An operational semantics for R^{5}RS Scheme

Jacob Matthews
University of Chicago
jacobm@cs.uchicago.edu

Robert Bruce Findler
University of Chicago
robby@cs.uchicago.edu

Abstract

This paper presents an operational semantics for the core of Scheme. Our specification improves over the existing R^{5}RS denotational specification in four ways. First, it is more complete, since it contains *eval*, **quote**, and *dynamic-wind*. Second, it models multiple values in a way that does not require changes to unrelated parts of the language. Third, it provides a more faithful model of Scheme's undefined order of evaluation. Finally, it is executable, because it is encoded as a program in PLT Redex, a domain-specific language for writing operational semantics. The executable specification allows others to experiment with our specification and allows us to build a specification test suite, which improves our confidence that our system is a faithful model of Scheme.

In addition to contributing a specification of Scheme, this paper presents several novel modeling techniques for Felleisen Hieb-style rewriting semantics that we discovered while developing our R^{5}RS Scheme semantics. All are applicable to a wider range of problems than the specific uses we have for them, and the fact that they combine seamlessly in our full R^{5}RS model shows that they scale to real languages.

1. Introduction

The Revised^5 Report on the Algorithmic Language Scheme [15], R^{5}RS, provides an informal, English specification of Scheme and a denotational model of a core Scheme language. The denotational specification is more precise than the informal specification, but is also incomplete with respect to it. For instance, the formal specification does not present the top-level mentioned throughout the informal specification, and is missing key procedures such as *dynamic-wind* and *eval* whose inclusion could have a significant impact on the formalism. While that is not necessarily a problem — the measure of a model is not its completeness but its ability to clearly and accurately explain its subject — Gasbichler et al's recent explanation of the difficulties involving dynamic contexts and threads [12], for instance, demonstrate that the formal model is insufficient for some important questions.

In this paper we give a new treatment of the R^{5}RS formal semantics that models more of the language described in the informal semantics section than the formal semantics section in the R^{5}RS Scheme document does. Rather than extending the denotational semantics with extra constructs, we present an alternate specification as a small-step operational semantics. We do this for two major reasons. First, to make the semantics natively executable: operational semantics are much more amenable to direct execution than denotational semantics. Second, to allow for nondeterminism and nonconfluence: small-step operational semantics are particularly well-suited for modeling programming languages with nondeterministic and nonconfluent behavior. We make important use of nondeterminism in our model, as we will explain in section 2.

As a side benefit of using a small-step operational encoding, we can use PLT Redex [17], a domain-specific language for context-sensitive term-rewriting systems, to give a directly executable operational encoding for our model. PLT Redex provides a graphical browser for exploring reduction graphs and allows us to maintain a large test suite of terms and their expected normal forms that we can run whenever we change any reduction rules. This test suite increases our confidence that our model is a faithful representation of Scheme.

While writing our model, we developed new techniques for modeling some of Scheme's features. In the rest of our paper we first introduce those techniques in isolation to explain our models for particular Scheme features, and then combine them into a single unified model. In section 2 we show how to use nondeterminism to model Scheme's unspecified application order; in section 3 we show a novel technique for modeling multiple return values; in section 4 we give a model for **quote** and *eval*; and in section 5 we give a model for *call/cc* in the presence of *dynamic-wind*. Finally in section 6 we combine all those models along with several other more straightforward features: **if**, *cons* and cons-cell mutation, variable-arity procedures, *apply*, and an object-identity-sensitive notion of *eqv?* equality.

We will assume the reader has a basic familiarity with context-sensitive reduction semantics. Readers unfamiliar with this system may wish to consult Felleisen and Flatt's monograph [5] or Wright and Felleisen [24] for a thorough introduction or our previous work with Flatt and Felleisen [17] for a somewhat lighter one. We should also emphasize before we proceed that this semantics still leaves out many important Scheme features — among them the numeric tower, the top-level environment, and macros — but that it models more features than the Report's formal semantics does and is more suitable for extension.

2. Unspecified application order

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

> *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*

Sixth Workshop on Scheme and Functional Programming. September 25, 2005, Tallinn, Estonia.

$$
\begin{array}{lcl}
p & ::= & (\textbf{store}\ ((x\ v)\cdots)\ e) \\
e & ::= & (e\ e\cdots) \mid (\textbf{set!}\ x\ e) \mid (\textbf{begin}\ e\ e\cdots) \mid v \\
v & ::= & (\textbf{lambda}\ (x\cdots)\ e) \mid n
\end{array}
\qquad
\begin{array}{lcl}
C & ::= & (v\cdots C\ e\cdots) \mid (\textbf{set!}\ x\ C) \mid (\textbf{begin}\ C\ e\ e\cdots) \mid [] \\
x & ::= & \text{identifiers, store locations for mutable bindings} \\
n & ::= & \text{numbers}
\end{array}
$$

$$
\begin{array}{lcll}
(\textbf{store}\ ((x_1\ v_1)\cdots)\ C[((\textbf{lambda}\ (x_2\cdots)\ e)\ v_2\cdots)]) & \rightarrow & (\textbf{store}\ ((x_1\ v_1)\cdots(x_2'\ v_2)\cdots)\ C[e[x_2'\cdots/\ x_2\cdots]]) & (\textsc{MApp}) \\
 & & (\#x_2 = \#v_2, \text{ each } x_2' \text{ fresh}) & \\
(\textbf{store}\ ((x_1\ v_1)\cdots)\ C[((\textbf{lambda}\ (x_2\cdots)\ e)\ v_2\cdots)]) & \rightarrow & \textbf{error:}\ \text{wrong number of arguments} & (\textsc{MAppErr}) \\
 & & (\#x_2 \neq \#v_2) & \\
(\textbf{store}\ ((x_1\ v_1)\cdots(x\ v)\ (x_2\ v_2)\cdots)\ C[(\textbf{set!}\ x\ v')]) & \rightarrow & (\textbf{store}\ ((x_1\ v_1)\cdots(x\ v')\ (x_2\ v_2)\cdots)\ C[0]) & (\textsc{MSet}) \\
(\textbf{store}\ ((x_1\ v_1)\cdots(x\ v)\ (x_2\ v_2)\cdots)\ C[x]) & \rightarrow & (\textbf{store}\ ((x_1\ v_1)\cdots(x\ v)\ (x_2\ v_2)\cdots)\ C[v]) & (\textsc{MLookup}) \\
(\textbf{store}\ ((x\ v)\cdots)\ C[(\textbf{begin}\ v\ e_1\ e_2\cdots)]) & \rightarrow & (\textbf{store}\ ((x\ v)\cdots)\ C[(\textbf{begin}\ e_1\ e_2\cdots)]) & (\textsc{MSeq}) \\
(\textbf{store}\ ((x\ v)\cdots)\ C[(\textbf{begin}\ e)]) & \rightarrow & (\textbf{store}\ ((x\ v)\cdots)\ C[e]) & (\textsc{MTrivSeq}) \\
(\textbf{store}\ ((x\ v)\cdots)\ C[(-\ \ulcorner n\urcorner)]) & \rightarrow & (\textbf{store}\ ((x\ v)\cdots)\ C[\ulcorner -n\urcorner]) & (\textsc{MNeg})
\end{array}
$$

Figure 1. Core Scheme with mutation

right since it suggests, incorrectly, that the order of evaluation is constant throughout a program . . .. [section 7.2]

In this section we present an operational technique that captures the intended semantics more faithfully. We begin by considering a core Scheme with arbitrary arity procedures, **set!**, numbers, and negation, but with a fixed left-to-right order of evaluation for applications, as shown in figure 1. It is a minor variation of Felleisen and Hieb's Λ_S [6]. A program consists of a store that associates variable names to values and an expression, where expressions are built up of numbers, arbitrary-arity lambda terms and applications, **set!**, and **begin** expressions, and a built-in negation operator. MApp gives the rule for application of a procedure to fully-evaluated arguments: make one fresh identifier x_i' for each formal parameter x_i, introduce a new binding in the store for each x_i' associating it with the corresponding argument v_i in the application, and then rewrite the application as the procedure's body with each occurrence of an x_i rewritten into the corresponding x_i' (in this figure as in all figures in this paper, we will use vertically-centered ellipses $\cdots$ to indicate any number of occurences, including zero, of the preceding element). MAppErr gives the rule for procedures applied to the wrong number of arguments: rewrite the term in its entirety to an error message, which halts the program immediately because it abandons the application's original context. MSet rewrites to the constant 0 but also replaces the value associated with the given identifier in the store with the given replacement. (We choose to have **set!** return the constant 0 in this semantics as a "quick and dirty" unique value; in the examples that follow 0 never appears in any program term except as the result of assignment.) MLookup replaces an identifier with its associated value in the store when that value becomes necessary (*i.e.*, when it appears as a redex in an evaluation context). MSeq drops the first subexpression in a **begin** when there are more expressions to evaluate, and MTrivSeq drops the **begin** when there is only one expression to evaluate. The last rule, MNeg, simply negates its argument (the notation $\ulcorner n\urcorner$ indicates the syntactic representation corresponding to the mathematical number n).

The order of evaluation is determined by the grammar for evaluation contexts (C). The first production of the grammar specifies that evaluation of a sub-expression of an application only takes place when all of the sub-expressions to its left are values (or have been reduced to values). If we replace that first production with this one:

$$C ::= (e\cdots C\ v\cdots) \mid \ldots$$

the semantics would specify a right-to-left order instead.

Either of these choices results in a system with unique decomposition. That is, each term can only be split into an evaluation context and a reducible sub-expression in one way (unless it is stuck or an answer). Accordingly, there is at most one way to reduce any expression.

To model a language with unspecified order of operations instead, we can use a reduction system with non-unique decomposition to model the choice. We might be tempted to use this definition of evaluation contexts:

$$C ::= (e\cdots C\ e\cdots) \mid \ldots$$

Since this definition allows the hole to appear in any subexpression of an application, this simple program that negates 1, negates 2, and then applies a trivial procedure to the results

$$((\textbf{lambda}\ (x\ y)\ y)\ (-\ 1)\ (-\ 2))$$

can be split into an evaluation context with either $(-\ 1)$ or $(-\ 2)$ as the reducible expression.

At first glance, this appears to be a faithful model of R^{5}RS Scheme. It is not. Consider this application of two **set!** expressions in a store binding x to 1.

$$
\begin{array}{l}
(\textbf{store}\ ((x\ 1)) \\
\quad((\textbf{set!}\ x\ (-\ x)) \\
\quad\ (\textbf{set!}\ x\ (-\ x))))
\end{array}
$$

In Scheme, this program should always reduce to the application of zero to zero with x set to 1 in the store (and then get stuck). According to R^{5}RS, no matter which of the application's subterms is reduced first, the result should be that x is negated twice. If we just modify evaluation contexts as above, however, we allow other interleavings. The problem is that that definition of evaluation contexts would allow a different argument of the same application to take one step of computation every step of the way, which may produce an outcome that could not be reached by any sequential ordering.

We discovered this problem while experimenting with that reduction system in PLT Redex. We encoded the erroneous reduction system in PLT Redex and automatically generated the reduction sequence for the above term, shown in figure 2. The first term is shown on the left. The top-most and the bottom-most paths correspond to the two sequential orderings and result in the proper store. In the middle section, the two assignments are interleaved, resulting in -1 being left in the store.

With that in mind, we can design a more sophisticated strategy that captures unspecified evaluation order but only allows sequential orderings. Figure 3 shows the necessary revisions to core Scheme to support R^{5}RS-style procedure applications (each replaces the appropriate rule from figure 1 — the other rules in that figure are unchanged). 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

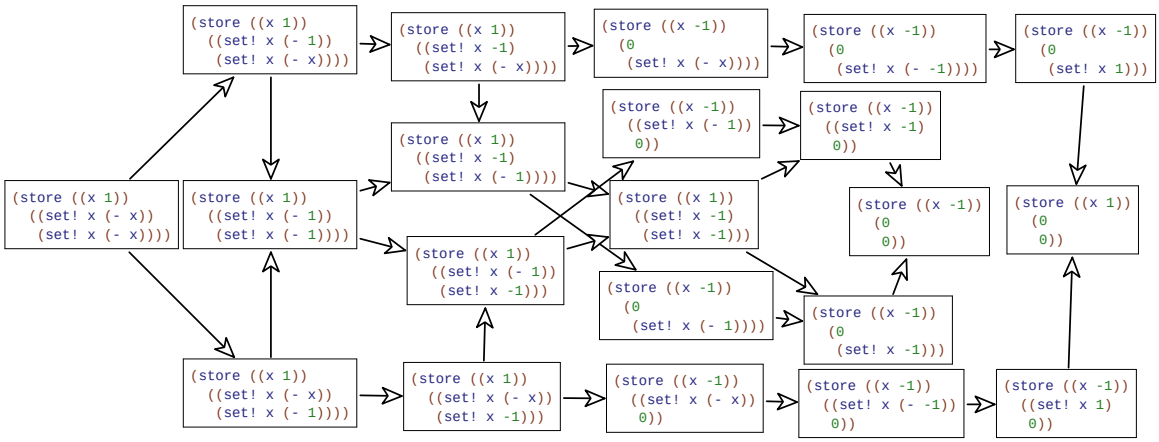

Figure 2. Interleavings possible with an erroneous unspecified-application-order model

$$\begin{array}{rcl} \mathit{inert} & ::= & v^{\circ} \mid e \\ C & ::= & (\mathit{inert} \cdots C^{\circ}\ \mathit{inert} \cdots) \mid \ldots \end{array}$$

$$\begin{array}{lcll} (\textbf{store}\ (\cdots)\ C[(\mathit{inert} \cdots e\ \mathit{inert} \cdots)]) & \rightarrow & (\textbf{store}\ (\cdots)\ C[(\mathit{inert} \cdots e^{\circ}\ \mathit{inert} \cdots)]) & (\text{UMARK}) \\ (\textbf{store}\ (\cdots)\ C[((\textbf{lambda}\ (x \cdots)\ e)^{\circ}\ v^{\circ} \cdots)]) & \rightarrow & (\textbf{store}\ (\cdots\ (x'\ v) \cdots)\ C[e[x' \cdots / x \cdots]]) & (\text{UAPP}) \\ & & (\#x = \#v, \text{ each } x' \text{ fresh}) & \\ (\textbf{store}\ (\cdots)\ C[(-^{\circ}\ \ulcorner n \urcorner^{\circ})]) & \rightarrow & (\textbf{store}\ (\cdots)\ C[\ulcorner -n \urcorner]) & (\text{UNEG}) \end{array}$$

Figure 3. Revisions to core Scheme to support unspecified application order

that effect, we introduce the non-terminal *inert* and the notion of a marked expression, denoted with the $\circ$ superscript. (These marks are not an extension to the general term-rewriting framework — e° and C° are just alternate typesettings of (**mark** *e*) and (**mark** *C*).) Marks identify chosen expressions: 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 finished reducing). We add the UMARK reduction rule that marks an arbitrary unmarked expression in an application on the condition that every other expression is inert, and we modify the MAPP and MNEG rules rules to apply only to fully-marked applications, becoming the UAPP and UNEG rules.

Figure 4 (also generated by PLT Redex) shows how our new system evaluates the term from figure 2. The initial term appears in the center on the left. That term is an application, so the first reduction either marks the first sub-expression or the second. If the first subexpression is marked, evaluation continues down to the bottom of the figure, over to the right and back up to the middle. If the second is marked, evaluation proceeds up, over, and back to the middle. In both paths there are a few other application expressions to evaluate, leading to smaller separations. Eventually, all of the terms join back together and the final result in the store is *1*, as shown in the center on the right.

One should not take that example to mean that this language has any kind of confluence property, however. Consider this program:

```
((lambda (choice)
   ((lambda (x y) choice)
    (set! choice 1)
    (set! choice 2)))
 0)
```

It will either will return either *1* or *2*, depending on the order of evaluation. This is the way we want it; the model's nonconfluence reflects the underspecification of R^{5}RS Scheme rather than a technical bug in our model. It does, however, always make progress. We formalize this with the following theorem statement:

THEOREM 2.1. *For any closed program p in the language of figure 3, either $p \rightarrow p'$, where p' is also closed, $p \rightarrow e$ where e is some error indicator, or p is of the form* (**store** *((x v) ⋯) v)*.

Proof is contained in the first author's master's thesis [16].

This technique has other uses besides giving semantics for unspecified application evaluation orders. In general, it is useful for modeling any kind of delimited nondeterminism, where evaluation may proceed arbitrarily but only at certain points in a program. This is useful for modeling unspecified behaviors and for complex nondeterministic features such as threads.

3. Multiple return values

R^{5}RS Scheme provides a facility for expressions to evaluate to multiple or no values rather than just a single value. The procedure *values* builds multiple values and *call-with-values* accepts multiple values. Unlike tuples in SML and Haskell, multiple values are not themselves values. For example, this program

```
(define (f x) (values (+ x x) (* x x)))
(define (g x y) y)
(g (f 3))
```

produces an error, since procedure application expects each of its arguments to be a single value (and the result of *f* is two values). Instead, the programmer must use *call-with-values* to catch multiple values. It expects a thunk as its first argument, applies the thunk, catches any number of values that thunk produces, and applies them

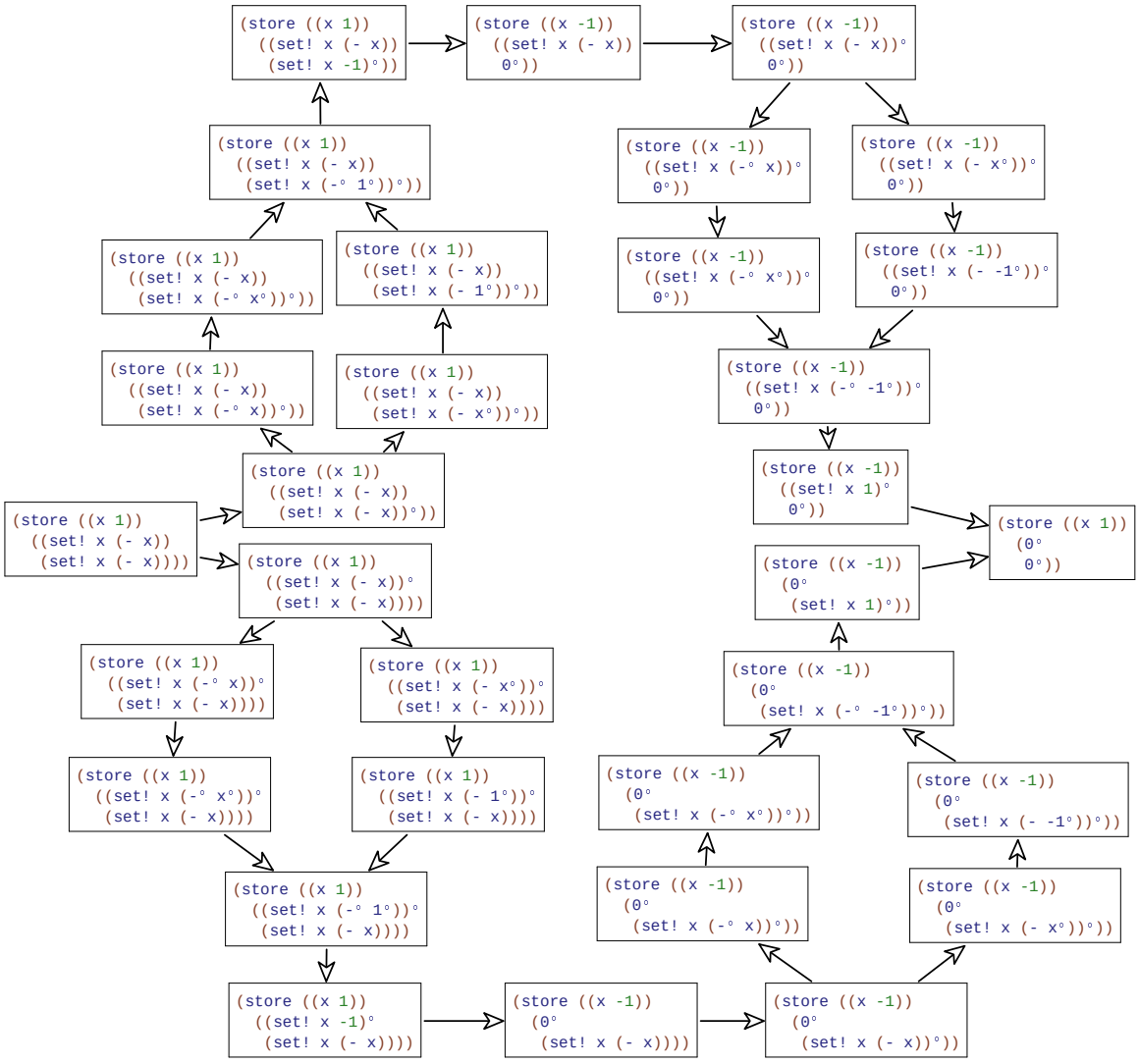

Figure 4. Evaluation in the unspecified-application-order model

to its second argument. So, a programmer could supply *f*'s results to *g* like this:

(*call-with-values* (**lambda** () (*f* 3)) *g*)

In addition, there is no difference between *values* applied to a single argument and that argument by itself, so (*g* (*values* 6) (*values* 9)) is the same as (*g* 6 9).

To model multiple values, R^{5}RS Scheme's formal semantics models continuations as functions from an arbitrary number of values to a final answer. The informal semantics says that "except for continuations created with the *call-with-values* procedure, all continuations take exactly one value" [15, section 6.4]. The formal semantics reflects this by checking the opposite property: in every context that expects a single value, it uses a helper function, *single*, to ensure that only a single value appears. This indirect checking impacts the entire semantics: it requires every continuation to accept any number of arguments initially and requires a call to *single* at every point where a continuation would be restricted.

Our semantic model captures the difference between contexts that accept multiple values and contexts that reject multiple values directly. Our strategy is distilled in figure 5. That figure contains a pure core Scheme extended with *values*, and **apply-values**, a syntactic form that has as its operands an expression that must evaluate to a procedure and another expression that may evaluate to any number of values, and calls the procedure with those values as arguments. We use **apply-values** in this section rather than *call-with-values* because the resulting model is clearer and both **apply-values** and *call-with-values* can be defined simply in terms of each other in R^{5}RS Scheme:

```
(define (call-with-values thunk f)
  (apply-values f (thunk)))

(define-syntax apply-values
  (syntax-rules ()
    [(_ f vs-expr)
     (call-with-values (lambda () vs-expr) f)]))
```

Our model uses a modest addition to the standard reduction-semantics formalism. We extend the notation so that holes have names (written as subscripts) but otherwise behave as unnamed

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

$$
\begin{array}{lcll}
C_{\circ}[((\textbf{lambda}\ (x\ \cdots)\ e)\ v\ \cdots)]_{\circledast} & \rightarrow & C_{\circ}[e[x\ \cdots / v\ \cdots]] \quad (\#v = \#x) & (\textsc{VApp}) \\
C_{\circ}[((\textbf{lambda}\ (x\ \cdots)\ e)\ v\ \cdots)]_{\circledast} & \rightarrow & \textbf{error:}\ \text{wrong number of arguments} \quad (\#v \neq \#x) & (\textsc{VAppErr}) \\
C_{\circ}[(\textbf{apply-values}\ v_1\ (\mathit{values}\ v_2\ \cdots))]_{\circledast} & \rightarrow & C_{\circ}[(v_1\ v_2\ \cdots)] & (\textsc{VAppVals}) \\
C_{\circ}[v]_{*} & \rightarrow & C_{\circ}[(\mathit{values}\ v)] & (\textsc{VPromote}) \\
C_{\circ}[(\mathit{values}\ v)]_{\circ} & \rightarrow & C_{\circ}[v] & (\textsc{VDemote}) \\
C_{\circ}[(\mathit{values}\ v\ \cdots)]_{\circ} & \rightarrow & \textbf{error:}\ \text{expected a single value} \quad (\#v \neq 1) & (\textsc{VDemoteErr})
\end{array}
$$

Figure 5. Pure core Scheme with multiple values

holes do. The context-matching syntax is now annotated with names as well, restricting legal decompositions to those where the hole has the same name.

In figure 5 we use this feature to give three distinct names to holes, indicated with subscripts. $[\,]_{\circ}$ indicates a hole in which any expression should reduce to an element of v, $[\,]_{*}$ indicates a hole in which any expression should reduce to (*values* $v\ \cdots$), and $[\,]_{\circledast}$ indicates a hole in which either result is acceptable. There are three parallel context nonterminals. The context $C_{\circ}$ produces an element of v, C_{*} produces (*values* $v\ \cdots$), and C might produce either.

Since each subexpression of an application is expected to produce a single value, the evaluation context inside an application is $C_{\circ}$. For the same reason, the evaluation context for the first subexpression of **apply-values** is $C_{\circ}$. The evaluation context for the second subexpression, however, is C_{*} because it is expected to produce multiple values.

Since procedure applications (defined by the VApp and VAppErr reductions) and **apply-values** uses (defined by the VAppVals reduction) may produce a single value or (*values* $v\ \cdots$), they take place in $[\,]_{\circledast}$ holes. VPromote, promotes a single value v to (*values* v). Because of the subscript * on the hole, it applies only when multiple values are expected. VDemote demotes a single value inside *values* to just the value, and VDemoteErr signals an error if *values* does not return exactly one value. These two rules apply only when a *values* expression appears where a single value is expected. All reductions take place in $C_{\circ}$ to ensure that the final result of any program is a single value. If we wanted to allow any number of values as the final result of a program we could replace $C_{\circ}$ with C_{*} in all rules.

To get a sense of how evaluation proceeds, consider this reduction sequence:

```
    ((lambda (y) y)
     (apply-values (lambda (x) (values x)) 1))
→   ((lambda (y) y)
     (apply-values (lambda (x) (values x))
                   (values 1)))                    (VPromote)
→   ((lambda (y) y)
     ((lambda (x) (values x)) (values 1)))         (VAppVals)
→   ((lambda (y) y)
     ((lambda (x) (values x)) 1))                  (VDemote)
→   ((lambda (y) y) (values 1))                   (VApp)
→   ((lambda (y) y) 1)                            (VDemote)
→   1                                             (VApp)
```

First, the VPromote applies and promotes 1 into (*values 1*) because it appears as the second argument of an **apply-values** expression. Then VAppVals applies, followed by VApp. Then the term (*values 1*) is used as an argument to a procedure, so VDemote applies and converts it to the single value *1*. Finally VApp applies and the result is *1*.

The erroneous expression from the beginning of this section signals an error due to the VDemoteErr rule.

```
    (g (f 3))
→ ···
→ (g (values 3 9))
→ error: expected a single value
```

The evaluation contexts and the three promotion and demotion rules are all that we need to add multiple values to the language. Furthermore, the extension of adding names to holes does not significantly complicate proof of progress for the system, and so we can prove the following theorem reasonably straightforwardly [16]:

Theorem 3.1. *For any closed program p in the language of figure 5, either $p \rightarrow p'$, where p' is also closed, $p \rightarrow e$ where e is an error indicator, or p is of the form (***store** *((x v) ···) v).*

Proof is contained in the first author's master's thesis [16].

The strategy described in this section can be used whenever the notion of a fully-evaluated subterm is different in different parts of a program. For instance, it can be used to model embedded sublanguages such as regular-expressions, format strings, and SQL commands, which could help develop theoretical underpinnings for work like Herman and Meunier's static analysis of embedded languages [14].

4. Quote and Eval

Scheme inherits the meta-programming facilities *eval* and **quote** from Lisp [22]. The **quote** operator turns a program into data and the *eval* procedure turns that data back into a program. When quoted, a program is represented as a list of lists and symbols, where lists represent parenthesized sequences and symbols represent identifiers. For example, (**quote** (*lambda* (*x*) *x*)) is a three el-

$$
\begin{array}{llll}
e & ::= & (e\ e\ \cdots) \mid v \mid x & \\
E & ::= & [\,] \mid (v \cdots E\ e \cdots) & \\
v & ::= & (\textbf{lambda}\ (x \cdots)\ e) \mid (\textbf{quote}\ sy) & \\
 & \mid & p \mid null \mid n \mid prim \mid \#t \mid \#f & \\
prim & ::= & eval \mid cons \mid car \mid cdr \mid eqv? & \\
p & ::= & \text{pointers} & \\
x & ::= & \text{program variables} & \\
 & & \text{(members of } sy \text{ except } \textbf{lambda}, \textbf{quote}, \textbf{ccons}) &
\end{array}
\qquad
\begin{array}{llll}
s & ::= & (s \cdots) \mid n \mid sy & \\
 & \mid & (s \cdots \textbf{dot}\ sy) \mid (s \cdots \textbf{dot}\ n) & \\
S & ::= & [\,] \mid (e \cdots S\ s \cdots) & \\
 & \mid & (\textbf{lambda}\ (x \cdots)\ S) & \\
 & \mid & (\textbf{ccons}\ v\ S) \mid (\textbf{ccons}\ S\ s) & \\
n & ::= & \text{numbers} & \\
sf & ::= & (p\ (cons\ v\ v)) & \\
sy & ::= & \text{names of symbols} & \\
 & & \text{(identifiers except } \textbf{dot}) &
\end{array}
$$

$$
\begin{array}{lcll}
(\textbf{store}\ (sf_1 \cdots)\ E[(cons\ v_1\ v_2)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_1\ v_2)))\ E[p]) & (\text{ECons}) \\
 & & (p \text{ fresh}) & \\
(\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[(car\ p)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[v_a]) & (\text{ECar}) \\
(\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[(cdr\ p)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots (p\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[v_d]) & (\text{ECdr}) \\
(\textbf{store}\ (sf_1 \cdots)\ E[(eqv?\ p\ p)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots)\ E[\#t]) & (\text{EEqv1}) \\
(\textbf{store}\ (sf_1 \cdots)\ E[(eqv?\ p_1\ p_2)]) & \rightarrow & (\textbf{store}\ (sf_1 \cdots)\ E[\#f]) & (\text{EEqv2}) \\
 & & (p_1 \neq p_2) & \\
(\textbf{store}\ (sf \cdots)\ E[((\textbf{lambda}\ (x \cdots)\ e)\ v \cdots)]) & \rightarrow & (\textbf{store}\ (sf \cdots)\ E[e[x \cdots / v \cdots]]) & (\text{EApp}) \\
 & & (\#x = \#v) & \\
 & & & \\
(\textbf{store}\ (sf \cdots)\ S[(\textbf{quote}\ sexp_1\ sexp_2 \cdots)] & \rightarrow & (\textbf{store}\ (sf \cdots)\ S[(\textbf{ccons}\ sexp_1\ (\textbf{quote}\ sexp_2)]) & (\text{EQuoteSeq}) \\
(\textbf{store}\ (sf \cdots)\ S[(\textbf{quote}\ ())] & \rightarrow & (\textbf{store}\ (sf \cdots)\ S[null]) & (\text{EQuoteNull}) \\
(\textbf{store}\ (sf \cdots)\ S[(\textbf{quote}\ n)] & \rightarrow & (\textbf{store}\ (sf \cdots)\ S[n]) & (\text{EQuoteNum}) \\
(\textbf{store}\ (sf \cdots)\ S[(\textbf{ccons}\ v_1\ v_2]) & \rightarrow & (\textbf{store}\ (sf \cdots (p\ (cons\ v_1\ v_2)))\ S[p]) & (\text{EQuotePair}) \\
 & & (p \text{ fresh}) & \\
 & & & \\
(\textbf{store}\ (sf \cdots)\ E[(eval\ v)]) & \rightarrow & (\textbf{store}\ (sf \cdots)\ E[\mathscr{R}\ [\![\ (sf \cdots), v\ ]\!]]) & (\text{EEval})
\end{array}
$$

$$
\begin{array}{l}
\mathscr{R} : (p \mapsto (cons\ v\ v)) \times v \rightarrow s \\
\mathscr{R}\ [\![\ S, null\ ]\!] = () \\
\mathscr{R}\ [\![\ S, n\ ]\!] = n \\
\mathscr{R}\ [\![\ S, \#t\ ]\!] = \#t \\
\mathscr{R}\ [\![\ S, \#f\ ]\!] = \#f \\
\mathscr{R}\ [\![\ S, (\textbf{quote}\ sy)\ ]\!] = sy \\
\mathscr{R}\ [\![\ S, p\ ]\!] = \mathscr{C}\ [\![\ \mathscr{R}\ [\![\ v_a\ ]\!], \mathscr{R}\ [\![\ v_d\ ]\!]\ ]\!] \\
\qquad \text{where } S \text{ binds } p \text{ to } (cons\ v_a\ v_d)
\end{array}
\qquad
\begin{array}{l}
\mathscr{C} : s \times s \rightarrow s \\
\mathscr{C}\ [\![\ sexp_1, (sexp_2 \cdots)\ ]\!] = (sexp_1\ sexp_2 \cdots) \\
\mathscr{C}\ [\![\ sexp_1, n\ ]\!] = (sexp_1\ \textbf{dot}\ n) \\
\mathscr{C}\ [\![\ sexp_1, sy\ ]\!] = (sexp_1\ \textbf{dot}\ sy) \\
\mathscr{C}\ [\![\ sexp_1, \#t\ ]\!] = (sexp_1\ \textbf{dot}\ \#t) \\
\mathscr{C}\ [\![\ sexp_1, \#f\ ]\!] = (sexp_1\ \textbf{dot}\ \#f)
\end{array}
$$

Figure 6. Core Scheme, extended with eval and quote

ement list whose first and third elements are symbols and whose second element is a list of one element:

```
(cons (quote lambda)
      (cons (cons (quote x) null)
            (cons (quote x) null)))
```

R^{5}RS suggests (but does not require) that quoted data be allocated only once, before the program runs. In systems with that behavior (including all Scheme implementations we tested), this program returns *#t*:

```
((lambda (f) (eqv? (f) (f)))
 (lambda () (quote (x))))
```

since the thunk passed as *f* returns the same result each time it is called.

Our core Scheme calculus for modeling *eval* and **quote** is shown in figure 6. (Note that this model simplifies R^{5}RS Scheme's *eval* procedure in that it does not accept an environment argument.) To ensure that a datum behind a **quote** is inserted into the store only once, the rewriting system is structured in two tiers roughly corresponding to "compile-time" and "run-time." Initially, programs are just viewed as uncompiled s-expressions (elements of the *s* nonterminal; note that we write dotted pairs with **dot** rather than a period to avoid meta-circular confusion in our PLT Redex implementation), which in particular include programs with quoted lists. Reduction rules that apply to these uncompiled expressions do not evaluate them, but instead compile them into program expressions that do not contain quoted lists (elements of the *e* nonterminal). Evaluation reductions only apply to a program after it has been completely compiled.

Each program consists of a store and an expression. Program expressions (*e*) can be applications, values, or identifiers. Evaluation contexts (*E*) dictate that evaluation takes place in a left to right order inside application expressions. The values (*v*) are procedures, quoted symbols, pointers (to cons cells), null, numbers, primitive operations, and booleans.

The first group of evaluation rules (from ECons to EApp) correspond to the language's runtime semantics, and show how the list primitives and procedure application behave. ECons models the application of *cons* to arguments by allocating a new pair in the store; and *car* and *cdr* select the first and second values in a pair by rules ECar and ECdr. EEqv1 and EEqv2 give *eqv?*'s semantics; it compares pointers for literal syntactic equality (and, for this language, operates only on pairs). As in the previous systems we have presented, procedure application is modeled by rule EApp as substitution. Since each reduction takes place in an *evaluation* (rather than *compilation*) context, they will only apply to programs that are completely compiled.

The second group of rules (from EQuoteSeq to EQuotePair) apply at compile-time and show how to compile a program by rewriting quoted constants into locations in the store. If those rules used the *E* context and quoted s-expressions were legal expressions, **quote** would merely be a short-hand notation for building lists at run-time and the above program would return *#f*, which would not capture our intended semantics.

$$
\begin{array}{llll}
p & ::= & (\textbf{store}\ ((x\ v)\cdots)\ (\textbf{dw}\ (dws\cdots)\ e)) & \\
e & ::= & \ldots \mid (\textbf{push}\ (x\ e\ e)) \mid (\textbf{pop}) & \\
v & ::= & \ldots \mid \textit{dynamic-wind} \mid \textit{call/cc} & \\
dws & ::= & (x\ e\ e) & \\
PC & ::= & (\textbf{store}\ ((x\ v)\cdots)\ DC) & \\
DC & ::= & (\textbf{dw}\ ((dws\cdots)\ C)) & \\
C & ::= & \text{(as in figure 1)} &
\end{array}
$$

$$
\begin{array}{lcll}
PC[(\textit{dynamic-wind}\ (\textbf{lambda}\ ()\ e_1)\ (\textbf{lambda}\ ()\ e_2)\ (\textbf{lambda}\ ()\ e_3))] & \rightarrow & PC[(\textbf{begin}\ e_1\ (\textbf{push}\ (x_1\ e_1\ e_3))\ ((\textbf{lambda}\ (x_2)\ (\textbf{begin}\ (\textbf{pop})\ e_3\ x_2))\ e_2))] & \text{(DWWIND)} \\
 & & (x_1, x_2 \text{ fresh}) & \\
PC[(\textbf{dw}\ (dws\cdots)\ C[(\textbf{push}\ x_2\ e_1\ e_2)])] & \rightarrow & PC[(\textbf{dw}\ (dws\cdots(x_2\ e_1\ e_2))\ C[\textit{0}])] & \text{(DWPUSH)} \\
PC[(\textbf{dw}\ (dws_1\cdots dws_n)\ C[(\textbf{pop})])] & \rightarrow & PC[(\textbf{dw}\ (dws_1\cdots)\ C[\textit{0}])] & \text{(DWPOP)} \\
PC[(\textbf{dw}\ (dws_1\cdots)\ C[(\textit{call/cc}\ v_1)])] & \rightarrow & PC[(\textbf{dw}\ (dws_1\cdots)\ C[(v_1\ (\textbf{lambda}\ (x)\ (\textbf{throw}\ (dws_1\cdots)\ C[x])))])] & \text{(DWCALLCC)} \\
 & & (x \text{ fresh}) & \\
PC[(\textbf{dw}\ (dws_1\cdots)\ C[(\textbf{throw}\ (dws_2\cdots)\ e_1)])] & \rightarrow & PC[(\textbf{dw}\ (dws_2\cdots)\ C[(\textbf{begin}\ \mathscr{T}[\![\ (dws_2\cdots), (dws_1\cdots)\ ]\!]\ e_1)])] & \text{(DWTHROW)}
\end{array}
$$

$$
\begin{array}{lcl}
\mathscr{T}[\![\ ((x_1\ e_1\ e_2)\ dws_1\cdots), ((x_1\ e_3\ e_4)\ dws_2\cdots)\ ]\!] & = & \mathscr{T}[\![\ (dws_1\cdots), (dws_2\cdots)\ ]\!] \\
\mathscr{T}[\![\ ((x_1\ e_1\ e_2)\cdots), ((x_2\ e_3\ e_4)\cdots)\ ]\!] & = & (\textbf{begin}\ e_2\cdots_r e_3\cdots) \\
 & & (x_1 \neq x_2)
\end{array}
$$

Figure 7. Additions to figure 1 to support call/cc and dynamic-wind

Instead, the second group of rewriting rules eliminate **quote**, turning s-expressions into Scheme programs. Though we have presented them second, these rules will actually come first in reduction sequences, making reduction sequences follow a two-phase pattern where the EQUOTE rules apply in the first phase and the evaluation rules apply in the second phase. Intuitively, programs in this first phase are arbitrary s-expressions and values are Scheme programs, whereas second-phase programs are Scheme expressions and values are Scheme values. This parallelism can be seen particularly clearly in the definition of the evaluation contexts for application expressions. In *S*, a rewrite may occur once all of the s-expressions to the left have become Scheme programs. In *E*, a rewrite may occur once all of the expressions to the left have become values. So, for the program above, the only rewriting rules that apply are those that rewrite the thunk's body. Once it contains only a pointer to a store value, the outer application can proceed.

To model *eval*, we use a technique similar to Muller's *reify* [18]. The $\mathscr{R}$ metafunction accepts a value and turns it back into a program (the $\mathscr{C}$ function is used by $\mathscr{R}$; it is just the syntactic analogue of *cons*). Once $\mathscr{R}$ completes, evaluation continues as usual. Of course, reification may produce an s-expression containing **quote**. In that case, the quote rules apply and put quoted date into the store before evaluation continues.[1]

[1] Most Scheme systems share quoted data even across calls to eval. For example, our semantics produces *#f* for the following program, but most Schemes produce *#t*.

```
((lambda (f)
   (eqv? (f)
         (eval (cons 'quote (cons (f) '())))))
 (lambda () '(x)))
```

We can adapt the definition of $\mathscr{R}$ to handle this by special handling of quoted forms during reification:

$\mathscr{R}[\![\ S, p_1\ ]\!] = v$ if S maps p_1 to (*cons* (**quote** *quote*) p_2) and maps p_2 to (*cons* v '()).

which causes our semantics to produce *#t* for the above example, but this technique does not scale to a full Scheme that includes macros.

As mentioned above, the *eval* we present here and in section 6 is not as full-featured as the *eval* of the R^{5}RS informal description because it does not accept an environment argument. Modeling an *eval* that took an environment argument would be somewhat more involved but would essentially require only running *eval*ed programs in an alternate store.

The technique used in this section applies generally to languages in which computation of a term proceeds in multiple phases that must be considered together — it is not sufficient in our case to write a preprocessor that moves quoted data in a program into the store because that program could call *eval* at runtime. Scheme's macros are similar in this respect, so the technique shown here could be used as a basis for modeling them. Staged and partial evaluation could also be modeled using this technique.

5. Call/cc and dynamic-wind

Scheme's *dynamic-wind* feature for annotating the dynamic extent of a procedure call with entry and exit code that run whenever the program flows into or out of that extent, either through normal program evaluation or through the invocation of continuation objects made by *call/cc* (the latter situation being the more interesting one, of course). Unfortunately, though *dynamic-wind* has a large impact on the meaning of continuation objects *call/cc* produces, the R^{5}RS formal semantics does not include any mention of it and models *call/cc* without respect to it. Here we will show how it works in the context of the core Scheme with mutation presented in section 2. Our strategy for modeling these new features is based heavily on earlier treatments [4, 10, 12].

The language in figure 7 consists of the core Scheme with mutation as shown in figure 1 augmented with *call/cc* and *dynamic-wind*. The basic strategy we take is to maintain a stack of all *dynamic-wind* calls entered but not yet exited, which we call the dynamic-wind stack. When we capture a continuation, we record the current dynamic-wind stack. When we throw to a continuation object, we use the difference between the current dynamic-wind stack and that recorded dynamic-wind stack to determine which *pre* and *post* thunks need to be called.

$$
\begin{array}{lcl}
p & ::= & (\textbf{store}\ ((ptr\ sv)\cdots)\ (\textbf{dw}\ (dws\cdots)\ e)) \\
e & ::= & (e\ e\cdots) \mid (\textbf{if}\ e\ e\ e) \mid (\textbf{if}\ e\ e) \mid (\textbf{set!}\ x\ e) \mid (\textbf{begin}\ e\ e\cdots) \\
 & \mid & (\textbf{throw}\ x\ dws\cdots \mathrm{EC}[e]) \mid (\textbf{push}\ (x\ e\ e)\ e) \mid (\textbf{pop}\ e) \\
 & \mid & lam \mid mulam \mid v \mid x \\
lam & ::= & (\textbf{lambda}\ (x\cdots)\ e\ e\cdots) \\
mulam & ::= & (\textbf{lambda}\ (x\cdots \textbf{dot}\ x)\ e\ e\cdots) \\
v & ::= & fun \mid nonfun \\
fun & ::= & cp \mid mp \mid \#\%cons \mid \#\%null? \mid \#\%pair? \\
 & \mid & \#\%car \mid \#\%cdr \mid \#\%set{-}car! \mid \#\%set{-}cdr! \mid \#\%list \\
 & \mid & \#\%+ \mid \#\%- \mid \#\%/ \mid \#\%* \mid \#\%call/cc \\
 & \mid & \#\%dynamic{-}wind \mid \#\%values \mid \#\%call{-}with{-}values \\
 & \mid & \#\%eqv? \mid \#\%apply \mid \#\%eval \\
nonfun & ::= & pp \mid number \mid \#\%null \mid \#t \mid \#f \\
 & \mid & (\textbf{quote}\ symbol) \mid unspecified \\
 & & \\
\mathrm{PC} & ::= & (\textbf{store}\ ((ptr\ sv)\cdots)\ \mathrm{DC}) \\
\mathrm{DC} & ::= & (\textbf{dw}\ (dws\cdots)\ \mathrm{EC}_\circ) \\
\mathrm{EC} & ::= & [] \mid (inert\cdots \mathrm{EC}_\circ{}^\circ\ inert\cdots) \\
 & \mid & (\textbf{if}\ \mathrm{EC}_\circ\ e\ e) \mid (\textbf{if}\ \mathrm{EC}_\circ\ e) \mid (\textbf{set!}\ x\ \mathrm{EC}_\circ) \\
 & \mid & (\textbf{begin}\ \mathrm{EC}\ e\ e\cdots) \\
 & \mid & (\#\%call{-}with{-}values^\circ\ (cwv\text{-}mark\ \mathrm{EC}_*)\ v^\circ) \\
\mathrm{EC}_\circ & ::= & [\,]_\circ \mid \mathrm{EC} \\
\mathrm{EC}_* & ::= & [\,]_* \mid \mathrm{EC} \\
inert & ::= & e \mid v^\circ
\end{array}
$$

$$
\begin{array}{lcl}
dws & ::= & (x\ cp\ cp) \\
sv & ::= & v \mid (\#\%cons\ v\ v) \mid lam \mid mulam \\
s & ::= & (s\cdots) \mid (s\cdots \textbf{dot}\ nss) \mid nss \\
nss & ::= & number \mid \#t \mid \#f \mid [\text{variable except } \textbf{dot}] \\
\mathrm{SC} & ::= & [] \mid (e\cdots \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{if}\ \mathrm{SC}\ s\ s) \mid (\textbf{if}\ e\ \mathrm{SC}\ s) \mid (\textbf{if}\ e\ e\ \mathrm{SC}) \\
 & \mid & (\textbf{if}\ \mathrm{SC}\ s) \mid (\textbf{if}\ e\ \mathrm{SC}) \\
 & \mid & (\textbf{set!}\ x\ \mathrm{SC}) \\
 & \mid & (\textbf{begin}\ \mathrm{SC}\ s\cdots) \mid (\textbf{begin}\ e\ e\cdots \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{throw}\ x\ dws\cdots \mathrm{SC}) \mid (\textbf{push}\ (x\ \mathrm{SC}\ s)\ s) \\
 & \mid & (\textbf{push}\ (x\ e\ \mathrm{SC})\ s) \mid (\textbf{push}\ (x\ e\ e)\ \mathrm{SC}) \mid (\textbf{pop}\ \mathrm{SC}) \\
 & \mid & (\textbf{lambda}\ (x\cdots)\ \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{lambda}\ (x\cdots)\ e\ e\cdots \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{lambda}\ (x\cdots \textbf{dot}\ x)\ \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{lambda}\ (x\cdots \textbf{dot}\ x)\ e\ e\cdots \mathrm{SC}\ s\cdots) \\
 & \mid & (\textbf{ccons}\ \mathrm{SC}\ s) \mid (\textbf{ccons}\ v\ \mathrm{SC}) \\
 & & \\
var & ::= & [\text{variable except } \textbf{dot} \text{ and keywords}] \\
x & ::= & [\text{variable names}] \\
pp & ::= & [\text{pair pointers}] \\
cp & ::= & [\text{closure pointers}] \\
mp & ::= & [\mu \text{ closure pointers}] \\
ptr & ::= & x \mid pp \mid cp \mid mp
\end{array}
$$

Figure 8. Grammar for full Scheme semantics

That strategy is formally encoded in three parts. First, we add a dynamic-wind stack to each program context. It contains one dynamic context frame (*dws*) for each annotated dynamic extent in which the current evaluation is taking place. A dynamic context frame is a triple consisting of a unique identifier and the *pre* and *post* thunks of the corresponding *dynamic-wind* call. The unique identifier allows us to disambiguate multiple dynamic evaluations of the same syntactic appearance of a *dynamic-wind* expression. Second, we add the primitive procedure value *dynamic-wind* to the set of values, which expects each of its three arguments to evaluate to a thunk. Then using the DWWIND rule it invokes its *pre* thunk, pushes a dynamic context frame onto the stack with a fresh identifier and its own *pre* and *post* thunks, evaluates its second thunk, pops its dynamic context frame off the stack, evaluates its *post* thunk, and finally returns the value its second thunk evaluated to. To allow the program to manipulate the stack, we introduce the **push** and **pop** forms and their associated reduction rules DWPUSH and DWPOP. The former pushes a new dynamic context frame onto the end of the stack, and the latter pops the last context frame off the stack (and then evaluates to the trivial value *0*, which is never used). These two forms are intended to be used only by *dynamic-wind*, never by the programmer directly.

The third piece is *call/cc*. When *call/cc* is called, the DWCALLCC rule builds a continuation object that consists of a procedure of one argument, a fresh identifier we will call *x*. That procedure's body is a **throw** form that consists of the current dynamic stack and the expression formed by plugging *x* into the hole of the evaluation context where the application of *call/cc* itself was found. A **throw** form is itself evaluated using the DWTHROW rule by discarding the evaluation context in which it was found, replacing the dynamic stack with its own stored dynamic stack, and replacing the entire program body with a specially-constructed **begin** expression built by the $\mathscr{T}$ metafunction (where T stands for "trim," because it trims away the common context frames leaving only the suffixes whose pre- or post-thunks need to be executed). That function compares its first argument, the dynamic-wind stack of the dynamic context being exited, with its second argument, the dynamic-wind stack of the context being entered. The first rule in its definition simply discards any common prefix the two stacks may have, which correspond to dynamic extents that were never left or entered during the transitions from the time the continuation object was created and the time it was invoked. Then, once the two stacks have been trimmed to the point where they have distinct heads, the metafunction produces a **begin** expression consisting of applications of all the *post* thunks from $\mathscr{T}$'s first argument, invoked in order, followed by all the *pre* thunks from $\mathscr{T}$'s second argument, invoked in reverse order (which we indicate with the special notation $\cdots_r$, indicating a sequence being expanded out backwards).

6. Operational semantics for R^{5}RS Scheme

This section combines the techniques from sections 2 through 5 with other known techniques for modeling programming languages to build a model of R^{5}RS Scheme that includes all the features from those sections along with **if** and booleans, mathematical operations (but not the numeric tower), list constructors, selectors, mutators and predicates, μ-lambda procedures[2], *apply*, and object identity-based equivalence. Although this section appears large and complex at first, it is mostly just a simple combination of the previous four sections.

This specification is executable, and the figures presented in this section were automatically generated from the source code that implements the specification. Since an executable specification was an explicit goal of our work, we have made some modeling choices that may not be obvious at first. For example, there are many expressions whose return values are explicitly unspecified in the R^{5}RS Scheme document, such as the result of a **set!** expression. A non-executable specification might model the evaluation of those expressions using the rule schema

$$\forall\, v.\ PC[unspecified] \rightarrow PC[v]$$

[2] Procedures declared with an improper list of formal arguments described in section 4.1.4 of the Report that accept an arbitrary number of arguments beyond a certain minimum. The name dates back at least to Indiana University's Scheme 84 system where `MULAMBDA` was a keyword used to declare procedures that accepted any number of arguments and collected them in a list [11].

meaning that an unspecified term reduces to any value. Instead, we model unspecified results with a special value *unspecified* that has no associated reduction rules and will cause programs that inspect it to get stuck. We also chose to ignore out-of-memory errors. These would be easy to add at the expense of a additional clutter when visualizing traces: reductions from each allocation site to the out-of-memory error would suffice.

6.1 Grammar

The grammar for R^{5}RS Scheme programs is given in figure 8. In that figure, a program (given by the *p* nonterminal) consists of a store, a dynamic-wind stack, and an expression. The *e* nonterminal gives expressions, which in addition to standard Scheme core forms can be **throw**, **push** and **pop**, as in section 5. Values (*v*) are either procedures or non-procedure values, but notice that syntactic **lambda** terms are not values themselves. Instead, procedure values (*fun*) can be references to procedures in the store (*cp* and *mp*) or the built-in procedures, while the **lambda** form, as we will see, places new procedure values into the store when evaluated. Non-procedure values (*nonfun*) include pair pointers, numbers, *null*, booleans, symbols, and the unspecified value.

As in section 4, we write dotted pairs (as in the parameter list of a μ-lambda) with **dot** rather than a period to avoid meta-circular confusion in our PLT Redex implementation.

Section 6 of the R^{5}RS Scheme specification indicates that primitive procedures are bound to names in the initial environment, but that those names can be mutated during the course of a program. To model that, we use special names with *#%* prefixes to indicate the actual built-in procedures, and we bind those values to their *#%*-less names in the initial store:

(**store** ((*list #%list*) (*cons #%cons*) (*car #%car*) (*cdr #%cdr*)
(*pair? #%pair?*) (*null #%null*) (*null? #%null?*)
(*set-car! #%set−car!*) (*set-cdr! #%set−cdr!*)
(+ *#%+*) (− *#%−*) (/ *#%/*) (∗ *#%∗*)
(*call/cc #%call/cc*) (*dynamic-wind #%dynamic−wind*)
(*values #%values*) (*call-with-values #%call−with−values*)
(*eqv? #%eqv?*) (*apply #%apply*) (*eval #%eval*)) ···)

There are three different contexts we will make use of: program evaluation contexts, dynamic-wind contexts, and expression contexts. Each program evaluation context (PC) contains a store, and a dynamic-wind context. Each dynamic-wind context (DC) contains a dynamic-wind stack and an expression context. Expression contexts (EC) are the contexts in which program evaluation takes place; they allow evaluation in marked sub-expressions of an application (as in section 2), the test positions of **if** expressions, in **set!** expressions and in the first position in a **begin** (as long as there are at least two expressions in the **begin**). The evaluation context for *#%call−with−values* is explained in section 6.7. The EC$_\circ$ and EC$_*$ evaluation contexts and *inert* work like $C_\circ$ and C_* and *inert* from section 3.

The *dws* non-terminal corresponds to one frame of dynamic-wind context information and its use is explained in section 5. The *sv* non-terminal generates values that appear in the store.

S-expressions (*s* and *nss*) and s-expression contexts (SC) correspond to s-expressions and s-expression contexts from section 4. There are more possible s-expression contexts in the full language because there are more possible syntactic forms.

Finally, the *x* nonterminal represents both program variables and binding locations, and the *pp*, *cp*, and *mp* nonterminals represent pointers to pairs, fixed-arity procedures, and variable-arity procedures, respectively. The *ptr* non-terminal is a short-hand for terms that index into the store. One subtle point here is that the *v* production produces *pp*, *cp*, and *mp* but not *x*. Those variables are not included because free variables are not values and bound variables have to be dereferenced before use, so neither qualifies as an irreducible value.

6.2 Relations

In the remaining figures, we will make heavy use of various reduction relation symbols. The basic reduction relation we will use is $\rightarrow$, which indicates that the program term on the left reduces in one step to the term on the right. We also use two other relations to aid in the system's readability, defined in terms of the $\rightarrow$ relation:

- $(e_1 \cdots e_n) \mapsto^\circ e'$ iff PC$[(e_1^\circ \cdots e_n^\circ)] \rightarrow$ PC$[e']$
  The application on the left reduces to the term on the right in a program context, assuming that all of the expressions in the application are marked.
- $e \rightarrow^e$ **error:** *s* iff PC$[e] \rightarrow$ **error:** *s*
  The term on the left signals an error, halting the program immediately.

6.3 Basic syntactic forms

Figure 9 shows rules for the basic syntactic forms. For the **if** form, if the test position evaluates to anything other than *#f*, the term rewrites to its "then" subexpression. If the test position evaluates to *#f*, it rewrites to its "else" subexpression, if present, *unspecified* otherwise. For the **begin** form, the evaluation contexts defined in figure 8 ensure that the first term of a **begin** expression containing at least two expressions is evaluated fully; then these rules cause **begin** expression that consists of a fully-evaluated value followed by one or more expressions to rewrite to a new **begin** expression with the initial value dropped. These rules also specify that a **begin** form with only a single expression reduces immediately to that expression, even if that expression is not yet a value.

Because our model does not take into account R^{5}RS Scheme's numeric tower, we model its numeric operations in terms of true mathematical functions. We assume that we can identify the true number represented by each numeric term and model each numeric procedure by performing the appropriate mathematical operation on those true numbers: + is modeled by summation on the represented numbers, $*$ is modeled by multiplication, and so on.

6.4 Cons and cons-cell mutation

The rules for constructing new *cons* cells are given in figure 10. Since all cons cells are mutable and therefore can be distinguished even when they hold identical values, we cannot allow (*#%cons v v*) to be a value itself. Instead, the *#%cons* rule introduces a new pair into the store and reduces to a pointer to that new pair. The (*#%list* v_1 ···) rule rewrites to ((**lambda** *x x*) v_1 ···), taking advantage of the μ-lambda application rules described in the next subsection.

Figure 11 gives rules for *car* and *cdr*. Application of either procedure to a pair pointer rewrites to the contents of the appropriate field in the pair being pointed to. If either selector is applied to a non-pair value, the term rewrites to an error message.

The predicates in figure 12 are similarly simple. The *#%pair?* procedure reduces to *#t* if its argument is identifiable as a pair pointer and *#f* otherwise. The *#%null?* procedure reduces to *#t* if and only if it is supplied with the built-in null value.

Figure 13 gives rules for *set-car!* and *set-cdr!*, for cons-cell mutation. The *#%set−car!* and *#%set−cdr!* rules are the same as the *car* and *cdr* rules, respectively, except that instead of reducing to the current value of appropriate component of the pair being pointed to, they replace that component with the given replacement then rewrite to an unspecified value.

6.5 Procedures and assignable variables

The rules in figure 14 handle variable lookup and variable assignment: a binding pointer is replaced with its value in the store when

$$
\begin{array}{lcl}
\text{PC}[(\textbf{if } v_1\ e_1\ e_2)] & \rightarrow & \text{PC}[e_1] \\
 & & (v_1 \neq \#f) \\
\text{PC}[(\textbf{if } \#f\ e_1\ e_2)] & \rightarrow & \text{PC}[e_2] \\
\text{PC}[(\textbf{if } v_1\ e_1)] & \rightarrow & \text{PC}[e_1] \\
 & & (v_1 \neq \#f) \\
\text{PC}[(\textbf{if } \#f\ e_1)] & \rightarrow & \text{PC}[\textit{unspecified}]
\end{array}
$$

$$
\begin{array}{lcl}
\text{PC}[(\textbf{begin } v\ e_1\ e_2\ \cdots)] & \rightarrow & \text{PC}[(\textbf{begin } e_1\ e_2\ \cdots)] \\
\text{PC}[(\textbf{begin } e_1)] & \rightarrow & \text{PC}[e_1] \\
(+\ \lceil n \rceil\ \cdots) & \mapsto^{\circ} & \lceil \Sigma\ n\ \cdots \rceil \\
(-\ \lceil n_1 \rceil\ \lceil n_2 \rceil\ \cdots) & \mapsto^{\circ} & \lceil n_1 - (\Sigma\ n_2\ \cdots) \rceil \\
(-\ \lceil n \rceil) & \mapsto^{\circ} & \lceil \text{-}n \rceil \\
(*\ \lceil n \rceil\ \cdots) & \mapsto^{\circ} & \lceil \Pi\ n\ \cdots \rceil \\
(/\ \lceil n_1 \rceil\ \lceil n_2 \rceil\ \cdots) & \mapsto^{\circ} & \lceil n_1 \,/\, (\Pi\ n_2\ \cdots) \rceil \\
(/\ \lceil n_1 \rceil) & \mapsto^{\circ} & \lceil 1 \,/\, n_1 \rceil
\end{array}
$$

Figure 9. Basic syntactic forms

$$
\begin{array}{lcl}
(\textbf{store } ((ptr_1\ sv_1)\ \cdots) \\
\quad \text{DC}[(\#\%cons^{\circ}\ v_{car}{}^{\circ}\ v_{cdr}{}^{\circ})]) & \rightarrow & (\textbf{store } ((ptr_1\ sv_1)\ \cdots\ (p_i\ (\#\%cons\ v_{car}\ v_{cdr}))) \\
 & & \quad \text{DC}[p_i]) \\
 & & (p_i\ \textit{fresh}) \\
\text{PC}[(\#\%list^{\circ}\ v_1{}^{\circ}\ \cdots)] & \rightarrow & \text{PC}[((\textbf{lambda } (\textbf{dot } l)\ l)^{\circ}\ v_1{}^{\circ}\ \cdots)]
\end{array}
$$

Figure 10. List constructors

$$
\begin{array}{lcl}
(\textbf{store } ((ptr_1\ sv_1)\ \cdots \\
\qquad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
\qquad (ptr_{i+1}\ sv_{i+1})\ \cdots) \\
\quad \text{DC}[(\#\%car^{\circ}\ pp_i{}^{\circ})]) & \rightarrow & (\textbf{store } ((ptr_1\ sv_1)\ \cdots \\
 & & \qquad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
 & & \qquad (ptr_{i+1}\ sv_{i+1})\ \cdots) \\
 & & \quad \text{DC}[v_{car}]) \\
(\#\%car^{\circ}\ v_i{}^{\circ}) & \rightarrow^{e} & \textbf{error:} \text{ can't take car of non-pair} \\
 & & (v_i \notin pp) \\
(\textbf{store } ((ptr_1\ sv_1)\ \cdots & \rightarrow & (\textbf{store } ((ptr_1\ sv_1)\ \cdots \\
\qquad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) & & \qquad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
\qquad (ptr_{i+1}\ sv_{i+1})\ \cdots) & & \qquad (ptr_{i+1}\ sv_{i+1})\ \cdots) \\
\quad \text{DC}[(\#\%cdr^{\circ}\ pp_i{}^{\circ})]) & & \quad \text{DC}[v_{cdr}]) \\
(\#\%cdr^{\circ}\ v_i{}^{\circ}) & \rightarrow^{e} & \textbf{error:} \text{ can't take cdr of non-pair} \\
 & & (v_i \notin pp)
\end{array}
$$

Figure 11. List accessors

$$
\begin{array}{lcl}
(\#\%null?\ \#\%null) & \mapsto^{\circ} & \#t \\
(\#\%null?\ v_i) & \mapsto^{\circ} & \#f \\
 & & (v_i \neq \#\%null) \\
(\#\%pair?\ pp) & \mapsto^{\circ} & \#t \\
(\#\%pair?\ v_i) & \mapsto^{\circ} & \#f \\
 & & (v_i \notin pp)
\end{array}
$$

Figure 12. List predicates

dereferenced, and mutation of a binding pointer is represented by replacing the value pointed to by the update. **lambda** is the only binding form in this semantics, so the rules for procedure calls are the only ones that introduce new bindings. Procedure calls are modeled by two features: closure introduction and procedure application.

The rules in figure 15 govern the introduction of closure values into the store. Like cons cells, procedures are not values, but pointers to them are; procedures are modeled this way so that we can model *eqv?* more accurately. The allocation rule for fixed-arity procedures is straightforward. The allocation for μ-lambda procedures always puts two procedures into the store: a stub μ-lambda procedure whose body contains a call to an ordinary procedure, and an ordinary procedure that contains the original μ-lambda's body expressions.

The reason for arranging the system this way is so that when a μ-lambda procedure is applied, we can rewrite it into a corresponding call to the fixed-arity code pointer and thereby use the same reduction for both kinds of applications. The rules in figure 16 show this and the rest of the rules for application in detail. The first rule shows how marks are placed in applications, which is just as in section 2. Application of a procedure pointer to arguments is modeled by creating one new binding pointer in the store per formal argument where the value being pointed to by each pointer is the argument supplied in the appropriate position, and rewriting to the procedure's body with these new bound-variable pointers substituted for occurrences of the formal arguments.

Application of a μ-lambda allocates a list for its extra arguments, applies the initial portion of the arguments as usual, and applies the extra arguments into the last argument of the procedure that actually contains the body expressions. The function $\mathcal{L}$ used here is a metafunction that builds the syntax of a *cons*-list from its arguments:

$$
\begin{array}{lcl}
\mathcal{L}\ [\![\ x\ y\ \cdots\ ]\!] & = & (\#\%cons\ x\ \mathcal{L}\ [\![\ y\ \cdots\ ]\!]) \\
\mathcal{L}\ [\![\ ]\!] & = & \#\%null
\end{array}
$$

The last rules specify the behavior of Scheme's *apply* procedure which accepts a procedure and an arbitrary number of arguments, the last of which must be a list. It calls the procedure with the arguments and the contents of the list as subsequent arguments. To model it, the first two *#%apply* rules flatten out the argument list and, when the list is exhausted, reduce to a normal application.

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) & & \quad (pp_i\ (\#\%cons\ v_{new}\ v_{cdr})) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(\#\%set\text{-}car!^{\circ}\ pp_i{}^{\circ}\ v_{new}{}^{\circ})])] & & \ \mathrm{DC}[unspecified])
\end{array}
$$

$$
(\#\%set\text{-}car!^{\circ}\ v_1{}^{\circ}\ v^{\circ}) \to^{e} \textbf{error:}\ \text{can't set-car! on a non-pair } (v_1 \notin pp)
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) & & \quad (pp_i\ (\#\%cons\ v_{car}\ v_{new})) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(\#\%set\text{-}cdr!^{\circ}\ pp_i{}^{\circ}\ v_{new}{}^{\circ})])] & & \ \mathrm{DC}[unspecified])
\end{array}
$$

$$
(\#\%set\text{-}cdr!^{\circ}\ v_1{}^{\circ}\ v^{\circ}) \to^{e} \textbf{error:}\ \text{can't set-cdr! on a non-pair } (v_1 \notin pp)
$$

Figure 13. Cons cell mutation

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ \mathrm{DC}[x_i]) & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots\ (x_i\ sv_i)\ (ptr_{i+1}\ sv_{i+1})\cdots)\ \mathrm{DC}[sv_i]) \\
(\textbf{store}\ ((ptr_1\ sv_1)\cdots\ (x_i\ sv_i)\ (ptr_{i+1}\ sv_{i+1})\cdots)\ \mathrm{DC}[(\textbf{set!}\ x_i\ v_{new})]) & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots\ (x_i\ v_{new})\ (ptr_{i+1}\ sv_{i+1})\cdots)\ \mathrm{DC}[unspecified])
\end{array}
$$

Figure 14. Variable mutation and lookup

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ \mathrm{DC}[lam_i]) & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots\ (cp_i\ lam_i))\ \mathrm{DC}[cp_i])\ (cp_i\ fresh) \\
(\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ \mathrm{DC}[(\textbf{lambda}\ (x_1\cdots\textbf{dot}\ x_r)\ e_1\ e_2\cdots)]) & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots\ (mp_i\ (\textbf{lambda}\ (x_1\cdots\textbf{dot}\ x_r)\ (cp_i\ x_1\cdots x_r)))\ (cp_i\ (\textbf{lambda}\ (x_1\cdots x_r)\ e_1\ e_2\cdots))) \\
 & & (mp_i, cp_i\ fresh)
\end{array}
$$

Figure 15. Procedure introduction

$$
\mathrm{PC}[(inert_1\cdots e_i\ inert_{i+1}\cdots)] \to \mathrm{PC}[(inert_1\cdots e_i{}^{\circ}\ inert_{i+1}\cdots)]
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (cp_i\ (\textbf{lambda}\ (x_1\cdots)\ e_{body1}\ e_{body2}\cdots)) & & \quad (cp_i\ (\textbf{lambda}\ (x_1\cdots)\ e_{body1}\ e_{body2}\cdots)) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots \\
\ \mathrm{DC}[(cp_i{}^{\circ}\ v_{arg1}{}^{\circ}\cdots)]) & & \quad (x_{arg2}\ v_{arg1})\cdots) \\
 & & \ \mathrm{DC}[(\textbf{begin}\ e_{body1}\ e_{body2}\cdots)[x_1\cdots/x_{arg2}\cdots])]) \\
 & & (\#x_{arg} = \#v_{arg}\ , x_{arg2}\cdots\ fresh)
\end{array}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr\ sv)\cdots & \to & \textbf{error:}\ \text{arity mismatch} \\
\quad (cp_i\ (\textbf{lambda}\ (x_1\cdots)\ e\ e\cdots)) & & (\#x_{arg} \neq \#v_{arg}) \\
\quad (ptr\ sv)\cdots) & & \\
\ \mathrm{DC}[(cp_i{}^{\circ}\ v_{arg1}{}^{\circ}\cdots)]) & &
\end{array}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (mp_i\ (\textbf{lambda}\ (x_1\cdots\textbf{dot}\ y)\ (cp_t\ x_1\cdots y))) & & \quad (mp_i\ (\textbf{lambda}\ (x_1\cdots\textbf{dot}\ y)\ (cp_t\ x_1\cdots y))) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(mp_i\ v_{n1}{}^{\circ}\cdots v_R{}^{\circ}\cdots)]) & & \ \mathrm{DC}[(cp_t\ v_{n1}{}^{\circ}\cdots\mathcal{L}\ [\![\ v_R{}^{\circ}\cdots\ ]\!]\ )]) \\
 & & (\#x = \#v_n)
\end{array}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr\ sv)\cdots & \to & \textbf{error:}\ \text{too few arguments} \\
\quad (mp_i\ (\textbf{lambda}\ (x_1\cdots\textbf{dot}\ x)\ (cp\ x\cdots))) & & (\#x_{arg} < \#v_{arg}) \\
\quad (ptr\ sv)\cdots) & & \\
\ \mathrm{DC}[(mp_i{}^{\circ}\ v_{arg1}{}^{\circ}\cdots)]) & &
\end{array}
$$

$$
(nonfun^{\circ}\ v^{\circ}\cdots) \to^{e} \textbf{error:}\ \text{can't apply non-function}
$$

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots & \to & (\textbf{store}\ ((ptr_1\ sv_1)\cdots \\
\quad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) & & \quad (pp_i\ (\#\%cons\ v_{car}\ v_{cdr})) \\
\quad (ptr_{i+1}\ sv_{i+1})\cdots) & & \quad (ptr_{i+1}\ sv_{i+1})\cdots) \\
\ \mathrm{DC}[(\#\%apply^{\circ}\ v_f{}^{\circ}\ v_{arg1}{}^{\circ}\cdots pp_i{}^{\circ})]) & & \ \mathrm{DC}[(\#\%apply^{\circ}\ v_f{}^{\circ}\ v_{arg1}{}^{\circ}\cdots v_{car}{}^{\circ}\ v_{cdr}{}^{\circ})])
\end{array}
$$

$$
(\#\%apply\ v_f\ v_{arg1}\cdots\#\%null) \mapsto^{\circ} (v_f\ v_{arg1}\cdots)
$$

$$
(\#\%apply^{\circ}\ v_f{}^{\circ}\ v_{arg1}{}^{\circ}\cdots v_{last}{}^{\circ}) \to^{e} \textbf{error:}\ \text{apply must take a list as its last argument } (v_{last} \notin pp \cup \{\#\%null\})
$$

Figure 16. Procedure application

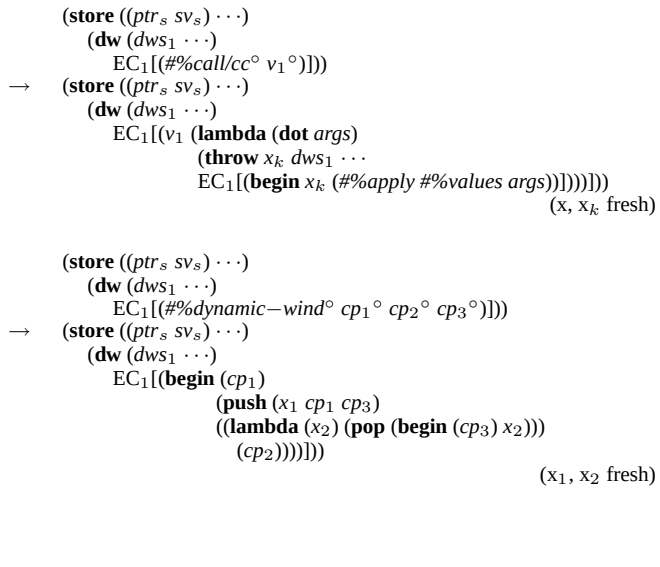

Figure 17. Call/cc and dynamic-wind

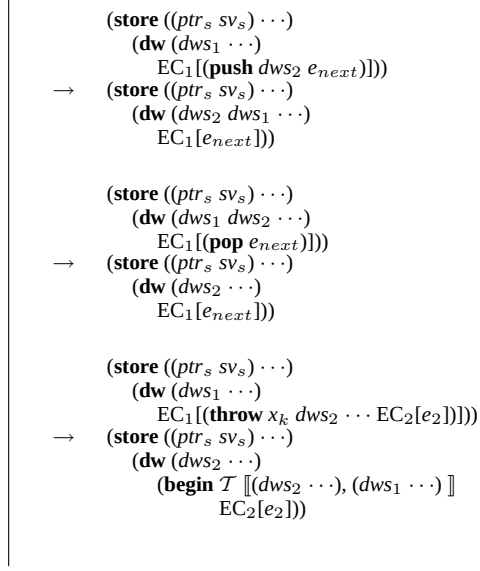

Figure 18. Call/cc and dynamic-wind support

PC[v_1]$_*$	→	PC[(*#%values*° v_1°)]
PC[(*#%values*° v_1°)]$_\circ$	→	PC[v_1]
PC[(*#%values*° v_1° ···)]$_\circ$	→	**error:** wrong number of values (*#$v_1 \neq 2$*)
(*#%call−with−values* v_{vals} v_{fun})	$\mapsto^\circ$	(*#%call−with−values*° (*cwv-mark* (v_{vals})) v_{fun}°)
PC[(*#%call−with−values*° (*cwv-mark* (*#%values*° v_{arg}° ···)) v_{fun}°)]	→	PC[(v_{fun}° v_{arg}° ···)]
(*#%call−with−values*° v_i° ···)	$\rightarrow^e$	**error:** arity mismatch (*#$v_i \neq 2$*)

Figure 19. Multiple values and call-with-values

(*#%eqv? pp$_i$ pp$_i$*)	$\mapsto^\circ$	*#t*
(*#%eqv? cp$_i$ cp$_i$*)	$\mapsto^\circ$	*#t*
(*#%eqv? number$_1$ number$_1$*)	$\mapsto^\circ$	*#t*
(*#%eqv? v_1 v_1*)	$\mapsto^\circ$	*#t*
PC[(*#%eqv?*° v_1° v_2°)]	→	PC[*#f*] (*$v_1 \neq v_2$*)
(*#%eqv?*° v_1° ···)	$\rightarrow^e$	**error:** arity mismatch (*#$v_1 \neq 2$*)

Figure 20. Eqv and equivalence

6.6 Call/cc

Our technique for modeling *call/cc* and *dynamic-wind*, shown in figures 17 and 18, is essentially the technique from section 5. Apart from the change of using procedure pointers rather than the literal source text of procedures as required to model equality (see section 6.8), the only substantial change is that the continuation procedures in this model accept any number of arguments. The trimming metafunction $\mathscr{T}$ is the same function defined in section 5.

6.7 Multiple values and call-with-values

Multiple values in the full language are nearly identical to multiple values in section 3, and in particular the context arrangement and promotion and demotion rules are the same. Furthermore, even though the present system is much larger than the system presented in section 3, the rules for multiple values are still completely orthogonal to the rules that implement the other features.

There is one twist, though, since rather than the **apply-values** primitive given in section 3, R^{5}RS Scheme provides *call-with-values*, so we model it directly. To do so, we have to use the mechanisms described in section 3, along with a new context containing *cwv-mark*. A term of the form (*#%call−with−values thunk f*) reduces to (*#%call−with−values* (*cwv-mark* (*thunk*)) *f*); that is, it places a special mark around the application of the thunk to no arguments. At that point the evaluation contexts defined in figure 8 will apply and reduce the applied thunk in a multi-value context. When that reduction sequence yields a result (which will be a multiple-values expression), the entire *call-with-values* expression reduces to the application of the second procedure to those produced values.

6.8 Eqv? and equivalence

Figure 20 shows the rules for *eqv?*. Since all mutable values (and procedures) are allocated in the store, *eqv?* is a simple matter

$$
\begin{array}{lcl}
(\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ \mathrm{DC}[(\#\%eval^{\circ}\ v_1{}^{\circ})]) & \rightarrow & (\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ \mathrm{DC}[\mathscr{R}\ [\![((ptr_1\ sv_1)\cdots), v_1]\!]]) \\
(\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ (\textbf{dw}\ (dws_1\cdots)\ \mathrm{EC}_1[\mathrm{SC}_1[(\textbf{quote}\ (s_1\ s_2\cdots))]])) & \rightarrow & (\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ (\textbf{dw}\ (dws_1\cdots)\ \mathrm{EC}_1[\mathrm{SC}_1[(\textbf{ccons}\ (\textbf{quote}\ s_1)\ (\textbf{quote}\ (s_2\cdots)))]])) \\
(\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ (\textbf{dw}\ (dws_1\cdots)\ \mathrm{EC}_1[\mathrm{SC}_1[(\textbf{quote}\ ())]])) & \rightarrow & (\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ (\textbf{dw}\ (dws_1\cdots)\ \mathrm{EC}_1[\mathrm{SC}_1[\#\%null]])) \\
(\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ (\textbf{dw}\ (dws_1\cdots)\ \mathrm{EC}_1[\mathrm{SC}_1[(\textbf{quote}\ number_1)]])) & \rightarrow & (\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ (\textbf{dw}\ (dws_1\cdots)\ \mathrm{EC}_1[\mathrm{SC}_1[number_1]])) \\
(\textbf{store}\ ((ptr_1\ sv_1)\cdots)\ (\textbf{dw}\ (dws_1\cdots)\ \mathrm{EC}_1[\mathrm{SC}_1[(\textbf{ccons}\ v_1\ v_2)]])) & \rightarrow & (\textbf{store}\ ((ptr_1\ sv_1)\cdots(pp_1\ (\#\%cons\ v_1\ v_2)))\ (\textbf{dw}\ (dws_1\cdots)\ \mathrm{EC}_1[\mathrm{SC}_1[pp_1]]))\ (pp_1\ \mathit{fresh})
\end{array}
$$

Figure 21. Quote and eval

of checking that the two values supplied have identical syntactic structure (which we indicate here, as PLT Redex does, by repeating the same subscript for both arguments to the *eqv?* procedure to indicate that the two subterms must be identical).

6.9 Quote and eval

The rules for *#%eval* and **quote** in figure 21 are essentially the same as the rules for *eval* and **quote** in section 4. The main difference is that the rewriting rules for replacing quote are nested an SC context inside an EC context. This only matters when using *#%eval*. In particular, if the call to *#%eval* is in some marked context, SC will not match properly, due to the marks. In the smaller calculus, we could get away with just using SC, since it also encompassed evaluation contexts, but here we must be explicit. The reify function ($\mathscr{R}$) used here is as defined in section 4.

7. Related Work

Reduction semantics has been used to model large programming languages many times and in many different ways. Felleisen's dissertation [3], 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," and Felleisen extends that model into the λ-v-CS calculus in later work [4]. Since then, reduction semantics have been used to model the cores of many languages including Emacs Lisp [19], MultiLisp [7], Java [9], ML [13, 24] and Concurrent ML [21] 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 a core Scheme in the development of the notion of space efficiency [2]. Gasbichler, Knauel, Sperber, and Kelsey have presented operational and denotational semantics for *dynamic-wind* [12]. Ramsdell presented a structural operational semantics for Scheme aimed at fixing the unspecified order of argument evaluation problem we discuss in subsection 2 [20]. His model is less complete than ours (for instance, it does not include multiple return values) and is tied much more closely to the R^{5}RS Scheme formal semantics. Van Straaten has written an interpreter based on the R^{5}RS Scheme denotational semantics [23], but we know of no formal correspondence between his program and the denotational semantics itself.

8. Conclusion

We have presented a semantics for R^{5}RS Scheme using context-sensitive reduction semantics developed using PLT Redex. To the best of our knowledge, it formalizes more of the language than any other semantics for the language. In addition it shows how to model R^{5}RS Scheme-style multiple return values in an small-step operational semantics setting for the first time, and gives a new model for unspecified sequential evaluation orders that uses nondeterministic choice. In the process, we have introduced several new techniques for modeling programming language features with term rewriting.

PLT Redex and the source code for all the models presented in this paper, including our executable model of R^{5}RS Scheme, are available for download at

http://www.cs.uchicago.edu/~jacobm/r5rs/

Acknowledgments

Thanks to Kent Dybvig and Matthew Flatt for helpful discussions of the technical details presented here and the inner workings of Chez Scheme [1] and MzScheme [8]. Thanks also to John Reppy and Dave MacQueen and the anonymous reviewers for their helpful suggestions.

References

[1] Cadence Research Systems. *Chez* Scheme Reference Manual, 1994.

[2] 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.

[3] 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.

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

[5] Matthias Felleisen and Matthew Flatt. Programming languages and lambda calculi. Available online: http://www.cs.utah.edu/plt/publications/pllc.pdf, 2003.

[6] Matthias Felleisen and Robert Hieb. The revised report on the syntactic theories of sequential control and state. *Theoretical*

Computer Science, 102:235–271, 1992. Original version in: Technical Report 89-100, Rice University, June 1989.

[7] Cormac Flanagan and Matthias Felleisen. The semantics of future and an application. *Journal of Functional Programming*, 9:1–31, 1999.

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

[9] 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.

[10] Daniel P. Friedman and Christopher T. Haynes. Constraining control. In *Proceedings of the ACM Conference Principles of Programming Languages*, 1985.

[11] Daniel P. Friedman, Christopher T. Haynes, Eugene Kohlbecker, and Mitchell Wand. Scheme 84 interim reference manual. Technical Report 153, Indiana University Computer Science, 1985.

[12] 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.

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

[14] David Herman and Philippe Meunier. Improving the static analysis of embedded languages via partial evaluation. In *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, pages 16–27, New York, NY, USA, 2004. ACM Press.

[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] Jacob Matthews. Operational semantics for Scheme via term rewriting. Technical Report TR-2005-02, University of Chicago, 2005.

[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] Robert Muller. M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transactions on Programming Languages and Systems*, 14(4), 1992.

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

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

[22] Gerald Jay Sussman and Jr Guy Lewis Steele. Scheme: An interpreter for extended lambda calculus. Technical Report AI Lab Memo AIM-349, MIT AI Lab, 1975.

[23] Anton van Straaten. An executable denotational semantics for Scheme. http://www.appsolutions.com/SchemeDS/.

[24] 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.