Abstract
This paper presents an operational semantics for the core of Scheme. Our specification improves over the existing R5RS 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 R5RS 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 R5RS model shows that they scale to real languages.

1. Introduction
The Revised5 Report on the Algorithmic Language Scheme [15], R5RS, 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 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 R5RS formal semantics that models more of the language described in the informal semantics section than the formal semantics section in the R5RS 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 R5RS document deliberately leaves unspecified the order in which arguments are evaluated, but section 4.1.3 specifies that

\[ \text{the effect of any concurrent evaluation of the operator and operand expressions is constrained to be consistent with some sequential order of evaluation. The order of evaluation may be chosen differently for each procedure call.} \]

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

\[ [w]e \text{ mimic [the order of evaluation] by applying arbitrary permutations permute and unpermute . . . to the arguments in a call before and after they are evaluated. This is not quite} \]

\[ \text{\dots} \]
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 AS [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 . . . to indicate any number of occurrences, 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 ‘n’ 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:

\[ C ::= (e \cdots C v \cdots) \]

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:

\[ C ::= (e \cdots C e \cdots) \]

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

\[ (\lambda (x y) y (- (I) (- 2))) \]

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 R5RS Scheme. It is not. Consider this application of two set! expressions in a store binding x to I.

\[ (\text{store } ((x I)) (\langle \text{set! } x (\langle x (\text{set!} x (- 3)) \rangle)) \] (set! x (- 3)))

In Scheme, this program should always reduce to the application of zero to zero with x set to I in the store (and then get stuck). According to R5RS, 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 –I 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 R5RS-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
that effect, we introduce the non-terminal inert and the notion of a marked expression, denoted with the ○ superscript. (These marks are not an extension to the general term-rewriting framework — ○ 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 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 sub-expression 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 I, 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))
```

It will either 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 R5RS 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 → p′, where p′ is also closed, p → 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

R5RS 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...
to its second argument. So, a programmer could supply \( f \)'s results to \( g \) like this:

\[
\text{(call-with-values (lambda () (f 3)) g)}
\]

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

To model multiple values, \( R^5 \text{RS} \) 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 continuation to accept multiple values and contexts that reject multiple values 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 \( R^5 \text{RS} \) Scheme:

\[
\text{(define (call-with-values thunk f)} \ (\text{apply-values f (thunk)})
\]

\[
\text{(define-syntx apply-values)} \ (\text{(syntax-rules} () \ ([\_f \_\_expr] \ (call-with-values (lambda () \_\_expr) f)))
\]

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 values.
The context-matching syntax is now annotated with $C$ parallel context nonterminals. The context indicates a hole in which either result is acceptable. There are three expressions of apply-values, however, is in which any expression should reduce to an element of $C$. If we wanted to allow any expression to produce either.

Since each subexpression of an application is expected to produce a single value, the evaluation context inside an application is $C_0$. For the same reason, the evaluation context for the first subexpression of apply-values is $C_0$. The evaluation context for the second subexpression, however, is $C_e$ 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 $Vapses and $VDEMOTES reduction) may produce a single value or $(values v \cdots)$, they take place in $[\_]_e$ 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. VDEMOTES demotes a single value inside values to just the value, and VDEMOTESERR 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 $C_e$ 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 $C_e$ with $C_0$ in all rules.

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

\[
\begin{align*}
\lambda y \cdot \lambda x \cdot (values y) & \rightarrow \lambda y \cdot (apply-values (lambda (x) (values x)) (values y)) \\
& \rightarrow (values y) \\
& \rightarrow (values y) \\
& \rightarrow (values y) \\
\end{align*}
\]

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 VDEMOTES 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 VDEMOTESERR rule.

\[
\begin{align*}
(g (f 3)) & \rightarrow \cdots \\
& \rightarrow (g (values 3 9)) \\
& \rightarrow error: expected a single value
\end{align*}
\]

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 $(store (lx v \cdots) v)$.

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

\[(\text{cons} \ (\text{quote} \ \text{lambda}) \ (\text{cons} \ (\text{quote} \ x) \ \text{null}) \ (\text{cons} \ (\text{quote} \ x) \ \text{null}))\]

\[\text{R}^3\text{RS} \text{suggests (but does not require) that quoted data be allocated only once, before the program runs. In systems with that behavior (including all Scheme implementations we tested), this program returns \#f:}\]

\[((\text{lambda} \ (f) \ (\text{eqv?} \ (f) \ (f))) \ (\text{lambda} \ () \ (\text{quote} \ (x)))\]

since the thunk passed as \(f\) returns the same result each time it is called.

Our core Scheme calculus for modeling \textit{eval} and \textit{quote} is shown in figure 6. (Note that this model simplifies \textit{R}^3\text{RS} Scheme’s \textit{eval} procedure in that it does not accept an environment argument.) To ensure that a datum behind a \textit{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 \text{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 \text{cons} cells), null, numbers, primitive operations, and booleans.

The first group of evaluation rules (from \text{ECONS} to \text{EAPP}) correspond to the language’s runtime semantics, and show how the list primitives and procedure application behave. \text{ECONS} models the application of \text{cons} to arguments by allocating a new pair in the store; and \text{car} and \text{cdr} select the first and second values in a pair by rules \text{ECAr} and \text{ECDr}. \text{EEqV1} and \text{EEqV2} give \text{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 \text{EAPP} as substitution. Since each reduction takes place in an \textit{evaluation} (rather than \textit{compilation}) context, they will only apply to programs that are completely compiled.

The second group of rules (from \text{EQuoteSeq} to \text{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, \textit{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.

---

\[\begin{align*}
\varepsilon & ::= \ (\varepsilon \ \ldots) \ v \ | \ v \\
E & ::= \ | \ (\varepsilon \ \ldots) \\
\nu & ::= \ (\lambda \ (\ldots) \ v) \ (\text{quote} \ s) \\
& | \ p \ | \ \text{null} \ | \ \text{prim} \ | \ \#f \\
\text{prim} & ::= \ \text{eval} \ | \ \text{cons} \ | \ \text{car} \ | \ \text{cdr} \ | \ \text{eqv?} \\
p & ::= \ \text{pointers} \\
x & ::= \ \text{program variables} \\
\text{members of sy except lambda, quote, cons} \\
s & ::= \ (\ldots) \ n \ | \ sy \\
\end{align*}\]
Instead, the second group of rewriting rules eliminate \texttt{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 \texttt{EQ\_QUOTE} 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 \texttt{eval}, we use a technique similar to Muller’s \texttt{reify} [18]. The \( F \) metafunction accepts a value and turns it back into a program (the \( C \) function is used by \( F \); it is just the syntactic analogue of \texttt{cons}). Once \( F \) completes, evaluation continues as usual. Of course, reification may produce an s-expression containing \texttt{quote}. In that case, the quote rules apply and put quoted data into the store before evaluation continues.\(^1\)

\(^1\) Most Scheme systems share quoted data even across calls to \texttt{eval}. For example, our semantics produces \#f for the following program, but most Schemes produce \#

\[
\begin{align*}
p & := \text{(lambda (f) (quote (f)) agent (begin f))} \\
e & := \text{(quote (quote (quote (f))))}
\end{align*}
\]

We can adapt the definition of \( F \) to handle this by special handling of quoted forms during reification:

\[
F[\langle S, p_1 \rangle] = v \quad \text{if } S \text{ maps } p_1 \text{ to } \langle \text{quote } v \text{ quote } \langle \text{quote } v \rangle \rangle.
\]

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

As mentioned above, the \texttt{eval} we present here and in section 6 is not as full-featured as the \texttt{eval} of the R²\_RS informal description because it does not accept an environment argument. Modeling an \texttt{eval} that took an environment argument would be somewhat more involved but would essentially require only running \texttt{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 \texttt{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. \texttt{call/cc} and dynamic-wind

Scheme’s \texttt{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 \texttt{call/cc} (the latter situation being the more interesting one, of course). Unfortunately, though \texttt{dynamic-wind} has a large impact on the meaning of continuation objects \texttt{call/cc} produces, the R²\_RS formal semantics does not include any mention of it and models \texttt{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 \texttt{call/cc} and \texttt{dynamic-wind}. The basic strategy we take is to maintain a stack of all \texttt{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 \texttt{pre} and \texttt{post} thunks need to be called.
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 dynamic stack with its own stored dynamic stack, and replacing the entire program body with a specially-constructed begin expression.

Second, we add the primitive procedure value call. The unique identifier allows us to disambiguate multiple dynamic evaluations of the same syntactic appearance of a dynamic-wind expression. We introduce the evaluation context where the application of call/cc itself was found.

The third piece is callcc. When callcc is called, the DW-CALLCC 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 callcc itself was found. A throw form is itself evaluated using the DWT-THROW 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.

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^2RS 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

\[
\forall v. \text{PC}(\text{unspecified}) \rightarrow \text{PC}(v)
\]
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 RRS 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, boolean, symbols, and the unspecified value.

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

Section 6 of the RRS 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 \texttt{%} prefixes to indicate the actual built-in procedures, and we bind those values to their \texttt{%}-less names in the initial store:

\begin{verbatim}
(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) )
\end{verbatim}

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 \textit{begin}). The evaluation context for \texttt{%call-with-values} is explained in section 6.7. The EC, and EC, evaluation contexts and inert work like \texttt{C}, and \texttt{C}, and \texttt{inert} from section 3.

The \texttt{dws} non-terminal corresponds to one frame of dynamic-wind context information and its use is explained in section 5. The \texttt{s} 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 n 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 \texttt{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 \texttt{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 \texttt{→}, 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 \texttt{→} relation:

- \texttt{e \rightarrow \texttt{error}: s} if \texttt{PC[e] → \texttt{error}: s}
The term 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.

- \texttt{e \rightarrow \texttt{error}: s} if \texttt{PC[e] → \texttt{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 \texttt{#f}, the term rewrites to its “then” subexpression. If the test position evaluates to \texttt{#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 \texttt{begin} expression containing at least two expressions is evaluated fully; then these rules cause \texttt{begin} expression that consists of a fully-evaluated value followed by one or more expressions to rewrite to a new \texttt{begin} expression with the initial value dropped. These rules also specify that a \texttt{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 RRS 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: \texttt{+} is modeled by summation on the represented numbers, \texttt{*} 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 (\texttt{%cons v v}) to be a value itself. Instead, the \texttt{%cons} rule introduces a new pair into the store and reduces to a pointer to that new pair. The \texttt{%cons} \texttt{v} \texttt{r} \texttt{c} rule rewrites to (\texttt{(lambda x1 x2 \ldots)}), taking advantage of the \mu-\textit{lambda} application rules described in the next subsection.

Figure 11 gives rules for \texttt{car} and \texttt{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 \texttt{%pair?} procedure reduces to \texttt{#t} if its argument is identifiable as a pair pointer and \texttt{#f} otherwise. The \texttt{%null?} procedure reduces to \texttt{#f} if and only if it is supplied with the built-in null value.

Figure 13 gives rules for \texttt{set-car!} and \texttt{set-cdr!}, for cons-cell mutation. The \texttt{%set-car!} and \texttt{%set-cdr!} rules are the same as the \texttt{car} and \texttt{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
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-λ procedures always puts two procedures into the store: a stub μ-λ procedure whose body contains a call to an ordinary procedure, and an ordinary procedure that contains the original μ-λ’s body expressions.

The reason for arranging the system this way is so that when a μ-λ 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-λ allocates a list for its extra arguments: 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 `L` used here is a metafunction that builds the syntax of a `cons`-list from its arguments:

\[
(\text{L} [ x\ y\ \ldots ] ) = (\text{#%cons } x\ \text{L} [ y\ \ldots ] ) \\
(\text{L} [ ] ) = \text{#fnull}
\]

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.
Figure 13. Cons cell mutation

\[
\begin{align*}
(\text{store } (\text{ptr } sv_1) \cdots) & \rightarrow (\text{store } (\text{ptr } sv_1) \cdots) \\
\text{DC}(\lambda x) & \rightarrow (\text{store } (\text{ptr } sv_1) \cdots) \\
\text{DC}(\lambda x) & \rightarrow (\text{store } (\text{ptr } sv_1) \cdots)
\end{align*}
\]

Figure 14. Variable mutation and lookup

\[
\begin{align*}
(\text{store } (\text{ptr } sv_1) \cdots) & \rightarrow (\text{store } (\text{ptr } sv_1) \cdots) \\
\text{DC}(\lambda x) & \rightarrow (\text{store } (\text{ptr } sv_1) \cdots) \\
\text{DC}(\lambda x) & \rightarrow (\text{store } (\text{ptr } sv_1) \cdots)
\end{align*}
\]

Figure 15. Procedure introduction

\[
\begin{align*}
\text{PC}(\text{inert}_1 \cdots e_1 \text{ inert}_{i+1} \cdots) & \rightarrow \text{PC}(\text{inert}_1 \cdots e_i \text{ inert}_{i+1} \cdots) \\
(\text{store } (\text{ptr } sv) \cdots) & \rightarrow (\text{store } (\text{ptr } sv) \cdots) \\
\text{DC}(\text{inert}_1 \cdots e_1 \text{ inert}_{i+1} \cdots) & \rightarrow \text{DC}(\text{inert}_1 \cdots e_i \text{ inert}_{i+1} \cdots) \\
(\text{store } (\text{ptr } sv) \cdots) & \rightarrow (\text{store } (\text{ptr } sv) \cdots) \\
\text{DC}(\text{inert}_1 \cdots e_1 \text{ inert}_{i+1} \cdots) & \rightarrow \text{DC}(\text{inert}_1 \cdots e_i \text{ inert}_{i+1} \cdots)
\end{align*}
\]

Figure 16. Procedure application

\[
\begin{align*}
\text{PC}(\text{inert}_1 \cdots e_1 \text{ inert}_{i+1} \cdots) & \rightarrow \text{PC}(\text{inert}_1 \cdots e_i \text{ inert}_{i+1} \cdots) \\
(\text{store } (\text{ptr } sv) \cdots) & \rightarrow (\text{store } (\text{ptr } sv) \cdots) \\
\text{DC}(\text{inert}_1 \cdots e_1 \text{ inert}_{i+1} \cdots) & \rightarrow \text{DC}(\text{inert}_1 \cdots e_i \text{ inert}_{i+1} \cdots)
\end{align*}
\]
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 \( \mathcal{T} \) 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.

6.8 Eqv? and equivalence

There is one twist, though, since rather than the apply-values primitive given in section 3, \( R^2 RS \) 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 \( (#\%call-with-values thunk) \) reduces to \( (#\%call-with-values (cwv-mark (thunk)) \) ; 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.

---

**Figure 17. Call/cc and dynamic-wind**

\[
\begin{align*}
\text{PC}[v_1], & \quad \Rightarrow \text{PC}[\text{#v} \ 1] \\
\text{PC}[\text{#v} \ 1], & \quad \Rightarrow \text{error: wrong number of values} \\
\text{PC}[\text{#v} \ 1 \ \ldots], & \quad \Rightarrow \text{error: wrong number of values} \\
\text{PC}[\text{#call-with-values vvals vfun}], & \quad \Rightarrow \text{error: wrong number of values} \\
\text{PC}[\text{#call-with-values vvals vfun}], & \quad \Rightarrow \text{error: wrong number of values} \\
\text{PC}[\text{#call-with-values vvals vfun}], & \quad \Rightarrow \text{error: wrong number of values} \\
\text{PC}[\text{#call-with-values vvals vfun}], & \quad \Rightarrow \text{error: wrong number of values} \\
\end{align*}
\]

**Figure 18. Call/cc and dynamic-wind support**

\[
\begin{align*}
\text{PC}[\text{#eqv? v1 v2}], & \quad \Rightarrow 0 \ #t \\
\text{PC}[\text{#eqv? v1 v2}], & \quad \Rightarrow 0 \ #t \\
\text{PC}[\text{#eqv? v1 v2}], & \quad \Rightarrow 0 \ #t \\
\text{PC}[\text{#eqv? v1 v2}], & \quad \Rightarrow 0 \ #t \\
\text{PC}[\text{#eqv? v1 v2}], & \quad \Rightarrow 0 \ #t \\
\text{PC}[\text{#eqv? v1 v2}], & \quad \Rightarrow 0 \ #t \\
\text{PC}[\text{#eqv? v1 v2}], & \quad \Rightarrow 0 \ #t \\
\text{PC}[\text{#eqv? v1 v2}], & \quad \Rightarrow 0 \ #t \\
\end{align*}
\]

**Figure 19. Multiple values and call-with-values**

**Figure 20. Eqv and equivalence**
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 eval 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 (#%null) 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-c-SC 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^5RS 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]. Gashichler, 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^5RS Scheme formal semantics. Van Straaten has written an interpreter based on the R^5RS 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^5RS 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^5RS 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^5RS 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


