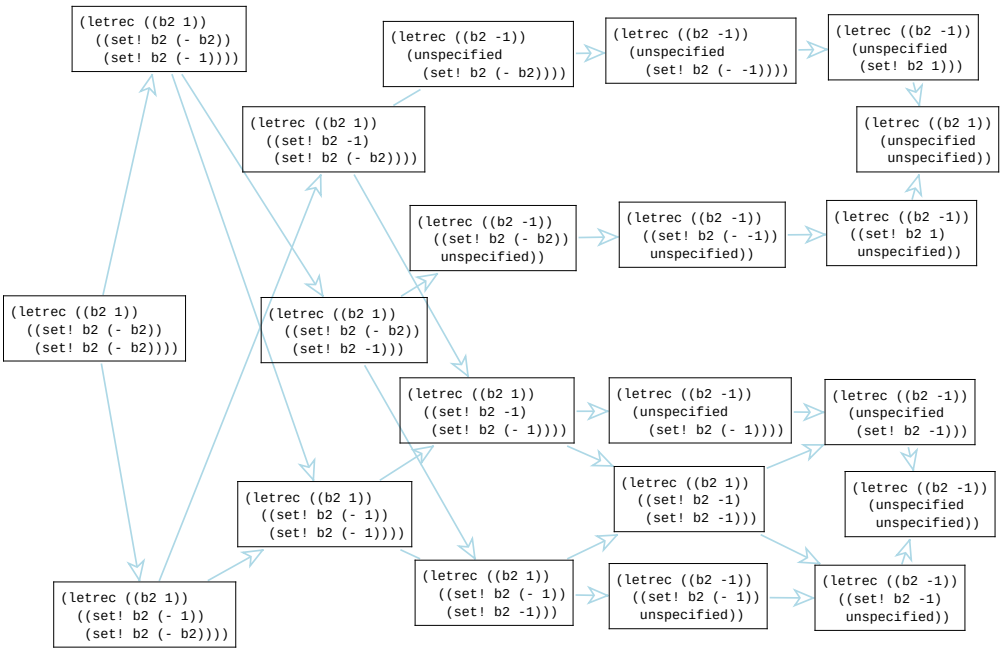



Figure 2.1: Interleavings possible with an erroneous unspecified-application-order model

$$
\begin{array}{rcl}
\mathit{inert} & = & v^\bullet \mid E \\
EC & = & (\mathit{inert} \cdots EC^\bullet \cdots)
\end{array}
$$

$$
\begin{array}{rcl}
\text{PC}[(\mathit{inert}_1 \cdots e \cdots \mathit{inert}_n)] & \rightarrow & \text{PC}[(\mathit{inert}_1 \cdots e^\bullet \cdots \mathit{inert}_n)] \\
\text{PC}[((\lambda\ (x_1 \cdots x_n)\ e)^\bullet\ v_1^\bullet \cdots v_n^\bullet)] & \rightarrow & \text{PC}[e[x_1 \cdots x_n := v_1 \cdots v_n]]
\end{array}
$$

Figure 2.2: Reduction rules for unspecified application order

the way to values). We also add a reduction rule that marks an arbitrary unmarked expression in an application on the condition that every other expression is inert, and we modify $\beta_v$ to apply only to fully-marked applications.

In addition to modifying evaluation contexts to add marking machinery and modifying the $\beta_v$-reduction rule to expect fully-marked applications, to model R$^5$RS Scheme we also modify the rules for the core built-in functions so that they can be reduced only when fully marked.

Figure 2.3 shows how this new system evaluates the term that our first attempt did not

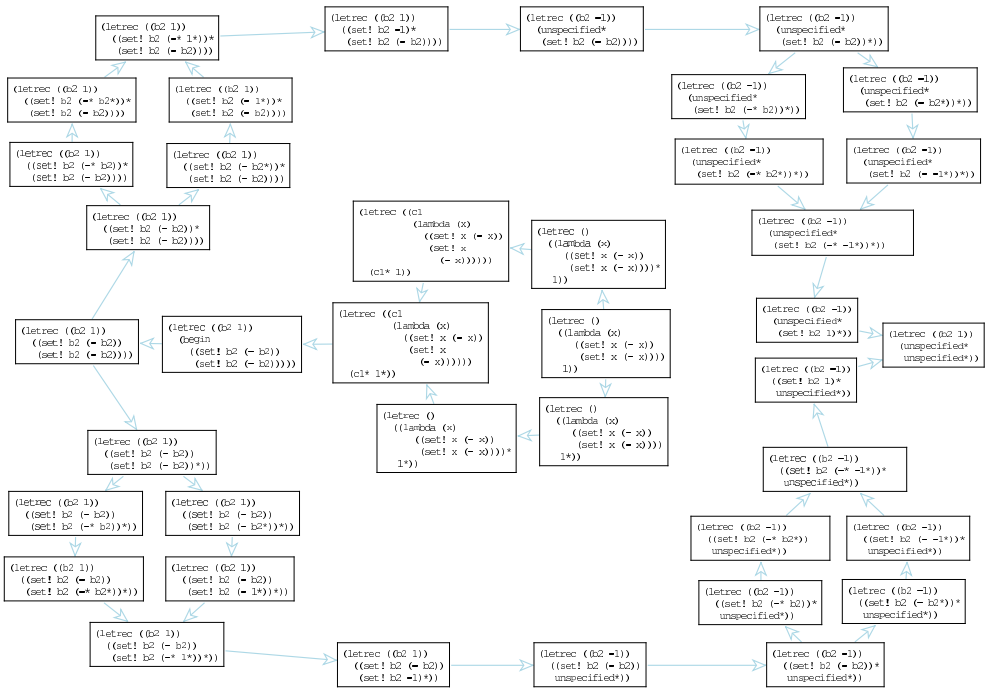


Figure 2.3: Evaluation in the unspecified-application-order model

evaluate correctly. That figure bears some explanation: the initial term,

(**letrec** ()
((**lambda** (*x*)
((**set!** *x* (− *x*))
(**set!** *x* (− *x*))))
*1*))

appears in the center. That term is an application, so it could evaluate the first argument to the application (the $\lambda$-expression) first, corresponding to the upward reduction from the first term, or it could evaluate the constant 1 first, corresponding to the downward reduction. From there it must evaluate the other argument, and in this case evaluation order does not matter so the two reduction paths converge. After that, the term can split again in

$$
\begin{array}{lcl}
p & ::= & (\textbf{letrec}\ ((x\ v)\cdots)\ e) \\
e & ::= & (e^m\ e^m\cdots) \\
 & | & (\textbf{let}\ ((x\ e))\ e) \\
 & | & (\textbf{set!}\ x\ e) \\
 & | & v \\
e^m & ::= & e \mid e^\bullet \\
v & ::= & (\lambda\ (x\cdots)\ e)
\end{array}
$$

$$
\begin{array}{lcl}
PC & ::= & (\textbf{letrec}\ ((x\ v)\cdots)\ C) \\
C & ::= & [\,] \\
 & | & (inert\cdots C^\bullet\ inert\cdots) \\
 & | & (\textbf{let}\ ((x\ C))\ e) \\
 & | & (\textbf{set!}\ x\ C) \\
 & | & (\textbf{begin}\ C\ e_1 \mid e_n)
\end{array}
$$

$$
\begin{array}{lcl}
inert & ::= & v^\bullet \mid e \\
PC[(inert\cdots e\ inert\cdots)] & \rightarrow & PC[(inert\cdots e^\bullet\ inert\cdots)] \\
PC[((\lambda\ (x\cdots)\ e)^\bullet\ v^\bullet\cdots)] & \rightarrow & PC[E[x\cdots/v\cdots]] \mid error \\
PC[(\textbf{begin}\ v\ e_1\ e_2\cdots)] & \rightarrow & PC[(\textbf{begin}\ e_1\ e_2\cdots)] \\
PC[(\textbf{begin}\ e)] & \rightarrow & PC[e] \\
(\textbf{letrec}\ ((x_1\ v_1)\cdots)\ C[(\textbf{let}\ ((x_i\ v_i))\ e)]) & \rightarrow & (\textbf{letrec}\ ((x_1\ v_1)\cdots(x_i'\ v_i))\ C[e[x_i'/x_i]])\ (x_i'\ \text{fresh}) \\
(\textbf{letrec}\ ((x_1\ v_1)\cdots(x_i\ v_i)\cdots)\ C[(\textbf{set!}\ x_i\ v_i')]) & \rightarrow & (\textbf{letrec}\ ((x_1\ v_1)\cdots(x_i\ v_i')\cdots)\ C[v_i']) \\
(\textbf{letrec}\ ((x_1\ v_1)\cdots)\ C[(\textbf{set!}\ x_i\ v_i')]) & \rightarrow & error\ (x_i\ \text{not in}\ \{x\cdots\})
\end{array}
$$

Figure 2.4: $\lambda_{!?}$, a language with assignment and unspecified application order

choosing whether to evaluate the first **set!** first or the second; the upward reduction from leftmost term in the center of figure is the path in which the first **set!** is reduced first and the downward reduction shows the opposite path. In both cases, there are two applications to be evaluated (the negations) and each branches off into two a left-first and right-first path and then converges after all expressions have been marked. The whole system eventually converges in the rightmost term at the center of the figure with the stored values of $x$ equal to 1 in all cases, as intended.

## 2.2 Progress for unspecified evaluation orders

Figure 2.4 shows the entire language we described in the previous section, which we will call $\lambda_{!?}$. One might suspect that the Church-Rosser diamond property holds for $\lambda_{!?}$ even though any individual term may reduce in various ways, and figure 2.3 lends credence to this suspicion. However, in fact $\lambda_{!?}$ does not have the diamond property. The following program shows the idea:

```
;; select-one : X Y → X ∪ Y
(define select-one
 (lambda (a b)
   (let ((ans (lambda (dummy) dummy)))
     ((lambda (x y) ans)
      (set! ans a)
      (set! ans b)))))
```

The function *select-one* takes two values *a* and *b* and returns one of them arbitrarily, making use of the fact that *ans* will be assigned to *b* if the arguments to the application on the last three lines are evaluated left-to-right and will be assigned to *a* if those arguments are evaluated right-to-left. That function indicates that we cannot even make the weaker claim that if a term $t$ reduces to a value when arguments are reduced one way then $t$ reduces to *some* value (rather than reducing to an error or diverging) under all possible orderings. This makes it clear:

```
;; call-one : (→ X) (→ Y) → X ∪ Y
(define call-one (lambda (t1 t2) ((select-one t1 t2))))
```

The *call-one* function calls one of its two argument thunks. Since the two thunks are unrestricted, it will only be a coincidence if an application of *call-one* has the same behavior under any evaluation order.

So we cannot prove the usual progress theorem (that any term any term either reduces to a unique answer or continues to compute forever) for $\lambda_{!?}$. Instead we will prove the weaker result that no term ever gets stuck. More formally:

**Theorem 2.2.1** $\forall t, t' \in \lambda_{!?}.closed(t) \wedge t \rightarrow t' \Rightarrow t' \in a \vee \exists t''.t' \rightarrow t''$.

Before we are prepared prove this theorem, we need the result that if a term reduces, any larger term that includes it in an evaluation context also reduces. Here and in the rest of this paper, we will use the metavariable $p$ to represent "progress terms", meaning either terms in the language or error terms (error terms are represented by the tag *error*; in the present semantics they can only arise from unclosed terms, but in later semantics more interesting runtime errors are possible), and we will use $a$ to represent answer terms, meaning terms recognizable as values[1] or error terms.

**Lemma 2.2.1 (context-stacking)** $\forall t, t_1 \in \lambda_{!?}$. *If* $t \equiv$ *(***letrec** *((*$x_1$ $v_1$*)* $\cdots$*)* $C_1[t_1]$*) and (***letrec** *((*$x_1$ $v_1$*)* $\cdots$*)* $t_1$*)* $\rightarrow p'$ *then* $\exists p.t \rightarrow p$.

**Proof** By the structure of $C$'s productions, all contexts must have the shape

$$C \rightarrow \cdots \rightarrow C \rightarrow [\,]$$

Observe that all reduction rules in $\lambda_{!?}$ are of the form (**letrec** (($x$ $v$) $\cdots$) $C[pattern_1]$) $\rightarrow$ $pattern_2$. So, since (**letrec** (($x$ $v$) $\cdots$) $t_1$) reduces, it must be possible to decompose $t_1$ into a context $C_2$ and a contractum $t_1'$, and furthermore for any other context $C_2'$, (**letrec** (($x$ $v$) $\cdots$) $C_2'[t_1']$) reduces.

Such a $C_2'$ must exist. $C_1[t_1] \equiv C_1[C_2[t_1']]$, and $C_1[C_2[\,]]$ is a legal context: the chain formed by $C_1$ necessarily ends with the production $C \rightarrow [\,]$ and the chain formed by $C_2$ is necessarily $C \rightarrow \cdots \rightarrow C \rightarrow [\,]$, so $C_1$'s final production could produce the head of $C_2$'s production chain rather than $[\,]$ and still be a legal production sequence. If we take this composed chain to be $C_2'$, $C_1[t_1] \equiv C_2'[t_1']$ and since (**letrec** (($x$ $v$) $\cdots$) $C_2'[t_1']$) reduces $C_1[t_1]$ reduces as well. □

**Lemma 2.2.2 (closedness-preservation)** $\forall t \in \lambda_{!?}$. *If* $t$ *is closed and* $t \rightarrow t'$ *then* $t'$ *is closed.*

---

1. Following Felleisen, we do not require that an answer term actually *be* a value, only that it be recognizable as the final result of the program.

**Proof** By reduction rule cases. Obvious in all cases. □

We can now prove the main lemma we will need, that every term reduces in one step to another term or an error.

**Lemma 2.2.3** $\forall t \in \lambda_{!?}$. *If* $t$ *is closed then* $\exists p.t \rightarrow p$ *or* $t \equiv$ **(letrec** *((*$x_1\ v_1$*)* $\cdots$*) v) for some* $v$.

**Proof** By induction on the structure of $t'$, where $t \equiv$**(letrec** $((x_1\ v_1) \cdots)\ t'$**)**.

1. If $t' \equiv v$ for some $v$, the theorem holds trivially.
2. If $t' \equiv (E_1^m\ E_2^m \cdots)$, then there are several subcases to consider.
   (a) If all $E^m$ subterms are of the form $v^\bullet$, then $\beta$-reduction or an error rule applies.
   (b) If there is some marked non-value expression subterm, then by the induction hypothesis that term reduces in the current store. Then by the definition of $C$ and lemma 2.2.1, $t$ reduces.
   (c) Otherwise, there must be no marked non-value expressions and at least one non-marked expression, so the expression-marking reduction applies.
3. If $t' \equiv$ **(let** $((x\ E_1))\ E_2$**)**, there are two possible cases. If $E_1$ is a value, then the **let**-reduction applies to it. If $E_1$ is not a value, then by the induction hypothesis it reduces and so by definition of $C$ and lemma 2.2.1 $t$ reduces.
4. If $t' \equiv$ **(begin** $E_1\ E_2 \cdots$**)**, then:
   (a) If $\#E_2 = 0$ then the rule taking **(begin** $E$**)** to $E$ applies (here and subsequently, the notation $\#NT_i$ will refer to the length of the sequence matched by $NT_i$ in the appropriate pattern).
   (b) If $\#E_2 \neq 0$ and $E_1$ is a value then the rule taking **(begin** $v\ E_1\ E_2 \cdots$**)** to **(begin** $E_1\ E_2 \cdots$**)** applies.
   (c) Otherwise, by induction hypothesis $E_1$ reduces and so by definition of $C$ and lemma 2.2.1 $t$ reduces.

5. If $t' \equiv$ (**set!** $x$ $E$), there are two subcases to consider. If $E$ is not a value, then by inductive hypothesis $E$ reduces and then by definition of $C$ and lemma 2.2.1 $t$ reduces. If $E$ is a value, then one of the two **set!**-reductions applies.

That exhausts all possibilities for $t'$, so the lemma holds. □

Now we can prove theorem 2.2.1. Here and subsequently, we will use the metavariable $a$ to represent "answers", either values or error terms.

**Proof** Since $\rightarrow$ is closedness-preserving by lemma 2.2.2 and all closed terms either reduce or are answers by lemma 2.2.3, $t'$ must either reduce or be an answer. □

## 2.3 Mutation and the store

To implement mutable bindings we take the approach used in Felleisen and Hieb's original paper [9] and introduce a top-level store that associates names with values as in the $\lambda_S$ language presented in section 1.3.2. However, that language only allows mutation of bindings. R$^5$RS Scheme, like most languages, allows structured values to be mutated as well. We model mutable structured values by defining their constructors — *e.g.* the *cons* function — not as syntactic constructors (as we do with immutable values such as numbers) but as reducible expressions that reduce to a fresh store variable that is bound in the store to an appropriate value. We model mutations to those values in the obvious way: performing *set-car!* on a *cons* cell in a given context becomes an unspecified value placed in the same context modified with the the mutation applied to the named cell.[2]

It may be surprising to see that in addition to bindings and pairs, the R$^5$RS Scheme model also moves closures into the store upon evaluation, since closures are not mutable in R$^5$RS Scheme. The reason we do that is so we can model R$^5$RS Scheme's *eq?* feature accurately: *eq?* is a constant-time comparison function that tests for equality of two values that relies on "pointer equality" to determine the equality of functions. (Technically the

2. With this change made, we could have done away with binding pointers altogether by introducing a new *cons* cell to hold the value for each binding and turning instances of **set!** into instances of *set-car!*, but we felt the method we used was a more direct model of R$^5$RS Scheme.

R$^5$RS Scheme specification requires that *eq?* returns *#t* when given pointer-equal values but does not require that pointer-unequal values cause *eq?* to return *#f*; we ignore that possibility here.)

## 2.4 Multiple return values

The R$^5$RS Scheme language provides a facility for expressions to evaluate to multiple or no values rather than just a single value. To model that feature, the report's formal semantics alters continuations so that instead of being functions that map a single value to a final answer, they become functions that map an arbitrary number of values to a final answer. However, with the exception of those created by *call-with-values*, no continuations should accept any number of values other than one. To ensure that, the formal semantics makes heavy use of a helper function called $single$ that performs the check in every appropriate place — in the argument positions of an application, in the test clause of an **if** statement, and so on. Rather than allowing only a single value by default and explicitly specifying exceptions to the rule as in the informal semantics, the formal semantics implicitly allows any number of values and the equations for each feature must impose extra constraints where only a single value is allowed. This strategy works, but does not mesh well with the informal semantics' intention.

This impedance mismatch may have been the cause of a subtle bug in the formal semantics. The R$^5$RS Scheme informal semantics mention that "except for continuations created with the *call-with-values* procedure, all continuations take exactly one value" [15, section 6.4], but the formal semantics does not enforce this restriction on sequences of expressions that are evaluated for effect, such as non-final body subexpressions inside a $\lambda$-expression. For that reason, the two definitions of Scheme's **begin** expression form given in the Report [15, section 7.3] are subtly incompatible with each other: the first requires that (**begin** (**values**) *1*) be an error, and the second requires that it not be.[3]

Our semantic model captures the difference between contexts that accept multiple values and contexts that reject multiple values more directly. Our strategy is distilled in $\lambda_{vs}$

3. The working draft of the R$^6$RS declares that intermediate expressions in **begin** statements may return multiple values, which will render this issue moot when R$^6$RS becomes final [6].

$$
\begin{array}{lcl}
E & ::= & (E\ E\ \cdots) \mid x \mid v \mid (\textbf{apply-values}\ E\ E) \\
v & ::= & (\lambda\ (x\ \cdots)\ E) \mid \textbf{values} \\
C_1 & ::= & [\,]_\circ \mid C \\
C_* & ::= & [\,]_* \mid C \\
C & ::= & [\,]_\circledast \mid (v\ \cdots\ C_1\ E\ \cdots) \mid (\textbf{apply-values}\ C_1\ E) \mid (\textbf{apply-values}\ v\ C_*)
\end{array}
$$

$$
\begin{array}{lcl}
C_1[(\textbf{apply-values}\ v_f\ (\textbf{values}\ v_{arg}\ \cdots))]_\circledast & \rightarrow & C_1[(v_f\ v_{arg}\ \cdots)] \\
C_1[((\lambda\ (x_1 \cdots x_n)\ E)\ v_1 \cdots v_n)]_\circledast & \rightarrow & C_1[E[x_1 \cdots x_n / v_1 \cdots v_n]] \\
C_1[((\lambda\ (x_1 \cdots x_n)\ E)\ v_1 \cdots v_m)]_\circledast & \rightarrow & \mathit{error}\ (n \neq m) \\
\\
C_1[v]_* & \rightarrow & C_1[(\textbf{values}\ v)] \\
\\
C_1[(\textbf{values}\ v)]_\circ & \rightarrow & C_1[v] \\
C_1[(\textbf{values}\ v\ \cdots)]_\circ & \rightarrow & \mathit{error}\ (\#v \neq 1)
\end{array}
$$

Figure 2.5: $\lambda_{vs}$, a $\lambda_v$-like language extended with multiple-value expressions

(figure 2.5). The $\lambda_{vs}$ language models $\lambda_v$ extended in three ways: first, it allows functions of arbitrary arity. Second, it adds a **values** form that represents multiple values and can be used as a higher-order function that produces multi-value results. Third, it adds a new syntactic form **apply-values** that accepts a function and any number of values and calls the function with those values as arguments. These new features are slightly different from the features R$^5$RS Scheme provides: R$^5$RS Scheme does not define **apply-values**, instead specifying a function *call-with-values* that evaluates its first argument as a thunk and applies the values it produces, in order, to its second argument as a function call. We use **apply-values** here because it clarifies the model and the two forms are are trivially interdefinable (we could write (*call-with-values thunk f*) as (**apply-values** *f* (*thunk*)) and (**apply-values** *f vs-expr*) as (*call-with-values* (**lambda** () *vs-expr*) *f*) if necessary). We will revisit *call-with-values* in section 2.6 and the appendix.

To model a language with multiple return values we introduce a modest generalization of the formalisms we have used so far in this paper that allows us to mix different kinds of evaluation in the same system. We extend the notation so that holes now have subscripts and the context-matching syntax $C[x]$ now takes the form $C[x]_i$ where $i$ matches the subscript of some hole. All holes regardless of subscript still match any term, but subscripted reduction rules only apply to a term decomposition if the hole used in that decomposition

has the appropriate subscript.

Figure 2.5 shows how to use this generalized formalism to model a language with multiple return values. The language $\lambda_{vs}$ is the usual call-by-value $\lambda$-calculus extended in three ways. First, it allows multiple-argument functions and applications. Second, it adds a special **values** function that returns all of its arguments. Finally, it adds the new syntax **apply-values**, which allows a programmer to apply a function to the values produced by an expression.

The interesting aspect of this model is how we have structured the evaluation contexts and reduction rules. Look at the definition of the nonterminal $C$ and the reduction rules subscripted with the $\circledast$ (pronounced "either") symbol. If you ignore the subscripts, these pieces themselves form a language with $\beta$-reduction and an **apply-values** form, and a **values** form. The only problem is that **apply-values** has a reduction rule that only allows it to reduce if its second argument is a **values** expression, but there is no guarantee that one will appear there. Similarly, the $\beta$-value reduction expects a function to be applied to a list of single values; if a **values** expression appears as an argument the term will be stuck. Normalizing a subterm in an application should yield a different result than normalizing the same subterm in the second position of an **apply-values** expression.

To address this problem, we could identify each separate position in which a single value is expected and add a reduction that converts (**values** $v$) to $v$ and (**values** $v_1\ v_2\ \cdots$) or (**values**) to an error. Similarly we could add rules for every position that expects multiple values that converted $v$ to (**values** $v$). That strategy is roughly speaking the one taken in the R$^5$RS Scheme formal semantics. Instead we introduce the notion of a $C_1$ context that expects a single value and a $C_*$ context that expects multiple values. Looking at the rules in figure 2.5 again, notice that the key distinguishing feature of $C_1$ is that $C_1$ can match $[\,]_\circ$ directly whereas $C_*$ and $C$ cannot. The two rules that apply only to terms that match $[\,]_\circ$ are "demotion" rules that convert a **values** expression to a value or an error. Thus, matching a term to a $C_1$ context guarantees that it will normalize to a single value, not a multiple-values expression (if it reduces to normal form at all). The opposite is true of $C_*$: since the single rule that applies to a term in a $[\,]_*$ hole "promotes" single values to **values** expressions, matching a term to a $C_*$ context guarantees that if it has a normal form it will

be a **values** expression rather than a single value. The $C$ context makes no such distinction: terms can normalize to a single value or to a values expression.

This way of modeling R$^5$RS Scheme's multiple-values feature has a major advantage over the method the R$^5$RS denotational semantics uses. Instead of spreading the rules for dealing with multiple-values appearing in a single-value context throughout the semantics (the R$^5$RS Scheme denotational semantics uses $single$ function eight times to restrict a continuation to accept only one argument), the semantics presented in figure 2.5 distill all rules related to multiple values into a single rule for identifying contexts that should receive them and three reduction rules explaining how to convert multiple values to single values and vice versa. No matter how many more features we add to the language, as long as those features do not concern multiple values we will not have to think about multiple values when we model them.

One could legitimately worry, though, that the advantage we gain using this modeling technique is outweighed by correspondingly harder proofs of properties of the $\lambda_{vs}$ model. We have some evidence to suggest that worry is unfounded. In the next section, we present a proof that all closed $\lambda_{vs}$ terms make progress. Our proof isolates the effects of multiple holes and multiple contexts and avoids considering reductions in $C_1$ contexts separately from reductions in $C_*$ contexts.

## 2.5 Proof of progress for $\lambda_{vs}$

In this section we show that $\lambda_{vs}$ always makes progress, in other words that our strategy for encoding values is not hopelessly incorrect.

Unfortunately, we will need a little machinery before we are prepared to prove the main theorem. Our extension introduces a new context in which the fundamental notion of irreducible terms changes. We have kept the rule that the end result of a computation should be a single value rather than a multiple-valued expression, but that decision is arbitrary: most Scheme systems, for example, allow a complete program to evaluate to any number of values and display them all as results (though this may technically violate the R$^5$RS Scheme specification). Though the decision is arbitrary, we have scattered its implications throughout $\lambda_{vs}$'s definition by specifying that each reduction rule has $C_1$ rather than $C_*$

as the context in which it applies. This will make it difficult to prove the result we want to prove, so we will need a little development to show that we can legitimately treat $C_1$ contexts as $C_*$ contexts and vice versa (except when the term in question is a value or a values expression).

**Definition 2.5.1** *The notation $\lambda_{vs}[C_*]$ refers to the language formed by replacing the $C_1$ contexts of the left-hand sides of all reduction rules in the definition of $\lambda_{vs}$ to $C_*$ contexts.*

We mentioned above that the choice to have all programs finally produce a single value was arbitrary; we will use $\lambda_{vs}[C_*]$ to represent the other choice we could have made, where all programs finally produce a multiple-values expression. In an effort to use consistent notation and as visual reminder when we are talking about evaluations that should produce a single value, we will refer to $\lambda_{vs}$ as $\lambda_{vs}[C_1]$ for the remainder of this proof.

**Definition 2.5.2** *The notation $t \xrightarrow{x} t'$ for $x \in \{C_1, C_*\}$ will mean that the term $t$ reduces in one step to $t'$ in $\lambda_{vs}[x]$.*

With that notation, we can prove that if a non-answer, non-multiple-values term is reducible in one language, it is reducible in both. (In this and all subsequent statements, we will assume where appropriate that $t$ is closed.)

**Lemma 2.5.1 (equivalence)** *$\forall t$, either $t \in a$, $t \equiv$ (**values** $v_1 \cdots$), or $t \xrightarrow{C_1} t' \Leftrightarrow t \xrightarrow{C_*} t'$.*

**Proof** By cases.

1. $t \in a$ or $t \in$ (**values** $v \cdots$): the lemma is trivially true.

2. $t$ has no reductions in either language: the lemma is trivially true. (We will not prove that this case cannot arise except as a restatement of the prior case, but it is nonetheless true.)

3. $t$ has a reduction in either language where the context is empty: then $t$ matched $[\,]_\circ$ in $\lambda_{vs}[C_1]$ and $[\,]_*$ in $\lambda_{vs}[C_*]$. All reductions rules that do not expect $v$ or (**values** $v \cdots$) in the hole will match regardless of the shape of the matched hole, so since a reduction is immediately possible in either $\lambda_{vs}[C_1]$ or $\lambda_{vs}[C_*]$ it is possible in both.

4. $t$ has a reduction involving a nonempty context in one language: since both $C_1$ and $C_*$ match only the appropriate hole and $C$, and since $C$ is exactly the same for $\lambda_{vs}[C_1]$ and $\lambda_{vs}[C_*]$, any way of decomposing $t$ into a nonempty context and a hole in either language works equally well for both languages. Thus any term that can be decomposed in such a way that a rewriting rule applies in one language can be decomposed that way in either language.

Thus the lemma holds in all cases, concluding the proof. □

Now we need a little more machinery that will aid us in proving an important lemma by induction. We need to show that if a term $t$ can reduce in an empty context, it can also reduce in a nonempty context.

**Lemma 2.5.2 (context stacking)** *If* $t \equiv C_1[r]_1$ *and* $r \xrightarrow{C_1} p'$ *then* $\exists p.t \xrightarrow{C_1} p$. *Similarly if* $t \equiv C_1[r]_*$ *and* $r \xrightarrow{C_*} p'$ *then* $\exists p.t \xrightarrow{C_1} p$.

**Proof** Since $r \xrightarrow{C_1} p'$, $r \equiv C_1'[t'']$ for some $C_1'$ and $t''$. Thus $t \equiv C_1[C_1'[t'']]_1$. The only way $C_1$ could match a term with $[\,]_\circ$ is with the production $C_1 \equiv [\,]_\circ$, and by construction $C_1$ produces $r$ as well. Thus we can build a larger context consisting of all the productions from $C_1$ to $r$ followed by all the productions from $r$ to $t''$ and have $t \equiv C_1[t'']$. Since all reduction rules in $\lambda_{vs}$ match a subterm within any context and some rule reduced $C_1'[t'']$, that same rule reduces $C_1[t'']$.

The multiple-values case proceeds by exactly the same argument, replacing $[\,]_*$ for $[\,]_\circ$ and $C_*$ for $C_1$ where appropriate. □

Two more lemmas concern the nature of reductions: one tells us that reducing a closed term does not produce an open term, and the other tells us that for any term at most one reduction applies.

**Lemma 2.5.3 (closedness preservation)** $\forall t \in E$. *if* $closed(t)$ *and* $t \to t'$ *then* $closed(t')$.

**Proof** By cases on the reduction rule that reduces $t$ to $t'$. The proof is immediate in every case. □

**Lemma 2.5.4 (unique reduction)** $\forall t \in E$. *if* $t \xrightarrow{C_1} p$ *and* $t \xrightarrow{C_1} p'$ *then* $p \equiv p'$.

**Proof** For any term, it is easy to see that there is a unique path through $C$-productions that leads to a hole in which reduction could possibly take place: for each syntactic form in the language, contexts restricted so that there can only be multiple context productions for that form if all but one are values. For instance, a term of the form (**apply-values** $v$ $E$) matches $C$ in two ways as (**apply-values** $[\,]_\circ$ $E$) and (**apply-values** $v$ $[\,]_*$), but the first decomposition is guaranteed to have an irreducible term in the hole.

So if the lemma does not hold, it must be that $t$ contains some subterm $t_1$ such that $t \equiv C_1[t_1]_x \rightarrow p$ and $t_1 \equiv C_x[t_2]$ where $t \equiv C_1[C_x[t_2]] \rightarrow p'$. But since $C_1[t_1]_x \rightarrow p$, $t_1$ must be a term in the left-hand side of some reduction rule and $x$ must be that reduction rule's subscript. It is immediate in all cases of the left-hand sides of reduction rules that $C_x$ must be the empty context, thus that $t_1 \equiv t_2$, and thus that $p \equiv p'$. □

**Lemma 2.5.5 (progress)** $\forall$ *closed* $t \in \lambda_{vs}[C_1]$, *either* $\exists p.t \rightarrow p$ *or* $t \in v$.

**Proof** By induction on the structure of $t$.

1. If $t \equiv v$ for some value $v$, the theorem holds trivially.

2. If $t \equiv (E_1\ E_2 \cdots E_n)$, then either all subexpressions are values, in which case either one of the $\beta_v$ rules or the **values** rule applies, or one of the expressions is not a value. If one of the expressions is not a value, then we can decompose it using the productions $C_1 = C$ and $C = (v \cdots C_1\ E \cdots)$ where the occurence of $C_1$ in the second production matches a non-value subexpression $E_{sub}$ of $t$. Since $E_{sub}$ is not a value expression by construction, $E_{sub} \xrightarrow{C_1} p'$ for some $p'$ by inductive hypothesis. Then by lemma 2.5.2, $\exists t'.t \rightarrow p$.

3. If $t \equiv$ (**apply-values** $E_1\ E_2$), there are four cases:

(a) If $E_1$ is not a value, then by definition of $C_1$ we have $t \equiv$ (**apply-values** $[\,]_\circ\ E_2$) and by inductive hypothesis we have $E_1 \xrightarrow{C_1} p'$. Thus by lemma 2.5.2 $\exists p.t \xrightarrow{C_1} p$.

(b) If $E_1$ is a value but $E_2$ is neither a value nor (**values** $v \cdots$), then by inductive hypothesis $E_2 \xrightarrow{C_1} p'$, then by lemma 2.5.1 $E_2 \xrightarrow{C_*} p'$, and finally by lemma 2.5.2 $t \xrightarrow{C_1} p$.

(c) If $E_1$ is a value and $E_2$ is a value, then the promotion rule applies to it and it is easy to see that $t \xrightarrow{C_1}$ (**apply-values** $E_1$ (**values** $E_2$)).

(d) If $E_1$ is a value and $E_2$ is (**values** $v \cdots$), then the reduction rule for **apply-values** applies to it and $t \xrightarrow{C_1}$ ($E_1\ v \cdots$).

Those are all the possible cases, concluding the proof. □

We are now prepared to prove that every closed $\lambda_{vs}$ term either reduces to a unique value or continues reducing forever. Here we use $\rightarrow$ to indicate the transitive closure of $\lambda_{vs}[C_1]$'s $\rightarrow$ relation.

**Theorem 2.5.1** $\forall$ *closed* $t \in E$. *either there exists a unique* $a \in v \cup \{wrong\}$ *such that* $t \rightarrow a$ *or* $\forall t'$ *such that* $t \rightarrow t'$. *there exists a unique* $t''$ *such that* $t' \rightarrow t''$.

**Proof** Assume the property does not hold. Then there must exist a non-value $t'$ such that $t \rightarrow t'$ but $t' \not\rightarrow a$ for any $a$ or $t' \rightarrow \{a_1, a_2\}$ for distinct $a_1$ and $a_2$. But since the $\rightarrow$ relation is closedness-preserving by lemma 2.5.3 and $t$ is closed, $t'$ is closed. Therefore by lemmas 2.5.4 and 2.5.5 $t' \rightarrow a$ for exactly one $a$. This is a contradiction, so no such $t'$ can exist and the theorem holds. Thus the property holds, completing the proof. □

## 2.6 From apply-values to call-with-values

The three features outlined in sections 2.1, 2.3, and 2.4 together make up the most interesting new features of the R$^5$RS Scheme semantics we present in appendix A. The other features of our R$^5$RS Scheme model not present in the smaller $\lambda$-calculus-based models

we have presented in earlier chapters include *call/cc*, scheme—if—, scheme—cons— lists, and so-called "$\mu - \lambda$" functions that can be invoked with any number of arguments beyond a specified minimum and represent all extra arguments as a list. We borrow our model of *call/cc* feature from Felleisen's dissertation [7], **if** is straightforward, and our model of*cons* lists and $\mu - \lambda$s are involved but uninteresting.

One interesting issue does arise when adapting our multiple return values technique to full Scheme. Recall that the language we describe in section 2.4 lets programmers usefully receive multiple-value returns with a special syntactic form called **apply-values**. As we mentioned in that section, R$^5$RS Scheme does not have an **apply-values** form and instead uses a built-in function called *call-with-values* that takes a "sender" thunk and a "receiver" function of any arity and applies the receiver to the values produced by the sender. We did not model *call-with-values* in section 2.4 because it obscures the essential idea that makes multiple return values work, but to be a faithful representation of R$^5$RS Scheme we do model it in our full semantics. To achieve that, we divide contexts into single-value and multiple-value contexts as in section 2.4, but the production we add for *call-with-values* is different from the production we used for **apply-values**:

$$C = (\textit{call-with-values}^\bullet\ EC*^{cwv}\ v^\bullet) \mid \cdots$$

That is, a *call-with-values* application must be fully-marked (to satisfy the unspecified evaluation order requirement explained in section 2.1), but with a new mark indicated with the superscript $cwv$ appearing in its middle term. Only expressions with this new mark can a multi-valued evaluation take place. The mark is produced and consumed by the reduction rules for *call-with-values* applications (abbreviated *cwv* here), which are otherwise much like **apply-values** rules:

$$\begin{array}{lcl} C[(\textit{cwv}^\bullet\ v_p^\bullet\ v_f^\bullet)] & \rightarrow & C[(\textit{cwv}^\bullet\ (v_p)^{cwv}\ v_f^\bullet)] \\ C[(\textit{cwv}^\bullet\ (\textbf{values}^\bullet\ v_1^\bullet \cdots)^{cwv}\ v_f^\bullet)] & \rightarrow & C[(v_f^\bullet\ v_1^\bullet \cdots)] \end{array}$$

The *call-with-values* function seen from this perspective is just a variant of **apply-values** that evaluates in two phases, evaluating its first argument in a single-valued context and then evaluating an invocation of the result in a multi-valued context.

# CHAPTER 3
# CONCLUSION

We have presented an introduction to and a history of context-sensitive reduction semantics, presented PLT Redex, an automated tool for experimenting with context-sensitive reduction systems, and have presented a semantics for R$^5$RS Scheme using context-sensitive reduction semantics developed using PLT Redex. To our knowledge, our R$^5$RS Scheme semantics formalizes more of the language than any other existing semantics for the language and shows how to model R$^5$RS Scheme-style multiple return values for the first time in an operational semantics and gives a novel and particularly aesthetically appealing model for unspecified sequential evaluation orders that takes advantage of the nondeterministic choice inherent in term-rewriting systems.

Our implementation of PLT Redex and the source code for our model of R$^5$RS Scheme are available for download at `http://www.cs.uchicago.edu/~jacobm/masters/`.

## 3.1 Related Work

Reduction semantics has been used to model large programming languages many times and in many different ways. Felleisen's dissertation, which introduced context-sensitive reduction semantics, gives a formulation of a substantially smaller language than the one we present here that he calls "idealized Scheme" [7], and Felleisen extends that model into the $\lambda$-$v$-$CS$ calculus in later work [8]. Since then, reduction semantics have been used to model the cores of many languages including Emacs Lisp [19], MultiLisp [10], Java [12], ML [14, 25] and Concurrent ML [22] among many others. Harper and Stone present a formal semantics for Standard ML that includes a dynamic semantics encoded using a variation on Wright and Felleisen's notation; it is the largest example of a programming language semantics given in a variant of reduction semantics we have found in the literature (with the possible exception of our own semantics for R$^5$RS Scheme).

There has also been extensive work on the semantics of Scheme. Clinger presented an operational semantics for core Scheme in the development of the notion of space efficiency [5]. Ramsdell presented a structural operational semantics for Scheme aimed at fixing the unspecified order of argument evaluation problem we discuss in section 2.1 [21]. His model is less complete than ours (most notably, it does not include multiple return values) and is tied much more closely to the R$^5$RS Scheme formal semantics.

Many researchers have implemented programs similar to our reduction tool. For example, Elan [2], Maude [3], and Stratego [24] all allow users to implement term-rewriting systems (and more), but are focused more on context-free term-rewriting. Stratego's strategies can be thought of as a generalization of the evaluation context notion. The ASF+SDF compiler [23] is very similar to PLT Redex but is geared towards language implementation rather than exploration and so makes tradeoffs that do not suit the needs of lightweight debugging (but that make it a better tool for building efficient large-scale language implementations). The CENTAUR system [4] is a toolbox for automatically generating language implementations based on specifications, but it is directed more towards static semantics than dynamic semantics.

Our reduction tool is focused on context-sensitive rewriting and aims to help its users visualize and understand rewriting systems rather than employ them for some other purpose. The **in**$^2$ graphical interpreter for interaction nets [16] also helps its users visualize sequences of reductions, but is tailored to a single language.

## 3.2 Future Work

We plan to extend PLT Redex to allow simple ways to express the binding structure of a language, which will allow us to synthesize capture-avoiding substitution rules automatically. We also plan to add more support for the reduction rules commonly used in the literature, such as source patterns that match only if other patterns did not.

We have identified several avenues for extensions to our model for R$^5$RS Scheme. First, though it is roughly as complete as the formal semantics given in R$^5$RS Scheme Report, it covers far less than the Report describes informally. We believe there are significant and interesting challenges in extending our semantics to cover the entire R$^5$RS Scheme Report:

macros, *eval*, *dynamic-wind*, and the top-level interaction environment, for example, are all left undescribed by our system and by the R$^5$RS Scheme semantics.[1] We believe that a model of R$^5$RS Scheme as complete as the formal definition of Standard ML [18] is feasible in our framework and that it would be a benefit to the Scheme community.

1. Gasbichler, Knauel, Sperber, and Kelsey have presented a semantics for *dynamic-wind* in a denotational style [13].

# APPENDIX A
# OPERATIONAL SEMANTICS FOR R$^5$RS SCHEME

---

The following code is an executable specification of R$^5$RS Scheme encoded in PLT Redex. It is presented here divided into sections and annotated, and is also available for download (along with the PLT Redex tool itself and executable models for all the calculi in this paper) at `http://www.cs.uchicago.edu/~jacobm/masters/`.

The code is presented with commentary in-line. Executable code is typeset in a fixed-width font and commentary is typeset in a variable-width font.

## A.1 Preliminaries

Before proceeding to the code, we say a few words about the style of our semantics and places where it differs from the semantics presented in the R$^5$RS Scheme formal specification. There are many expressions whose return values are explicitly unspecified in the R$^5$RS Scheme document — for instance, the result of a **set!** expression. Since this specification is intended to be executable, we modeled unspecified results with a special value *unspecified* that has no associated reduction rules and will stick any program that inspects it. If we did not intend for our model to be executable, we could add the rule schema $\forall v.PC[unspecified] \rightarrow PC[v]$.

We also chose to elide reductions that indicate out-of-memory errors. These would be easy to add back in at the expense of a little extra clutter when visualizing traces: reductions from each allocating in any program context to the "out of memory" error would be all that was necessary.

Another guiding principle of these semantics was to introduce a minimum of "scaffolding" constructions that were not legal typable Scheme syntax and to always treat syntactic

constructs with their regular meanings so that any intermediate term produced by the semantics was a legitimate executable Scheme program that when run had the same result the semantics would give it. (Of course the initial term fed in to the system has this property, but it is easy to construct systems in which the reduction rules introduce illegal syntax or misuse syntax in ways that other rules understand — it is that kind of system we want to avoid.) We compromised on this point in three ways: our pair representation (see section A.5.3), our marking scheme for applications (see sections 2.1 and A.5.4) and *call-with-values* (see sections 2.6 and A.5.6), and our *call/cc* model (see section A.5.5) contain violations of this principle. All but one are fixable; see the discussion in those sections for more details.

Now we are ready for the code.

---

```
(module r5rs mzscheme
  (require (lib "reduction-semantics.ss" "reduction-semantics")
           (lib "subst.ss" "reduction-semantics")
           (prefix srfi1: (lib "1.ss" "srfi")))

  (provide lang reductions)
```

---

## A.2 Grammar

The R$^5$RS Scheme grammar is much larger than the grammars presented in the rest of this paper but not much more complicated. Expressions can now take on several additional syntactic forms such as **if** and **begin** and there are many more built-in functions such as *set-car!* and *call/cc*, but these extensions are all straightforward. Evaluation contexts are extended in the same straightforward way from the evaluation contexts described in section 2.4.

One somewhat subtle extension we make to the systems described so far the distinction between different kinds of variables. Variables representing code and value pointers are thought of as values, and are included in the values productions. Other variables (in particular those representing binding pointers) are thought of as reducible expressions. For that reason we partition variable names into separate areas for these kinds of values and

they can be thought of as tagged names or memory allocated from distinct pools. We encode the distinction by using different prefixes for different kinds of pointers: *b* for binding pointers (see section A.5.2), *p* for pair pointers (section A.5.3), *c* for code pointers (section A.5.4), and *m* for $\mu$-$\lambda$ pointers (also section A.5.4). To specify these partitions in PLT Redex we use a feature of PLT Redex's pattern language we have not yet described: in a PLT Redex pattern, (*side-condition pattern scheme-expr*) matches if *pattern* matches and the associated Scheme expression evaluates to a true value.

---

```
(define lang
  (language (p (letrec ((x sv) ...) e))

            (e (e e ...)
               (if e e e)
               (set! x e)
               (begin e e ...)
               (abort e)
               lam
               mulam
               v
               (side-condition
                x_1
                (and (not (prefixed-by? (term x_1) 'c))
                     (not (prefixed-by? (term x_1) 'p)))))

            (lam (lambda (x ...) e e ...))
            (mulam (lambda (x ... dot x) e e ...))

            (pc (letrec ((x sv) ...) ec1))

            (ec hole
                (inert ... (mark ec1) inert ...)
                (if ec1 e e)
                (set! x ec1)
                (begin ec e e ...)
                ((mark call-with-values)
                 (cwv-mark ec*)
                 (mark v)))

            (ec1 (hole single) ec)
            (ec* (hole multi) ec)

            (inert e (mark v))

            (sv v
                (cons v v)
                lam
                mulam)

            (v fun nonfun)

            (fun cp      ; user functions
                 mp
                 cons    ; primitive functions
```

```
      null?
      cons?
      car cdr set-car! set-cdr!
      + - / *
      call/cc
      values call-with-values
      eq?
      apply)

(nonfun pp
        number
        null
        true
        false
        unspecified)

(x (variable-except
    lambda if dot loc set!    ; core syntax names
    mark begin

    null true false           ; non-function values
    unspecified
    pair closure

    error                     ; signal an error

    cons cons? null? car cdr  ; list functions
    set-car! set-cdr!
    + - * /                   ; math functions
    call/cc abort             ; call/cc functions
    values call-with-values   ; values functions
    cwv-mark))

; binding pointer
(bp (side-condition x_bp (prefixed-by? (term x_bp) 'b)))
; pair pointer
(pp (side-condition x_pp (prefixed-by? (term x_pp) 'p)))
; pointer to a lambda function
(cp (side-condition x_pp (prefixed-by? (term x_pp) 'c)))
; pointer to a mu-lambda function
(mp (side-condition x_mp (prefixed-by? (term x_mp) 'm)))))
```

---

## A.3 Relations

It will be useful in specifying our reduction rules to make some distictions among several different styles of reduction. In this section we introduce four relations specific to this semantics, all of which we will find use for in the next section. These new relations are intended to clarify and categorize different kinds of reduction found in the system; they are all expressible as normal term reductions in the style of the rest of this paper and in

fact are defined as small macros that expand to uses of the **reduction** syntax described in chapter 1.3.

The four reductions we will use are `-->`, used to indicate the normal reduction relation (equivalent to our uses of → elsewhere in this paper), `/->`, used to indicate an "in-place" reduction, `*->`, used to indicate an application, and `e->` used to indicate a reduction that signals an error.

Note that the PLT Scheme reader treats the expression (*a* . → . *b*) as identical to (→ *a b*). We use this as a cheap way to use infix notation for clarity.

---

```
;; --> : one-step reduction, full term to full term.
(define-syntax (--> stx)
  (syntax-case stx ()
    [(_ term result)
     #'(reduction lang term result)]))

;; /-> : one-step reduction, pc[term] to pc[term]
(define-syntax (/-> stx)
  (syntax-case stx ()
    [(_ term result)
     #'(reduction/context lang pc term result)]))

;; *-> : one-step reduction, pc[marked application] to pc[term]
(define-syntax (*-> stx)
  (define (src-item->result-item stx)
    (syntax-case stx (...)
      [... stx]
      [other #'(mark other)]))
  (syntax-case stx ()
    [(_ (item ...) result)
     (with-syntax ([(result-item ...)
                    (map
                     src-item->result-item
                     (syntax-e #'(item ...)))])
       #'((result-item ...) . /-> . result))]))

;; e-> : error reduction, pc[term] -> error
(define-syntax (e-> stx)
  (syntax-case stx ()
    [(_ t msg) #'(reduction lang
                            (in-hole pc t)
                            (term (error msg)))]))
```

---

## A.4 Primitives

Because our model does not take into account R$^5$RS Scheme's numeric tower, we model its numeric operations in terms of the host language's underlying functions. In this section we introduce syntax that automates that process.

---

```
;; prims : procedure ... -> listof reductions
(define-syntax (prims stx)
  (syntax-case stx ()
    [(_ oper ...)
     (andmap identifier? (syntax->list #'(oper ...)))
     #'(list
        ((oper number_1 (... ...))
         . *-> .
         (apply oper (term (number_1 (... ...)))))
        ...
        ((side-condition
          ((mark oper) (mark v_1) (... ...))
          (not (andmap number? (term (v_1 (... ...))))))
         . e-> .
         "attempt to apply numeric operator to non-numbers")
        ...)]))
```

---

## A.5 Reductions

In this section we define the reduction steps for R$^5$RS Scheme.

---

```
(define reductions
  (list*
```

---

### *A.5.1 Basic syntactic forms*

These two forms, **begin** and **if**, are simple to model and unsurprising.

---

```
;; begin
((begin v e_1 e_2 ...) . /-> . (term (begin e_1 e_2 ...)))
((begin e_1) . /-> . (term e_1))

;; if
((side-condition
  (if v_1 e_1 e_2)
  (not (eq? (term v_1) (term false))))
 . /-> .
 (term e_1))
((if false e_1 e_2) . /-> . (term e_2))
```

---

### *A.5.2 Variables, binding mutation, and the store*

These rules govern the introduction of closure values into the store, binding lookup, and binding mutation. (Binding introduction is governed by the $\beta$-reduction rules introduced in subsection A.5.4.)

---

```
;; variable lookup
((letrec ((x_1 sv_1) ...
          (bp_i sv_i)
          (x_i+1 sv_i+1) ...)
   (in-hole ec_1 bp_i))
 . --> .
 (term
  (letrec ((x_1 sv_1) ...
           (bp_i sv_i)
           (x_i+1 sv_i+1) ...)
    ,(replace (term ec_1) (term hole) (term sv_i)))))

;; closure introduction
((letrec ((x_1 sv_1) ...)
   (in-hole ec_1 lam_i))
 . --> .
 (term-let ((cp_i (variable-not-in (term (x_1 ...)) 'c)))
   (term (letrec ((x_1 sv_1) ... (cp_i lam_i))
           ,(replace (term ec_1) (term hole) (term cp_i))))))

;; mu-closure introduction
((letrec ((x_1 sv_1) ...)
```

```
  (in-hole ec_1 (lambda (x_arg1 ... dot x_rest) e_1 e_2 ...)))
 . --> .
 (term-let ((mp_i (variable-not-in (term (x_1 ...)) 'm))
            (cp_i (variable-not-in (term (x_1 ...)) 'c)))
   (term
    (letrec ((x_1 sv_1) ...
             (mp_i (lambda (x_arg1 ... dot x_rest)
                     (cp_i x_arg1 ... x_rest)))
             (cp_i (lambda (x_arg1 ... x_rest) e_1 e_2 ...)))
      ,(replace (term ec_1) (term hole) (term mp_i))))))

;; set!
((letrec ((x_1 sv_1) ... (bp_i sv_i) (x_i+1 sv_i+1) ...)
   (in-hole ec_1 (set! bp_i v_new)))
 . --> .
 (term
  (letrec ((x_1 sv_1) ... (bp_i v_new) (x_i+1 sv_i+1) ...)
    ,(replace (term ec_1) (term hole) (term unspecified)))))
```

---

### *A.5.3 Cons and cons-cell mutation*

These rules govern the introduction and manipulation of cons cells. Most of them follow a simple format: locate the pair specified by a particular operation and perform the operation requested on it. The *car*, *cdr*, *set-car!*, and *set-cdr!* rules all follow this pattern. Of the other rules, *cons* introduces a new pair to the store with an appropriate store name, and *cons?* and *null?* perform the appropriate tests.

We represent pairs by moving them into the store as explained in section 2.3. One problem with that model is that the store is represented syntactically as a **letrec** term, but this representation neglects a restriction placed on **letrec** in the $R^5RS$ Scheme specification:

> *One restriction on letrec is very important: it must be possible to evaluate each [right-hand-side expression] without assigning or referring to the value of any [left-hand-side variable]. If this restriction is violated, then it is an error.*

We could fix this using a lazy-evaluation scheme, but we feel that would complicate the semantics with little benefit.

---

```
;; cons
```

```
((letrec ((x_1 sv_1) ...)
   (in-hole ec_1 ((mark cons) (mark v_car) (mark v_cdr))))
 . --> .
 (let ((p_i (variable-not-in (term (x_1 ...)) 'p)))
   (term
    (letrec ((x_1 sv_1) ... (,p_i (cons v_car v_cdr)))
      ,(replace (term ec_1) (term hole) (term ,p_i))))))

;; car
((letrec ((x_1 sv_1) ...
          (pp_i (cons v_car v_cdr))
          (x_i+1 sv_i+1) ...)
   (in-hole ec_1 ((mark car) (mark pp_i))))
 . --> .
 (term (letrec ((x_1 sv_1) ...
                (pp_i (cons v_car v_cdr))
                (x_i+1 sv_i+1) ...)
         ,(replace (term ec_1) (term hole) (term v_car)))))
((side-condition
  ((mark car) (mark v_i))
  (not (cons-v? (term v_i))))
 . e-> .
 "can't take car of non-pair")

;; cdr
((letrec ((x_1 sv_1) ...
          (pp_i (cons v_car v_cdr))
          (x_i+1 sv_i+1) ...)
   (in-hole ec_1 ((mark cdr) (mark pp_i))))
 . --> .
 (term (letrec ((x_1 sv_1) ...
                (pp_i (cons v_car v_cdr))
                (x_i+1 sv_i+1) ...)
         ,(replace (term ec_1) (term hole) (term v_cdr)))))
((side-condition
  ((mark cdr) (mark v_i))
  (not (cons-v? (term v_i))))
 . e-> .
 "can't take cdr of non-pair")

;; null?
((null? null) . *-> . 'true)
((side-condition
  ((mark null?) (mark v_i))
  (not (null-v? (term v_i))))
 . /-> .
 (term false))

;; cons?
((cons? pp)  . *-> . 'true)
((side-condition
  ((mark cons?) (mark v_i))
  (not (prefixed-by? (term v_i) 'p)))
 . /-> . (term false))

;; set-car!
((letrec ((x_1 sv_1) ...
          (pp_i (cons v_car v_cdr))
          (x_i+1 sv_i+1) ...)
   (in-hole ec_1 ((mark set-car!) (mark pp_i) (mark v_new))))
 . --> .
 (term (letrec ((x_1 sv_1) ...
                (pp_i (cons v_new v_cdr))
                (x_i+1 sv_i+1) ...)
```

```
         ,(replace (term ec_1) (term hole) (term unspecified)))))

;; set-car! error
(((mark set-car!)
  (mark (side-condition v_1 (not (cons-v? (term v_1)))))
  (mark v))
 . e-> .
 "can't set-car! on a non-pair")

;; set-cdr!
((letrec ((x_1 sv_1) ...
                     (pp_i (cons v_car v_cdr))
                     (x_i+1 sv_i+1) ...)
   (in-hole ec_1 ((mark set-cdr!) (mark pp_i) (mark v_new))))
 . --> .
 (term (letrec ((x_1 sv_1) ...
                (pp_i (cons v_car v_new))
                (x_i+1 sv_i+1) ...)
         ,(replace (term ec_1) (term hole) (term unspecified)))))

;; set-cdr! error
(((mark set-cdr!)
  (mark (side-condition v_1 (not (cons-v? (term v_1)))))
  (mark v))
 . e-> .
 "can't set-car! on a non-pair")
```

---

### *A.5.4 Functions and function application*

Function application proceeds as described in section 3.1, with marks introduced nondeterministically and reductions happening only within marks. These applications also introduce bindings into the store.

One extra complication is that R$^5$RS Scheme has a form of function not mentioned so far called a $\mu$-$\lambda$ function that in addition to a sequence of required named arguments can also accept any number of extra arguments, bundled up as a list and bound to a given name in the body of the function. We model these functions by introducing to the store two new pointers: one a normal code pointer and one a special $\mu$-$\lambda$ pointer. The $\mu$-$\lambda$ pointer is associated with the term ($\lambda$ ($x_1 \cdots x_n$ . $x_r$) ($cp_i$ $x_1 \cdots x_n$ $x_r$)), a variadic function that just passes its arguments along to the $cp_i$ function, implicitly converting unnamed arguments to a list. That function is a normal fixed-arity function whose body is the original $\mu$-$\lambda$ function's body.

---

```
;; mark introduction
((inert_1 ... e_i inert_i+1 ...)
 . /-> .
 (term (inert_1 ... (mark e_i) inert_i+1 ...)))

;; lambda application
((side-condition
  (letrec ((x_1 sv_1) ...
           (cp_i (lambda (x_arg1 ...) e_body1 e_body2 ...))
           (x_i+1 sv_i+1)  ...)
    (in-hole ec_1 ((mark cp_i) (mark v_arg1) ...)))
  (= (length (term (x_arg1 ...))) (length (term (v_arg1 ...)))))
 . --> .
 (term-let ([(bp_arg1 ...)
             (variables-not-in
              (term (v_arg1 ...))
              (term (x_1 ... cp_i x_i+1 ...))
              'b)])
           (term
            (letrec ((x_1 sv_1) ...
                     (cp_i (lambda (x_arg1 ...)
                             e_body1 e_body2 ...))
                     (x_i+1 sv_i+1) ...
                     (bp_arg1 v_arg1) ...)
              ,(replace (term ec_1)
                        (term hole)
                        (term
                         (begin
                           ,@(r5rs-subst-all
                              (term (x_arg1 ...))
                              (term (bp_arg1 ...))
                              (term (e_body1 e_body2 ...)))))))))

((side-condition
  (letrec ((x sv) ...
           (cp_i (lambda (x_arg1 ...) e e ...))
           (x sv) ...)
    (in-hole ec ((mark cp_i) (mark v_arg1) ...)))
  (not (= (length (term (x_arg1 ...)))
          (length (term (v_arg1 ...))))))
 . --> .
 (term (error "arity mismatch")))

;; mu-lambda application
((side-condition
  (letrec ((x_1 sv_1) ...
           (mp_i (lambda (x_arg1 ... dot x_argrest) (cp_target x_arg1 ... x_argrest)))
           (x_i+1 sv_i+1) ...)
    (in-hole ec_1 ((mark mp_i) (mark v_arg1) ...)))
  (>= (length (term (v_arg1 ...))) (length (term (x_arg1 ...)))))
  . --> .
  (let-values ([(named rest)
                (divide (term (v_arg1 ...)) (length (term (x_arg1 ...))))])
    (term-let (((v_named1 ...) named)
               (v_rest (mkmarkedlist rest)))
      (term
       (letrec ((x_1 sv_1) ...
                (mp_i (lambda (x_arg1 ... dot x_argrest) (cp_target x_arg1 ... x_argrest)))
                (x_i+1 sv_i+1) ...)
         ,(replace (term ec_1)
                   (term hole)
                   (term ((mark cp_target)
                          (mark v_named1) ...
                          (mark v_rest)))))))))
```

```
;; mu-lambda too few arguments case
((side-condition
  (letrec ((x sv) ...
           (mp_i (lambda (x_arg1 ... dot x) (cp x ...)))
           (x sv) ...)
    (in-hole ec ((mark mp_i) (mark v_arg1) ...)))
  (< (length (term (v_arg1 ...)))
     (length (term (x_arg1 ...)))))
 . --> .
 (term (error "too few arguments")))

;; application of non-function
(((mark nonfun) (mark v) ...)
 . e-> .
 "can't apply non-function")

;; apply built-in function
((letrec ((x_1 sv_1) ...
          (pp_i (cons v_car v_cdr))
          (x_i+1 sv_i+1) ...)
   (in-hole ec_1 ((mark apply) (mark v_f) (mark v_arg1) ... (mark pp_i))))
 . --> .
 (term
  (letrec ((x_1 sv_1) ...
           (pp_i (cons v_car v_cdr))
           (x_i+1 sv_i+1) ...)
    ,(replace (term ec_1)
              (term hole)
              (term ((mark apply)
                     (mark v_f)
                     (mark v_arg1) ...
                     (mark v_car)
                     (mark v_cdr)))))))

((apply v_f v_arg1 ... null)
 . *-> .
 (term (v_f v_arg1 ...)))

((side-condition
  ((mark apply) (mark v_f) (mark v_arg1) ... (mark v_last))
  (not (list-v? (term v_last))))
 . e-> .
 "apply must take a list as its last argument")
```

---

### *A.5.5 Call/cc*

R$^5$RS Scheme's *call/cc* feature is modeled by capturing the context surrounding its application and building a function that accepts an argument, substitutes the hole in that context with the argument when applied, and aborts after evaluating the entire context.

We take this modeling technique directly from a paper by Felleisen [8], extending it only to support multiple values. Our extension works by changing the function used as a

continuation so that rather than being a $\lambda$ function that accepts exactly one argument and returning that argument to *call/cc*'s calling context, it becomes a $\mu - \lambda$ function that accepts zero or more arguments and returns all its arguments as values to *call/cc*'s calling context.

The **abort** primitive violates our guideline that we should not introduce extra scaffolding, but in a relatively benign way: if we modified terms such that program contexts were

(**letrec** (($x$ $sv$) $\cdots$)
(*let/cc* **abort** *EC*))

the abort construction would be legal. We chose to omit that extra layer because it is a potentially confusing constant addition to every term that has no particular benefit other than making terms slightly more aesthetic.

---

```
;; call/cc
((letrec ((x_1 sv_1) ...)
   (in-hole ec_1 ((mark call/cc) (mark v_arg))))
 . --> .
 (let ((k (variable-not-in (term (ec_1 x_1 ...)) 'k)))
   (term
    (letrec ((x_1 sv_1) ...)
      ,(replace
        (term ec_1)
        (term hole)
        (term ((mark v_arg)
               (mark
                (lambda (dot ,k)
                  (abort
                   ,(replace
                     (term ec_1)
                     (term hole)
                     (term (apply values ,k))))))))))))

;; abort (introduced by call/cc)
((letrec ((x_1 sv_1) ...)
   (in-hole ec_1 (abort e_1)))
 . --> .
 (term (letrec ((x_1 sv_1) ...) e_1)))
```

---

### *A.5.6 Multiple values and call-with-values*

Our multiple values model proceeds exactly as described in sections 2.4 and 2.6. The mark we place around *call-with-values* violates the aesthetic principle that intermediate terms

should be legal Scheme programs in their own right, and we know of no way to eliminate that mark without resorting to **apply-values** or another non-Scheme construct. This is the most significant breach of our aesthetic principle in our model.

---

```
;; values promotion
((in-named-hole multi
                hole
                pc_1
                v_1)
 . --> .
 (replace (term pc_1)
          (term hole)
          (term ((mark values) (mark v_1)))))

;; values demotion
((in-named-hole single
                hole
                pc_1
                ((mark values) (mark v_1)))
 . --> .
 (replace (term pc_1) (term hole) (term v_1)))

((in-named-hole single
                hole
                pc_1
                (side-condition
                 ((mark values) (mark v_1) ...)
                 (not (= (length (term (v_1 ...))) 1))))
 . --> .
 (term (error "context received the wrong number of values")))

; resolving call-with-values statements
((call-with-values v_vals v_fun)
 . *-> .
 (term ((mark call-with-values)
        (cwv-mark (v_vals))
        (mark v_fun))))

((in-hole
  pc_1
  ((mark call-with-values)
   (cwv-mark ((mark values) (mark v_arg) ...))
   (mark v_fun)))
 . --> .
 (replace (term pc_1)
          (term hole)
          (term ((mark v_fun) (mark v_arg) ...))))

((side-condition
  ((mark call-with-values) (mark v_i) ...)
  (not (= (length (term (v_i ...))) 2)))
 . e-> .
 "arity mismatch")
```

---

### *A.5.7 Eq? and equivalence*

Other reductions in the semantics ensure that compound values are placed in the store as described in section 2.3. For that reason, equality testing is a symbol equality test in all cases.

---

```
((eq? pp_i pp_i) . *-> . 'true)
((eq? cp_i cp_i) . *-> . 'true)
((eq? number_1 number_1) . *-> . 'true)
((eq? v_1 v_1) . *-> . 'true)
((side-condition
  ((mark eq?) (mark v_1) (mark v_2))
  (not (eq? (term v_1) (term v_2))))
 . /-> .
 (term false))

((side-condition
  ((mark eq?) (mark v_1) ...)
  (not (= (length (term (v_1 ...))) 2)))
 . e-> .
 "arity mismatch")

; primitives
(prims + - * /)))
```

---

## A.6 Odds and ends

The remainder of the code implements various helpers and convenience features that generally are not spelled out in a formal semantics document but that must be specified to produce an executable specification. The most notable function here is *r5rs-subst-one*, the capture-avoiding substitution function. It is defined using a domain-specific language for building capture-avoiding substitution functions that is included in PLT Redex; for details see PLT Redex documentation within DrScheme.

---

```
(define r5rs-subst-one
  (subst
   ['cons (constant)]
   ['null (constant)]
   ['abort (constant)]
```

```
    [‘call/cc (constant)]
    [‘mark (constant)]
    [‘cwv-mark (constant)]
    [(? symbol?) (variable)]
    [(? number?) (constant)]
    [‘(lambda ,(xs ... ’dot last) ,b)
      (all-vars (cons last xs))
      (build (lambda (vars body) ‘(lambda ,(xs ’dot last) ,body)))
      (subterm xs b)]
    [‘(lambda ,(xs ...) ,b)
      (all-vars xs)
      (build (lambda (vars body) ‘(lambda ,xs ,body)))
      (subterm xs b)]
    [‘(lambda ,x ,b)
      (all-vars (list x))
      (build (lambda (vars body) ‘(lambda ,x ,body)))
      (subterm x b)]
    [‘(set! ,x ,e)
      (all-vars ’())
      (build (lambda (vars x exp) ‘(set! ,x ,exp)))
      (subterm ’() x)
      (subterm ’() e)]
    [(f args ...)
     (all-vars ’())
     (build (lambda (vars f . args) ‘(,f ,@args)))
     (subterm ’() f)
     (subterms ’() args)]))

(define (r5rs-subst-all params args bodies)
  (map
   (lambda (body) (srfi1:fold r5rs-subst-one body params args))
   bodies))

(define (variables-not-in items exp prefix)
  (cond
    [(null? items) null]
    [else
     (let ((this (variable-not-in exp prefix)))
       (cons
        this
        (variables-not-in
         (cdr items)
         (cons this exp)
         prefix)))]))

; mklist : listof term -> term
; makes a term representing a cons-list of the given list
(define (mklist args)
  (srfi1:fold-right
   (lambda (this rest) (term (cons ,this ,rest))
   (term null)
   args)))

; mkmarkedlist : listof term -> term
; like mklist, but pre-marks all values in all applications
(define (mkmarkedlist vals)
  (srfi1:fold-right
   (lambda (this rest)
     (term ((mark cons) (mark ,this) (mark ,rest))))
   (term null)
   vals))

(define cons-v? (language->predicate lang ’pp))
(define (null-v? v) (eq? v ’null))
```

```
(define (list-v? v) (or (cons-v? v) (null-v? v)))

(define (prefixed-by? s prefix)
  (let* ([sym (symbol->string s)]
         [pre (symbol->string prefix)]
         [len (string-length pre)])
    (string=? (substring sym 0 len) pre)))

(define (divide li n)
  (let loop ((n n)
             (li li)
             (acc '()))
    (cond
      [(zero? n) (values (reverse acc) li)]
      [else (loop (sub1 n) (cdr li) (cons (car li) acc))])))));
```

# REFERENCES


[1] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*, volume 103 of *Studies in Logics and the Foundations of Mathematics*. North Holland, Amsterdam, The Netherlands, 1981.

[2] P. Borovansky, C. Kirchner, H. Kirchner, P. E. Moreau, and M. Vittek. ELAN: A logical framework based on computational systems. In *Proc. of the First Int. Workshop on Rewriting Logic*, volume 4. Elsevier, 1996.

[3] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and José F. Quesada. Maude: Specification and programming in rewriting logic. *Theoretical Computer Science*, 2001.

[4] Dominique Clément, Janet Incerpi, and Gilles Kahn. Centaur: Towards a ”software tool box” for programming environments. In *International Workshop on Software Engineering Environments*, pages 287–304, 1989.

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

[6] Marc Feeley. The R6RS status report. In *Proceedings of the Fifth Workshop on Scheme and Functional Programming*, 2004. Indiana University Computer Science Department TR 600.

[7] Matthias Felleisen. *The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State In Imperative Higher-Order Programming Languages*. PhD thesis, Indiana University, 1987.

[8] Matthias Felleisen. Lambda-v-CS: and extended lambda-calculus for Scheme. In *Proceedings of the Conference on LISP and Functional Programming*, 1988.

[9] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical Computer Science*, pages 235–271, 1992.

[10] Cormac Flanagan and Matthias Felleisen. The semantics of future. Technical Report TR 94-238, Rice University, 1994.

[11] Matthew Flatt. PLT MzScheme: Language manual. Technical Report TR97-280, Rice University, 1997. `http://www.plt-scheme.org/software/mzscheme/`.

[12] Matthew Flatt, Shriram Krishnamurthi, and Matthias Felleisen. A programmer's reduction semantics for classes and mixins. *Formal Syntax and Semantics of Java*, 1523:241–269, 1999. Preliminary version appeared in proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University technical report TR 97-293, June 1999.

[13] Martin Gasbichler, Eric Knauel, Michael Sperber, and Richard A. Kelsey. How to add threads to a sequential language without getting tangled up. In *Proceedings of the 2003 Scheme Workshop*, 2003.

[14] Robert Harper and Mark Lillibridge. Explicit polymorphism and CPS conversion. In *Proceedings of the ACM Conference Principles of Programming Languages*, 1993.

[15] Rickard Kelsey, William Clinger, and Jonathan Rees (Editors). Revised$^5$ report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, 33(9):26–76, 1998.

[16] S. Lippi. in2: A Graphical Interpreter for Interaction Nets (system description). In S. Tison, editor, *Rewriting Techniques and Applications, 13th International Conference, RTA-02*, LNCS 2378, pages 380–384, Copenhagen, Denmark, July 22–24, 2002. Springer.

[17] Jacob Matthews, Robert Bruce Finder, Matthew Flatt, and Matthias Felleisen. A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA)*, 2004.

[18] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML*. The MIT Press, revised edition, 1997.

[19] Matthias Neubauer and Michael Sperber. Down with Emacs Lisp: Dynamic scope analysis. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, 2001.

[20] G. D. Plotkin. Call-by-name, call-by-value and the $\lambda$-calculus. *Theoretical Computer Science*, 1:125–159, 1975.

[21] John D. Ramsdell. An operational semantics for Scheme. *Lisp Pointers*, volume 2, April–June 1992.

[22] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[23] M. G. J. van den Brand, J. Heering, P. Klint, and P. A. Oliver. Compiling language definitions: The ASF+SDF compiler. *ACM Transactions on Programming Languages and Systems*, 24(4):334–368, 2002.

[24] Eelco Visser. Stratego: A language for program transformation based on rewriting strategies. System description of Stratego 0.5. In A. Middeldorp, editor, *Rewriting Techniques and Applications (RTA'01)*, volume 2051 of *Lecture Notes in Computer Science*, pages 357–361. Springer-Verlag, May 2001.

[25] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, pages 38–94, 1994. First appeared as Technical Report TR160, Rice University, 1991.