JFP **18** (1): 47–86, 2008.
doi:10.1017/S0956796807006478 First published online 1 August 2007 Printed in the United Kingdom

An operational semantics for Scheme[1]

JACOB MATTHEWS and ROBERT BRUCE FINDLER
University of Chicago
(*e-mail:* {jacobm,robby}@cs.uchicago.edu)

Abstract

This paper presents an operational semantics for the core of Scheme. Our specification improves over the denotational semantics from the Revised[5] Report on Scheme specification in four ways. First, it covers a larger part of the language, specifically *eval*, **quote**, *dynamic-wind*, and the top level. Second, it models multiple values in a way that does not require changes to unrelated parts of the language. Third, it provides a faithful model of Scheme's undefined order of evaluation. Finally, we have implemented our specification in PLT Redex, a domain-specific language for writing operational semantics. The implementation 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 a specification of Scheme, this paper contributes three novel modeling techniques for Felleisen Hieb-style rewriting semantics. All three techniques are applicable to a wider range of problems than modeling Scheme, and they combine seamlessly in our model, suggesting that they would scale to complete models of other languages.

1 Introduction

The Revised[5] Report on Scheme (Kelsey *et al.* 1998), hereafter referred to as the Report, 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. For instance, the formal specification does not include the top level, and is missing key procedures such as *dynamic-wind* and *eval* whose inclusion would probably require significant changes to 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 (2003) recent explanation of the difficulties involving dynamic contexts and threads, for instance, demonstrates that the formal model is insufficient for some important questions. Furthermore, denotational semantics have fallen out of favor among programming language researchers in recent years; it is just too difficult to specify nondeterministic language features and establishing theorems about particular semantics, such as type-soundness (Wright & Felleisen 1994), has proven to be much easier in an operational setting. In addition, denotational semantics

[1] All of the systems presented in this paper (and test suites for them) are available as PLT Redex implementations at http://www.cs.uchicago.edu/~robby/r5rs-jfp/

requires much more mathematical sophistication than an operational semantics, making it less appropriate for a standard intended for use by working programmers.

In this paper we give a new treatment of Scheme's formal semantics that models more of the language described in the informal semantics section than the formal semantics section in the Report document does. It is also executable by design and comes with an implementation as a program in PLT Redex, a domain-specific language for context-sensitive rewriting (Matthews *et al.* 2004). PLT Redex provides facilities for modeling nondeterminism and nonconfluence, both of which are necessary for modeling Scheme, and provides a graphical browser for exploring reduction graphs. Modeling Scheme in this manner also allowed us to build a large test suite of terms and their expected normal forms that we run whenever we change any reduction rules; this test suite increases our confidence that our model is a faithful representation of Scheme (see section 19 for more about the test suite).

This paper consists of two parts. Part 1 introduces small models that explain particular features of Scheme, and part 2 combines them. Before part 1 begins, section 2 discusses related work, and section 3 provides a brief overview of the formalism we use. In section 4, the first section in part 1, we show how to use a controlled form of nondeterminism to model Scheme's unspecified application order; in section 5 we show a novel technique for modeling multiple return values; in section 6 we give a model for **quote** and *eval* that exploits a technique for reduction semantics with multiple phases. Section 7 gives a model for Scheme's top level that illustrates a subtle interaction between top-level expressions and continuations and section 8 gives a model for *call/cc* in the presence of *dynamic-wind*. In part 2, we combine all those models along with several other more straightforward features: **if** and **begin**, *cons* and cons-cell mutation, variable-arity procedures, and an object-identity-sensitive notion of *eqv?* equality.

This work extends the first author's master's paper (Matthews 2005) and a paper that appeared at the Scheme and Functional Programming workshop (Matthews & Findler 2005).

2 Related work

Reduction semantics has been used to model large programming languages many times and in many different ways. Felleisen's dissertation (1987), 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 (1988) extends that model into the λ-v-CS calculus in later work. Since then, reduction semantics have been used to model the cores of many languages including SML (Harper & Lillibridge 1993; Wright & Felleisen 1994; Harper & Stone 1996), MultiLisp (Flanagan & Felleisen 1999), Concurrent ML (Reppy 1999), Java (Flatt *et al.* 1999), and Emacs Lisp (Neubauer & Sperber 2001) among others.

There has also been extensive work on the semantics of Scheme. Clinger (1998) presented an operational semantics for a core Scheme for his work on tail recursion. Gasbichler, Knauel, Sperber, and Kelsey (2003) have presented operational and denotational semantics for *dynamic-wind*. Ramsdell (1992) presented a structural

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

operational semantics for Scheme aimed at fixing the unspecified order of argument evaluation problem we discuss in subsection 4. His model is less complete than ours; it matches more closely the language of the denotational semantics from the Report. Also, he considers a program whose results depend on the order of evaluation to be invalid. As we discuss in section 4, that is not the intent of the Report's authors. Van Straaten (2002) has developed a definitional interpreter that is syntactically very similar to the denotational semantics in the Report, although we know of no formal correspondence between them. Our work does not have any formal connection either, but our language is much larger and our semantics, being small-step and executable, allows us to provide programmers with a stepper that rewrites object programs to object programs.

There have also been other efforts to work with large semantics. Oliva *et al.* (1995) proved a VLISP compiler correct, and Lee *et al.* (2006) have also implemented Harper and Stone's semantics using Twelf. The latter 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 this one).

3 Preliminaries

As a rough guide, we define the operational semantics of a language via a relation on program terms, where the relation corresponds to a single step of an abstract machine. The relation is defined using evaluation contexts, namely terms with a distinguished subterms in them, called *holes*, where the next step of evaluation occurs. We say that a term e decomposes into an evaluation context E and another term e' if e is the same as E but with the hole replaced by e'. We write $E[e']$ to indicate the term obtained by replacing the hole in E with e' and we write [ ] to indicate the hole.

For example, assuming that we have defined a grammar containing nonterminals for evaluation contexts (E), expressions (e), variables (x), and values (v), we would write

$$E[((\textbf{lambda}\ (x\ \cdots)\ e)\ v\ \cdots)] \rightarrow E[\{x\ \cdots \mapsto v\ \cdots\}e] \qquad (\#x = \#v)$$

to define the β_v rewriting rule (as a part of the $\rightarrow$ single-step relation). We use the names of the nonterminals (possibly with subscripts) in a rewriting rule to restrict the application of the rule, so it applies only when some element produced by that grammar appears in the corresponding position in the term. If the same nonterminal (with an identical subscript) appears multiple times, the rule applies only when the corresponding terms are structurally identical. Thus, the occurrence of E on both the left-hand and right-hand sides of the rule above means that the context of the application expression does not change when using this rule. The ellipses are a form of Kleene star, meaning that zero or more occurrences of terms matching the pattern proceeding the ellipsis may appear in place of the the ellipsis and the pattern preceding it. Note that, for example, $(x_1, \ldots)$ is not a short-hand for $(x_1, x_2, x_3, x_4, \ldots)$. Instead it means that the meta-variable x_1 is used to match an entire sequence. The meta-variables x_2, x_3, and x_4 are all independent of this.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

We use the notation $\{x \cdots \mapsto v \cdots\}e$ for capture-avoiding substitution; in this case it means that each x is replaced with the corresponding v in e. Finally, we write side conditions in parenthesis beside a rule; the side condition in the above rule indicates that the number of xs must be the same as the number of vs. Sometimes, we use equality in the side conditions; when we do it merely means simple term equalitys *i.e.*, the two terms must be syntactically identical.

Making the evaluation context E explicit in the rule allows us to define relations that manipulate their context. As a simple example, we can add another rule that signals an error when a procedure is applied to the wrong number of arguments by discarding the evaluation context on the right-hand side of a rule:

$$E\,[((\textbf{lambda}\ (x \cdots)\ e)\ v \cdots)] \rightarrow \textbf{error:}\ \text{wrong argument count} \qquad (\#x \neq \#v)$$

Later we show how to take advantage of the explicit evaluation context in more sophisticated ways.

To learn more about this style of semantics, we refer readers to Felleisen and Flatt's monograph (2006).

PART ONE
Small Reduction Systems

This part introduces smaller reduction systems that explain the details of particular features of the Report and how we model them. In each case, the modeling techniques are the same as in the full semantics, but we hope that seeing them in isolation makes them easy to understand later. Since the models are just intended to be illustrative, we include only the a minimal amount of detail in each. For example, each of the systems contains stuck states that would correspond to errors in a more fleshed out semantics.

Section 4 illustrates how we model the Report's underspecification of the order of evaluation for application expressions. Section 5 shows how we model multiple values. Section 6 demonstrates quoted constants and *eval* and the interplay between the two of them. Section 7 contains a model of the top level and explains a subtle interaction with *call/cc*, and finally our model of *dynamic-wind* in section 8 concludes part 1.

4 Unspecified evaluation order for applications

In evaluating a procedure call, the Report document deliberately leaves unspecified the order in which arguments are evaluated, but in 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 7.2), the authors explain how they model this ambiguity:

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

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$$
\begin{array}{lll}
p & ::= & (\textbf{store}\ (sf\ \cdots)\ e) \\
sf & ::= & (x\ v) \\
e & ::= & (e\ e\ \cdots) \mid (\textbf{set!}\ x\ e) \mid (\textbf{begin}\ e\ e\ \cdots) \mid (\textbf{if}\ e\ e\ e) \mid x \mid v \\
v & ::= & (\textbf{lambda}\ (x\ \cdots)\ e) \mid n \mid \#t \mid \#f \mid - \mid \mathit{unspecified} \\
P & ::= & (\textbf{store}\ (sf\ \cdots)\ E) \\
E & ::= & (v\ \cdots\ E\ e\ \cdots) \mid (\textbf{set!}\ x\ E) \mid (\textbf{begin}\ E\ e\ e\ \cdots) \mid (\textbf{if}\ E\ e\ e) \mid [\,] \\
x & ::= & \text{[identifiers and store locations for mutable bindings]} \\
n & ::= & \text{[numbers]}
\end{array}
$$

$$
\begin{array}{lll}
(\textbf{store}\ ((x_1\ v_1)\ \cdots)\ E[((\textbf{lambda}\ (x_2\ \cdots)\ e)\ v_2\ \cdots)]) \rightarrow & & \text{[MApp]} \\
(\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x_2'\ v_2)\ \cdots)\ E[\{\ x_2\ \cdots \mapsto x_2'\ \cdots\ \}e]) & (\#x_2 = \#v_2, \text{each } x_2' \text{ fresh}) & \\
(\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x\ v)\ (x_2\ v_2)\ \cdots)\ E[(\textbf{set!}\ x\ v')]) \rightarrow & & \text{[MSet]} \\
(\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x\ v')\ (x_2\ v_2)\cdots)\ E[\mathit{unspecified}]) & & \\
(\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x\ v)\ (x_2\ v_2)\ \cdots)\ E[x]) \rightarrow & & \text{[MLookup]} \\
(\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x\ v)\ (x_2\ v_2)\ \cdots)\ E[v]) & & \\
P[(\textbf{begin}\ v\ e_1\ e_2\ \cdots)] \rightarrow P[(\textbf{begin}\ e_1\ e_2\ \cdots)] & & \text{[MSeq]} \\
P[(\textbf{begin}\ e)] \rightarrow P[e] & & \text{[MTrivSeq]} \\
P[(\textbf{if}\ v_1\ e_1\ e_2)] \rightarrow P[e_1] & (v_1 \neq \#f) & \text{[MIfT]} \\
P[(\textbf{if}\ \#f\ e_1\ e_2)] \rightarrow P[e_2] & & \text{[MIfF]} \\
P[(-\ \ulcorner n \urcorner)] \rightarrow P[\ulcorner -n \urcorner] & & \text{[MNeg]}
\end{array}
$$

Fig. 1. Core Scheme with mutation.

In this section we present an operational technique that captures the intended semantics faithfully. We begin by considering a core Scheme with arbitrary arity procedures, **set!**, sequencing, conditionals, booleans, and numbers, but with a fixed left-to-right order of evaluation for applications, as shown in Figure 1. It is a variation of Felleisen and Hieb's Λ_S (1992). 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!**, **if**, and **begin** expressions, and a built-in negation operator (in order to facilitate a coming example).

The [MApp] rule gives the rule for application of a procedure to fully evaluated arguments: make one fresh identifier x_2' for each formal parameter x_2, introduce a new binding in the store for each x_2' associating it with the corresponding argument in the application, and then rewrite the application as the procedure's body with each occurence of an x_2 replaced by the corresponding x_2'. The [MSet] rule replaces the value associated with the given identifier in the store with the given replacement. The Report does not specify the result of a **set!** operation, so we follow the lead of many Scheme implementations and rewrite to a special value called *unspecified*. The [MLookup] rule corresponds to the evaluation of an variable, replacing it with its associated value in the store.

The rules for **begin** expressions exploit the definition of the evaluation contexts (E) for **begin** expressions, which allow evaluation only in the first subexpression of a **begin** and only when there are at least two subexpressions. [MSeq] drops the

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

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 rules [MIfT] and [MIfF] handle **if** expressions and the last rule, [MNeg], simply negates its argument (the notation $\ulcorner n \urcorner$ indicates the syntactic representation of the mathematical number n).

The order of evaluation is determined by the grammar for evaluation contexts. The first production of the grammar specifies that evaluation of a subexpression of an application takes place only when all of the subexpressions to its left are values. If we replace that first production with this one:

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

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

Either of these choices results in a system with unique decomposition. That is, each noncanonical term can be split into exactly one evaluation context and reducible subexpression. Accordingly, there is at most one way to reduce any expression.

To model a language with unspecified order of operations, like that in the Report, we can use a reduction system with nonunique decomposition to model the choice of which argument to evaluate. We might be tempted to use this definition of evaluation contexts:

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

Since this definition allows the hole to appear in any subexpression of an application, this simple program

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

which negates 1, negates 2, and then applies a trivial procedure to the results, can be split into an evaluation context with either $(-\ 1)$ or $(-\ 2)$ as the reducible expression.

Although, this might appear to be a faithful model of the Report, it is flawed. Consider the application of two **set!** expressions in a store binding x to 1:

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

This program should always reduce to the application of the unspecified value to itself with x set to 1 in the store because, according to the Report, 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 different arguments of the same application to alternate steps of computation. This, in turn, 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 at the top. The outermost 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.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

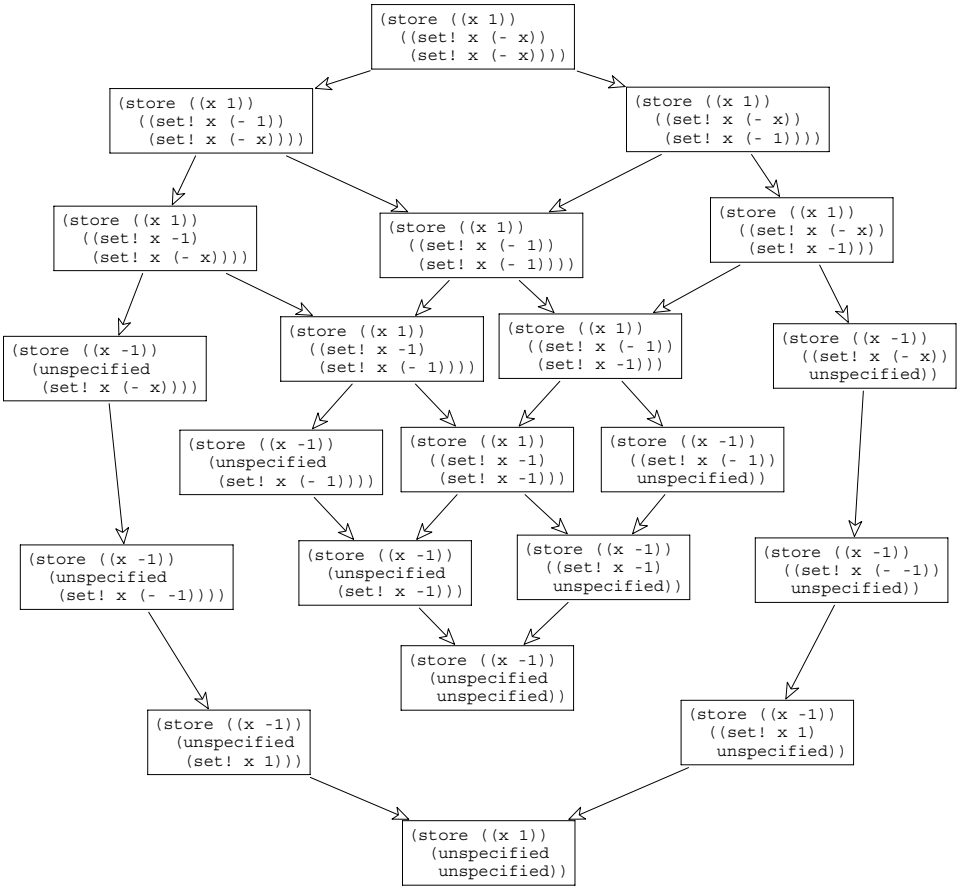

Fig. 2. Interleavings possible with an erroneous unspecified-application-order model.

With that in mind, we can design a more sophisticated strategy that captures unspecified evaluation order but allows only sequential orderings. The basic idea is to use nondeterministic choice to pick a subexpression to reduce only when we have not already committed to reducing some other subexpression. To achieve that effect, we introduce the notion of a marked expression, denoted with the $\diamond$ superscript. (These marks are not an extension to the general term-rewriting framework—the mark is simply a term constructor, albeit typeset specially.) Marks identify chosen expressions: only marked expressions may be reduced, and only one reducible marked expression may appear in any application at one time.

Figure 3 shows the necessary revisions to core Scheme to support the Report's style procedure applications. The E nonterminal replaces the one from Figure 3 and we add application expressions that contain marked subexpressions to e. The two rules [Mark] and [Unmark] are added to the existing rules in Figure 3. The [Mark] reduction rule marks an arbitrary unmarked expression in an application on the condition that the expression under the mark is not already a value, and the [Unmark] rule removes the mark when the marked expression is fully reduced. The

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$$
\begin{array}{lcl}
E & ::= & (e \cdots E^{\diamond} \; e \cdots) \mid (\textbf{set!}\; x\; E) \mid (\textbf{begin}\; E\; e\; e \cdots) \mid (\textbf{if}\; E\; e\; e) \mid [\,] \\
e & ::= & \ldots \mid (e \cdots e^{\diamond} \; e \cdots)
\end{array}
$$

$$
\begin{array}{lcll}
P[(e_1 \cdots e_2\; e_3 \cdots)]) & \rightarrow & P[(e_1 \cdots e_2^{\diamond}\; e_3 \cdots)]) & \text{[Mark]} \\
 & & (e_2 \notin v) & \\
P[(e_1 \cdots v^{\diamond}\; e_2 \cdots)]) & \rightarrow & P[(e_1 \cdots v\; e_2 \cdots)]) & \text{[Unmark]}
\end{array}
$$

Fig. 3. Unspecified application order semantics, as an extension of Figure 1.

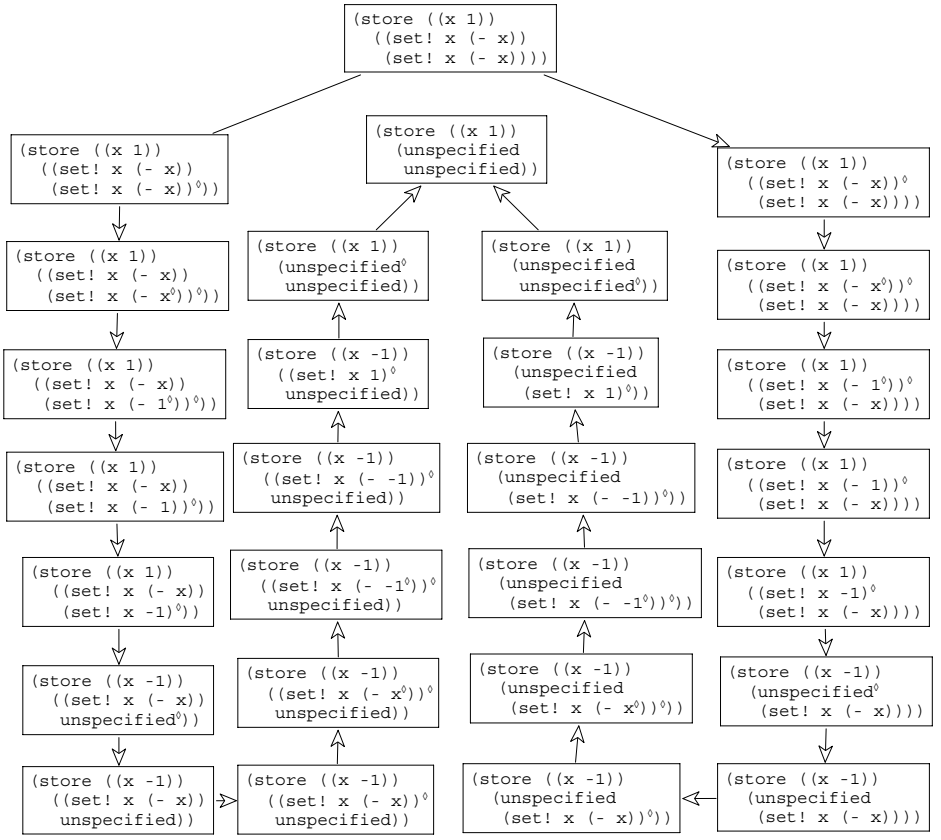

Fig. 4. Evaluation in the unspecified-application-order model.

new evaluation contexts E ensure that evaluation inside an application expression can occur only inside a marked expression.

Figure 4 shows how our new system evaluates the term from Figure 2. The initial term appears at the top. That term is an application, so the first reduction marks either the first subexpression or the second. If the first subexpression is marked, evaluation continues down to the right of the figure, and back up to the middle. If the second is marked, evaluation proceeds down to the left and back up the middle.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

Eventually, all of the terms join back together and the final result in the store is *1*, as shown in the center just under the initial term.

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 return either *1* or *2*, depending on the order of evaluation, and this result is desirable. The model's nonconfluence precisely reflects the underspecification of the Report's informal language.

This technique of using evaluation contexts to partially control where evaluation occurs 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 and only in certain ways. Threads and futures are good examples of this kind of language feature.

5 Multiple return values

The Report specifies a facility for expressions to evaluate simultaneously to multiple or no values rather than just a single value. The procedure *values* introduces multiple values and *call-with-values* eliminates multiple values. Unlike tuples in SML and Haskell, however, a collection of multiple values is not itself a value. 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 eliminate 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:

```
(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, the Report's formal semantics uses functions from an arbitrary number of values to a final answer as continuations. In section 6.4, the Report says that "[e]xcept for continuations created by the call-with-values procedure, all continuations take exactly one value." Perhaps surprisingly, though, the formal semantics ensures that by checking the opposite property: in every context

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$$
\begin{array}{lcl}
p & ::= & (\textbf{store}\ (sf\ \cdots)\ e) \\
sf & ::= & (x\ v) \\
e & ::= & (e\ e\ \cdots) \mid (\textbf{set!}\ x\ e) \mid (\textbf{begin}\ e\ e\ \cdots) \mid (\textbf{if}\ e\ e\ e) \mid x \mid v \\
v & ::= & (\textbf{lambda}\ (x\ \cdots)\ e) \mid n \mid \#t \mid \#f \mid - \mid \mathit{unspecified} \mid \mathit{call\text{-}with\text{-}values} \mid \mathit{values} \\
P & ::= & (\textbf{store}\ (sf\ \cdots)\ E^\star) \\
E & ::= & (v\ \cdots\ E^\circ\ e\ \cdots) \mid (\textbf{set!}\ x\ E^\circ) \mid (\textbf{begin}\ E^\star\ e\ e\ \cdots) \mid (\textbf{if}\ E^\circ\ e\ e) \\
 & \mid & (\mathit{call\text{-}with\text{-}values}\ (\textbf{lambda}\ ()\ E^\star)\ v) \mid [\,] \\
E^\circ & ::= & E \mid [\,]_\circ \\
E^\star & ::= & E \mid [\,]_\star \\
x & ::= & \text{[identifiers and store locations for mutable bindings]} \\
n & ::= & \text{[numbers]}
\end{array}
$$

$$P[v]_\star \rightarrow P[(\mathit{values}\ v)] \qquad \text{[VPromote]}$$

$$P[(\mathit{values}\ v)]_\circ \rightarrow P[v] \qquad \text{[VDemote]}$$

$$P[(\mathit{values}\ v\ \cdots)]_\circ \rightarrow \textbf{error:}\ \text{context received wrong \# of values} \quad (\#v \neq 1) \qquad \text{[VDemoteErr]}$$

$$P[(\mathit{call\text{-}with\text{-}values}\ (\textbf{lambda}\ ()\ (\mathit{values}\ v_2\ \cdots))\ v_1)] \rightarrow P[(v_1\ v_2\ \cdots)] \qquad \text{[VCwv]}$$

$$P[(\mathit{call\text{-}with\text{-}values}\ v_1\ v_2)] \rightarrow P[(\mathit{call\text{-}with\text{-}values}\ (\textbf{lambda}\ ()\ (v_1))\ v_2)] \quad (v_1 \neq (\textbf{lambda}\ ()\ e)) \qquad \text{[VCwvApp]}$$

$$(\textbf{store}\ ((x_1\ v_1)\ \cdots)\ E^\star[((\textbf{lambda}\ (x_2\ \cdots)\ e)\ v_2\ \cdots)]) \rightarrow (\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x_2'\ v_2)\ \cdots)\ E^\star[\{\ x_2\ \cdots \mapsto x_2'\ \cdots\ \}e]) \quad (\#x_2 = \#v_2, \text{each } x_2' \text{ fresh}) \qquad \text{[VApp]}$$

$$(\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x\ v)\ (x_2\ v_2)\ \cdots)\ E^\star[(\textbf{set!}\ x\ v')]) \rightarrow (\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x\ v')\ (x_2\ v_2)\cdots)\ E^\star[\mathit{unspecified}]) \qquad \text{[VSet]}$$

$$(\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x\ v)\ (x_2\ v_2)\ \cdots)\ E^\star[x]) \rightarrow (\textbf{store}\ ((x_1\ v_1)\ \cdots\ (x\ v)\ (x_2\ v_2)\ \cdots)\ E^\star[v]) \qquad \text{[VLookup]}$$

$$P[(\textbf{begin}\ (\mathit{values}\ v\ \cdots)\ e_1\ e_2\ \cdots)] \rightarrow P[(\textbf{begin}\ e_1\ e_2\ \cdots)] \qquad \text{[VSeq]}$$

$$P[(\textbf{begin}\ e)] \rightarrow P[e] \qquad \text{[VTrivSeq]}$$

$$P[(\textbf{if}\ v_1\ e_1\ e_2)] \rightarrow P[e_1] \quad (v_1 \neq \#f) \qquad \text{[VIfT]}$$

$$P[(\textbf{if}\ \#f\ e_1\ e_2)] \rightarrow P[e_2] \qquad \text{[VIfF]}$$

$$P[(-\ \ulcorner n \urcorner)] \rightarrow P[\ulcorner -n \urcorner] \qquad \text{[VNeg]}$$

Fig. 5. Core Scheme with multiple values.

that accepts only a single value, it uses a helper function, *single*, to ensure that only a single value appears.

Our semantic model captures the difference between contexts that accept multiple values and contexts that reject multiple values directly. The basic strategy is to add a rule that demotes (*values* v) to v and another rule that promotes v to (*values* v), but to only allow demotion in a context expecting a single value and only allow promotion in a context expecting multiple values. We obtain this behavior with a

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

small extension to the Felleisen–Hieb framework. We label holes to distinguish them from each other, written as subscripts (for instance $[\,]_\star$ or $[\,]_\circ$). We also extend the context-matching operation so it may demand a hole of a particular name, also written with a subscript. For example $E[e]_\star$ would be a legal decomposition only if the hole in E is $[\,]_\star$; neither $[\,]$ nor $[\,]_\circ$ would be allowed. The extension allows us to give different names to the holes in which multiple values are expected and those in which single values are expected, and structure the grammar of contexts accordingly.

Figure 5 shows the extension of core Scheme to support multiple values. $[\,]_\circ$ indicates a hole in which any expression should reduce to an element of v, $[\,]_\star$ indicates a hole in which any expression should reduce to (*values* v $\cdots$), and the subscript-less $[\,]$ indicates a hole in which either result is acceptable. There are also three context nonterminals. The final result of a program in the context E° produces an element of v, $E^\star$ produces (*values* v $\cdots$), and E might produce either.

The definition of E follows the informal specification of the Report, using $E^\star$ when multiple values are expected and E° when single values are expected. Since the thunks passed to *call-with-values* are allowed to produce multiple values, we use $E^\star$ there. Similarly, since the final result of a program may be multiple values, we use $E^\star$ in the definition of P, and everywhere else we use E°.

The first five rules are new, beyond the rules for core Scheme. [VPromote] promotes a single value v to (*values* v). Because of the subscript $\star$ 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. The [VCwv] rule reduces a call to *call-with-values* when the body of the thunk passed to *call-with-values* has been fully evaluated. The [VCwvApp] adds a thunk wrapper around the first argument to *call-with-values* when it is not already a thunk. For example, (*call-with-values values values*) reduces to (*values*) by first using [VCwvApp].

All reductions take place in $E^\star$, allowing the final result of any program to be multiple values. If we wanted to allow only a single values as the final result of a program, we could replace $E^\star$ with E° in all of the rules and in P.

To get a sense of how evaluation proceeds, consider this reduction sequence (shown here without the store):

	(– (*call-with-values* (**lambda** () *1*) (**lambda** (*x*) (*values x*))))	
→	(– (*call-with-values* (**lambda** () (*values 1*)) (**lambda** (*x*) (*values x*))))	[VPromote]
→	(– ((**lambda** (*x*) (*values x*)) *1*))	[VCwv]
→^{2}	(– (*values 1*))	[VApp],[VLookup]
→	(– *1*)	[VDemote]
→	*–1*	[VNeg]
→	(*values –1*)	[VPromote]

The first term helps illustrate how the labels on the evaluation contexts ensure that

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

only appropriate promotion and demotion occur. Consider this evaluation context from the first term:

$$(- \ (\textit{call-with-values} \ (\textbf{lambda} \ () \ [\,]_\star) \\ (\textbf{lambda} \ (x) \ (\textit{values} \ x))))$$

Since it has $[\,]_\star$ in it, the [VPromote] rule applies to turn *1* into (*values 1*), producing the second term. At this point, the [VDemote] rule does not apply to that same context, because it requires the hole in the context to be $[\,]_\circ$. If it were to apply, the term would have to decompose into this evaluation context:

$$(- \ (\textit{call-with-values} \ (\textbf{lambda} \ () \ [\,]_\circ) \\ (\textbf{lambda} \ (x) \ (\textit{values} \ x))))$$

but that evaluation context is not generated by E.

Instead, [VCwv] applies, passing the results of the first argument of *call-with-values* to the second argument to *call-with-values*. After that, the application occurs, using rules [VApp] and [VLookup]. Then the term (*values 1*) is used as an argument to a procedure, so [VDemote] converts it to the single value *1*. Next, [VNeg] negates *1*, producing *−1*. Finally, [VPromote] applies (since the outermost context for each rule is $E^\star$) and the final result is (*values −1*).

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

$$\begin{aligned} & (g \ (f \ 3)) \\ \rightarrow^* \ & (g \ (\textit{values} \ 6 \ 9)) \\ \rightarrow \ & \textbf{error:} \text{ context received wrong \# of values} \end{aligned}$$

In general, this strategy 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 (2004) static analysis of embedded languages. It also can be used to model interoperability (Matthews & Findler 2007).

6 Quote and eval

Scheme inherits the meta-programming facilities *eval* and **quote** from Lisp (Sussman & Guy Lewis Steele 1975). The **quote** operator turns program code into a datum and the *eval* procedure turns that datum back into code. 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 element list whose first and third elements are symbols and whose second element is a list of one element:

$$(\textit{list} \ (\textbf{quote} \ \textit{lambda}) \ (\textit{list} \ (\textbf{quote} \ x)) \ (\textbf{quote} \ x))$$

Because of the presence of both **quote** and *eval*, the process of turning quoted program text into a datum and then back into program text is interleaved with

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

ordinary evaluation. For example, this program

```
(define (f x y)
  (eval (list (quote /) x y)))

(f (quote (+ 2 3 4))
   (quote (eval (quote (+ 5 6)))))
```

first turns the arguments to f into data, and then calls f, which constructs a quotient expression and passes it to *eval*. To continue evaluation, *eval* turns the datum back into program text and we are left with this program:

```
(/ (+ 2 3 4)
   (eval (quote (+ 5 6))))
```

At this point, however, the sum of *5* and *6* must be turned into a datum before proceeding, since it is quoted. Once that happens, the remaining call to *eval* turns it back into program text, and the program evaluates to *9/11*.

The Report 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 pointer each time it is called.

Our core Scheme calculus for modeling *eval* and **quote** is shown in Figure 6, extending Figure 1. It adds quoted expressions, pointers, null, and the primitive functions: *eval*, *cons*, *car*, *cdr*, and *eqv?*. The *ptr* nonterminal generates pointers, which are used indicies into the store and are compared for equality.

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, i.e., terms generated by the *s* nonterminal,[2] 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 apply to a program only after it has been completely compiled.

The first group of evaluation rules (from [ECons] to [EEqv2]) corresponds to the language's runtime semantics, and shows how the list primitives behave. [ECons] models the application of *cons* to arguments by allocating a new pair in the store; *car* and *cdr* select the first and second values in a pair by the rules [ECar] and [ECdr]. The [EEqv1] and [EEqv2] rules give *eqv?*'s semantics; it compares pointers for literal syntactic equality (and, for this language, operates only on pairs). Since each of these reduction takes place in an *evaluation* (rather than *compilation*) context, they apply only to programs that are completely compiled.

[2] We write dotted pairs with **dot** rather than a period to avoid metacircular confusion in our PLT Redex implementation.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$$
\begin{array}{lcl}
v & ::= & \ldots \mid (\textbf{quote}\ sy) \mid ptr \mid null \mid prim \\
sf & ::= & \ldots \mid (ptr\ (cons\ v\ v)) \\
prim & ::= & eval \mid cons \mid car \mid cdr \mid eqv? \\
ptr & ::= & \text{pointers} \\
sy & ::= & \text{[names of symbols]} \\
 & & \text{(identifiers except } \textbf{dot}\text{)} \\
x & ::= & \text{[names of variables]} \\
 & & \text{(members of } sy \text{ except keywords)}
\end{array}
\qquad
\begin{array}{lcl}
s & ::= & (s \cdots) \mid sqv \mid sy \\
 & \mid & (s \cdots \textbf{dot}\ sy) \mid (s \cdots \textbf{dot}\ n) \\
sqv & ::= & n \mid \#t \mid \#f \\
SP & ::= & (\textbf{store}\ (sf \cdots)\ S) \\
S & ::= & [\,] \mid (e \cdots S\ s \cdots) \\
 & \mid & (\textbf{lambda}\ (x \cdots)\ S) \\
 & \mid & (\textbf{if}\ S\ s\ s) \mid (\textbf{if}\ e\ S\ s) \mid (\textbf{if}\ e\ e\ S) \\
 & \mid & (\textbf{ccons}\ v\ S) \mid (\textbf{ccons}\ S\ s)
\end{array}
$$

$$(\textbf{store}\ (sf_1 \cdots)\ E[(cons\ v_1\ v_2)]) \rightarrow \quad \text{[ECons]}$$
$$(\textbf{store}\ (sf_1 \cdots (ptr\ (cons\ v_1\ v_2)))\ E[ptr]) \qquad (ptr \text{ fresh})$$

$$(\textbf{store}\ (sf_1 \cdots (ptr\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[(car\ ptr)]) \rightarrow \quad \text{[ECar]}$$
$$(\textbf{store}\ (sf_1 \cdots (ptr\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[v_a])$$

$$(\textbf{store}\ (sf_1 \cdots (ptr\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[(cdr\ ptr)]) \rightarrow \quad \text{[ECdr]}$$
$$(\textbf{store}\ (sf_1 \cdots (ptr\ (cons\ v_a\ v_d))\ sf_2 \cdots)\ E[v_d])$$

$$P[(eqv?\ ptr_1\ ptr_1)] \rightarrow P[\#t] \quad \text{[EEqv1]}$$

$$P[(eqv?\ ptr_1\ ptr_2)] \rightarrow P[\#f] \quad (ptr_1 \neq ptr_2) \quad \text{[EEqv2]}$$

$$SP[(\textbf{quote}\ (s_1\ s_2 \cdots))] \rightarrow SP[(\textbf{ccons}\ (\textbf{quote}\ s_1)\ (\textbf{quote}\ (s_2 \cdots))] \quad \text{[EQSeq]}$$

$$SP[(\textbf{quote}\ ())] \rightarrow SP[null] \quad \text{[EQNull]}$$

$$SP[(\textbf{quote}\ (s_1\ s_2\ s_3 \cdots \textbf{dot}\ s_4))] \rightarrow SP[(\textbf{ccons}\ (\textbf{quote}\ s_1)\ (\textbf{quote}\ (s_2\ s_3 \cdots \textbf{dot}\ s_4))] \quad \text{[EQSeqD]}$$

$$SP[(\textbf{quote}\ (s_1\ \textbf{dot}\ s_2))] \rightarrow SP[(\textbf{ccons}\ (\textbf{quote}\ s_1)\ (\textbf{quote}\ s_2))] \quad \text{[EQDot]}$$

$$SP[(\textbf{quote}\ sqv)] \rightarrow SP[sqv] \quad \text{[EQNum]}$$

$$(\textbf{store}\ (sf \cdots)\ S[(\textbf{ccons}\ v_1\ v_2)]) \rightarrow (\textbf{store}\ (sf \cdots (ptr\ (cons\ v_1\ v_2)))\ S[ptr]) \quad (ptr \text{ fresh}) \quad \text{[EQPair]}$$

$$(\textbf{store}\ (sf \cdots)\ E[(eval\ v)] \rightarrow (\textbf{store}\ (sf \cdots)\ E[\mathscr{R}\ [\![\ (sf \cdots), v\ ]\!]]) \quad \text{[EEval]}$$

$$
\begin{array}{lcll}
\mathscr{R} : (ptr \mapsto (cons\ v\ v)) \times v \rightarrow s & & \mathscr{C} : s \times s \rightarrow s & \\
\mathscr{R}\ [\![\ S, null\ ]\!] & = & () & \\
\mathscr{R}\ [\![\ S, (\textbf{quote}\ sy)\ ]\!] & = & sy & \\
\mathscr{R}\ [\![\ S, ptr\ ]\!] & = & \mathscr{C}\ [\![\ \mathscr{R}\ [\![\ v_a\ ]\!], \mathscr{R}\ [\![\ v_d\ ]\!]\ ]\!] & \\
\mathscr{R}\ [\![\ S, v\ ]\!] & = & v \quad \text{(otherwise)} &
\end{array}
\qquad
\begin{array}{lcl}
\mathscr{C} : s \times s \rightarrow s & & \\
\mathscr{C}\ [\![\ s1, (s2 \cdots)\ ]\!] & = & (s1\ s2 \cdots) \\
\mathscr{C}\ [\![\ s1, s2\ ]\!] & = & (s1\ \textbf{dot}\ s2) \\
\multicolumn{3}{l}{\text{(where } S \text{ binds } ptr \text{ to } (cons\ v_a\ v_d))}
\end{array}
$$

Fig. 6. Eval and quote semantics, as an extension of Figure 1.

The second group of rules (from [EQSeq] to [EQPair]) apply at compile-time and show how to compile a program by rewriting quoted constants into locations in the store. If these 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*.

Instead, the second group of rewriting rules eliminate **quote** before any other evaluation happens, turning s-expressions into Scheme programs. Although we have presented them second, these rules actually come first in reduction sequences, making reduction sequences follow a two-phase pattern where the [EQ...] rules apply in the

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

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 step takes place once all of the s-expressions to the left have become Scheme programs. In E, a rewrite step takes place 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 may proceed.

To model *eval*, we use a technique similar to Muller's *reify* (1992). The $\mathscr{R}$ metafunction accepts a value and turns it back into a program. The $\mathscr{C}$ function (used by $\mathscr{R}$) is the syntactic analogue of *cons*; the first case applies whenever the second argument has parenthesis (i.e., both proper and improper lists), otherwise the second case applies. 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 data into the store before evaluation continues.[3]

The *eval* we present here and in part 2, we should point out, is not as fully-featured as the *eval* of the Report's informal description because it does not accept an environment argument. It does, however, behave most like the *eval* from the report where the second argument is the result of (*interaction-environment*) where that environment contains the parts of the language that the semantics models.

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 macro systems 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.

7 Top-level program structure and call/cc

Section 5 of the Report specifies that the top level of a Scheme program is a mixture of definitions and expressions, and that "[a]t the top level of a program (**begin** <*form1*> $\cdots$) is equivalent to the sequence of expressions, definitions, and syntax

[3] 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) null)))))
 (lambda () '(x)))
```

We can adapt $\mathscr{R}$ to match these implementations via special handling of quoted forms during reification:

$$\mathscr{R}[\![\, S, p_1 \,]\!] = v \qquad \text{if } S \text{ maps } p_1 \text{ to } (\mathit{cons}\ (\textbf{quote}\ \mathit{quote})\ p_2) \text{ and maps } p_2 \text{ to } (\mathit{cons}\ v\ \mathit{null}).$$

which would cause our semantics to produce #*t* for the above example. This technique does not scale to a full Scheme that includes macros, however.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

definitions that form the body of the **begin**." Although that section of the report does not discuss *call/cc*,[4] the equivalence has subtle implications for the semantics of *call/cc* at the top level.

Nearly all of the implementations we tried treat continuations as delimited by a top-level expression.[5] That is, the continuation of one top-level expression does not contain the evaluation of any subsequent top-level expressions. In these implementations, however, the continuation of one of the expressions in a top-level **begin** contains all of the following expressions in the **begin**, violating the Report's mandate that removing top-level **begin**s does not change the program's behavior. For example, this program

```
(define k #f)
(define x 1)
(begin (call/cc (lambda (k2) (set! k k2) 1))
       (set! x (+ x 1)))
(k 1)
```

finishes with x bound to *3* when the **begin** is present, but with x bound to *2* when it is not present.

The source of the inconsistency between continations inside and outside of a **begin** is probably caused by the implementation of the loop that iterates over and evaluates the subexpressions of a **begin**. It appears that the loop variable holding the current subexpression to evaluate is a function parameter, and so its value is held in the continuation and returning to an earlier continuation returns the loop variable to its old value. In contrast, expressions at the top level are probably being read imperatively from a port, so continuations do not return the state of the port to earlier states. To make the continuations consistently delimited, implementations could use an eval function like the one given in Figure 7. It defines *b-eval* in terms of *eval*, the implementation's original evaluator. The essence of the fix is in the body of *evals*. It does not recur with the *cdr* of *exprs* to continue evaluating the body of the **begin**. Instead, it uses a **set!** so that a continuation that jumps back to an earlier iteration of the loop does not see the earlier value of *exprs*. The variable *answers* and the call to *call-with-values* are only there to cope with multiple values that may

[4] The authors' intent seems to be to ensure that the scope of a top-level definition that occurs inside a **begin** expression does not change when the surrounding **begin** is removed.

[5] We tried Bigloo version 2.8b (Serrano 2006), Chicken version 2, build 41 (Winkelmann 2006), Gambit version 4.0 beta 17 (Feeley 2006), Guile version 1.6.8 (Project GNU 2005), MIT Scheme release 7.7.9 (GNU 2006), MzScheme version 352 (Flatt 2006), Petit Chez Scheme version 7.0a (Dybvig 2005), Petit Larceny version 0.92 (Clinger & Hansen 1994), Scheme 48 version 1.3 (Kelsey *et al.* 2005), and SISC version 1.51.1 (Miller & Radestock 2006). With the exception of MIT Scheme, they all behave as explained in the text above. MIT Scheme suffers from a related, but not identical problem. In particular, this program

```
(begin (define k (call-with-current-continuation (lambda (x) x)))
       (define y 1))
(set! y 2)
(k (lambda (x) x))
```

finishes with y bound to *1* in MIT Scheme. Without the **begin**, the program finishes with y bound to *2*.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

```
;; b-eval : expr[fully-expanded] → any
(define (b-eval expr)
  (if (and (pair? expr)
           (eq? (car expr)
                'begin))
      (evals (cdr expr))
      (eval expr)))
```

```
;; evals : (listof expr[fully-expanded]) → any
(define (evals exprs)
  (let ((answers 'dummy))
    (let loop ()
      (if (pair? exprs)
          (let ((expr (car exprs)))
            (set! exprs (cdr exprs))
            (call-with-values
              (lambda () (b-eval expr))
              (lambda args (set! answers args)))
            (loop))
          (apply values answers)))))
```

Fig. 7. Top-level-begin sensitive evaluator.

be produced by the expressions being evaluated. If we were not interested in the result from *eval*, but only its effects, we could have used this simpler *evals* function:

```
(define (evals exprs)
  (let loop ()
    (if (pair? exprs)
        (let ((expr (car exprs)))
          (set! exprs (cdr exprs))
          (b-eval expr)
          (loop)))))
```

Aside from the constraint on top-level **begin** expressions, the Report allows many different semantics for top level. Over time, the implementors of Scheme systems have essentially converged on a consensus semantics. Ironically, this consensus disagrees with the Report on the one element for which it had fixed the interpretation. In this paper, we base our model of the top level on the implementor's consensus, but adjust it so that our model also satisfies the Report.

Figure 8 contains our semantics, as an extension of figure 1. Programs consist of a store and a series of definitions and expressions, where **begin**$^{\mathbf{D}}$ marks a top-level **begin**, distinguishing it from an internal **begin** in order that they might behave differently (note that top-level **begin**$^{\mathbf{D}}$ expressions can be nested). While the semantics needs two different forms of **begin**, the surface language that an implementation provides does not need to. Instead, it can compile **begin**s that appear at the top level into **begin**$^{\mathbf{D}}$ expressions, before evaluation.

The reduction rules from Figure 1 also apply to this system, but where the first three rules use the evaluation context E, they must use D here. The rules [TIDef] and [TIReDef] handle definitions, either allocating a new place in the store for undefined variables, or updating the store for redefinitions. The [TIToss] rule discards a completed expression, unless it is the last one. The [TIBegin] rule just erases **begin**$^{\mathbf{D}}$ expressions, before any evaluation of the **begin**$^{\mathbf{D}}$'s arguments occurs.

Finally, the last two rules, [TICallcc] and [TIThrow], handle continuations. A *call/cc* expression packages up the context into a **throw** expression in the body of a **lambda** expression and passes that into *call/cc*'s argument. When that function is

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$$
\begin{array}{llll}
p & ::= & (\textbf{store}\ ((x\ v)\cdots)\ d\cdots) & P ::= (\textbf{store}\ ((x\ v)\cdots)\ D\ d\cdots) \\
d & ::= & e \mid (\textbf{define}\ x\ e) \mid (\textbf{begin}^{\textbf{D}}\ d\cdots) & D ::= (\textbf{define}\ x\ E) \mid E \\
e & ::= & \ldots \mid (\textbf{throw}\ d) & E ::= \text{(as in figure 1)} \\
v & ::= & \ldots \mid \mathit{call/cc} &
\end{array}
$$

$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ (\textbf{define}\ x_1\ v_1)\ d_1\cdots) \rightarrow \quad \text{[TIDef]}$$
$$(\textbf{store}\ ((x_s\ v_s)\cdots(x_1\ v_1))\ \mathit{unspecified}\ d_1\cdots) \qquad (x_1 \notin \{\, x_s\cdots\,\})$$

$$(\textbf{store}\ ((x_b\ v_b)\cdots(x_1\ v_1)\ (x_a\ v_a)\cdots)\ (\textbf{define}\ x_1\ v_2)\ d\cdots) \rightarrow \quad \text{[TIReDef]}$$
$$(\textbf{store}\ ((x_b\ v_b)\cdots(x_1\ v_2)\ (x_a\ v_a)\cdots)\ \mathit{unspecified}\ d\cdots)$$

$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ v_1\ d_1\ d_2\cdots) \rightarrow \quad \text{[TIToss]}$$
$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ d_1\ d_2\cdots)$$

$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ (\textbf{begin}^{\textbf{D}}\ d_1\cdots)\ d_2\cdots) \rightarrow \quad \text{[TIBegin]}$$
$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ d_1\cdots d_2\cdots)$$

$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ D[(\mathit{call/cc}\ v_1)]\ d\cdots) \rightarrow \quad \text{[TICallcc]}$$
$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ D[(v_1\ (\textbf{lambda}\ (x_2)\ (\textbf{throw}\ D[x_2])))]\ d\cdots) \qquad (x_2 \text{ fresh})$$

$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ D[(\textbf{throw}\ d_1)]\ d_2\cdots) \rightarrow \quad \text{[TIThrow]}$$
$$(\textbf{store}\ ((x_s\ v_s)\cdots)\ d_1\ d_2\cdots)$$

Fig. 8. Top-level semantics, as an extension of Figure 1.

applied, its argument is substituted into the hole in the context where *call/cc* was originally invoked, and then the [TIThrow] rule replaces the current context with the context saved from the point where *call/cc* was invoked.

Because this system splits each **begin**$^{\textbf{D}}$ expression into its constituent pieces (via the [TIBegin] rule) before evaluating their bodies, it guarantees that the continuations grabbed by *call/cc* do not include **begin**$^{\textbf{D}}$ expressions and thus evaluating the example program from the beginning of this section results in a store that binds x to *3*.

8 Dynamic wind

Scheme's *dynamic-wind* allows for annotating the dynamic extent of a procedure call with entry and exit code that runs whenever the program flows into or out of that extent, either through normal program evaluation or through the invocation of the procedures made by *call/cc*. For example, a programmer may wish to ensure that a log file is always open during logging and properly closed when the program is not logging, even if the computation uses a continuation to jump in and out of the logging extent. The *with-logging* procedure provides this functionality:

```
;; with-logging : (((string → unspecified) → any) → any)
;; calls to-log-proc with a function that logs its argument
(define (with-logging to-log-proc)
  (let ((port #f))
    (dynamic-wind
      (lambda () (set! port (open-output-file "logfile" 'append)))
      (lambda () (to-log-proc (lambda (x) (write x port) (newline port))))
      (lambda () (close-output-port port)))))
```

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

If no continuation jumps occur, *dynamic-wind* just calls its three thunks in order, so *with-logging* would first open an output file,[6] then call *to-log-proc* with a function that writes to the port, and finally close the port. If, however, a continuation jump does occur during the call to *dynamic-wind*'s second argument, *dynamic-wind* would call its third argument as the continuation jumps out. Similarly, if a continuation is captured during the call to the second thunk, and is later used to jump back into the dynamic extent of *to-log-proc*'s application, *dynamic-wind* invokes the first thunk as control transfers back into the body of the second thunk. Taken together, this behavior ensures that *port* is always an open file during the call to *to-log-proc* and is closed otherwise.

Even though *dynamic-wind* has a large impact on the meaning of continuations, the Report formal semantics does not model it. Here we present a model of *dynamic-wind* that conforms to the Report, but not all implementations conforming to the Report necessarily match this model. In particular, section 6.4 of the Report says, "The effect of using a captured continuation to enter or exit the dynamic extent of a call to [the first thunk] or [the last thunk] is undefined." Our semantics, however, follows the working draft of the next version of the Report (Dybvig *et al.* 2006), which says that "[t]he in and out thunks of a dynamic-wind are considered 'outside' of the dynamic-wind; that is, escaping from either does not cause the [third] thunk to be invoked, and jumping back in does not cause the [first] thunk to be invoked." Our model of *dynamic-wind* is based on earlier treatments (Haynes & Friedman 1987; Felleisen 1988; Gasbichler *et al.* 2003).

The language in Figure 9 consists of the core Scheme with mutation as shown in figure 1 augmented with *call/cc* and *dynamic-wind*. The basic strategy is to maintain a stack of all *dynamic-wind* calls entered but not yet exited. When a continuation is captured, the semantics records the current dynamic-wind stack. When throwing to a continuation, the semantics uses the difference between the current dynamic-wind stack and the one that is associated with the captured continuation to determine which 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 *dynamic-wind* that has been entered in the current evaluation. 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 produced. 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.

[6] This code uses the '*append* mode specifier so the file is not overwritten, but that is not part of the Report.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$$
\begin{array}{lcl}
p & ::= & (\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws\ \cdots)\ e)) \\
dws & ::= & (x\ e\ e) \\
e & ::= & \ldots \mid (\textbf{push}\ dws) \mid (\textbf{pop}) \mid (\textbf{throw}\ dws\ \cdots\ e) \\
v & ::= & \ldots \mid \mathit{dynamic\text{-}wind} \mid \mathit{call/cc} \\
P & ::= & (\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws\ \cdots)\ E))
\end{array}
$$

$$
\begin{array}{ll}
P[(\mathit{dynamic\text{-}wind}\ (\textbf{lambda}\ ()\ e_1)\ (\textbf{lambda}\ ()\ e_2)\ (\textbf{lambda}\ ()\ e_3))] \rightarrow & \text{[DWWind]} \\
P[(\textbf{begin}\ e_1\ (\textbf{push}\ (x_1\ e_1\ e_3))\ ((\textbf{lambda}\ (x_2)\ (\textbf{begin}\ (\textbf{pop})\ e_3\ x_2))\ e_2))] & (x_1, x_2 \text{ fresh}) \\
(\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots)\ E[(\textbf{push}\ dws_2)])) \rightarrow & \text{[DWPush]} \\
(\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots\ dws_2)\ E[\mathit{unspecified}])) & \\
(\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots\ dws_n)\ E[(\textbf{pop})])) \rightarrow & \text{[DWPop]} \\
(\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots)\ E[\mathit{unspecified}])) & \\
(\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots)\ E[(\mathit{call/cc}\ v_1)])) \rightarrow & \text{[DWCallcc]} \\
(\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots)\ E[(v_1\ (\textbf{lambda}\ (x)\ (\textbf{throw}\ dws_1\ \cdots\ E[x])))])) & (x \text{ fresh}) \\
(\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots)\ E[(\textbf{throw}\ dws_2\ \cdots\ e_1)])) \rightarrow & \text{[DWThrow]} \\
(\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots)\ (\textbf{begin}\ \mathscr{T}\ [\![\ (dws_1\ \cdots), (dws_2\ \cdots)\ ]\!]\ e_1))) &
\end{array}
$$

$$
\begin{array}{ll}
\mathscr{T} : (\mathit{sequence\text{-}of}\ dws) \times (\mathit{sequence\text{-}of}\ dws) \longrightarrow (\mathit{listof}\ e) & \\
\mathscr{T}[\![\ ((x_1\ e_1\ e_2)\ dws_1\ \cdots), ((x_2\ e_3\ e_4)\ dws_2\ \cdots)\ ]\!] = & \\
\quad \mathscr{T}[\![\ (dws_1\ \cdots), (dws_2\ \cdots)\ ]\!] & (\text{if } x_1 = x_2) \\
\mathscr{T}[\![\ ((x_1\ e_1\ e_2)\ \cdots), ((x_2\ e_3\ e_4)\ \cdots)\ ]\!] = & \\
\quad (\textbf{begin}\ (\textbf{pop})\ e_2)\ \cdots_r\ (\textbf{begin}\ e_3\ (\textbf{push}\ x_2\ e_3\ e_4))\ \cdots & (\text{otherwise})
\end{array}
$$

Fig. 9. Call/cc and dynamic-wind semantics, as an extension of Figure 1.

Third, when *call/cc* is called, the [DWCallcc] rule builds a continuation that consists of a procedure of one argument, 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 evaluated using the [DWThrow] rule. It discards the evaluation context in which it was found, replacing the entire program body with a specially constructed **begin** expression built by combining the result of the $\mathscr{T}$ metafunction and the body of the **throw**.

The $\mathscr{T}$ function trims away the common context frames leaving only the suffixes that need to be executed to return the dynamic context to its state when the continuation was captured. Intuitively, it constructs a sequence of *pre* and *post* thunks that correspond to the shortest path through the state space, in the sense of Haynes and Friedman (1987). The metafunction 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 not left or entered from the time the continuation was created to the time it was invoked. Then, once the two stacks have been trimmed, the metafunction produces a list of expressions consisting of applications of all

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

the *post* thunks from $\mathscr{T}$'s first argument and **pop** expressions to erase the current dynamic context, followed by all the *pre* thunks from $\mathscr{T}$'s second argument and **push** expressions to restore the old dynamic context. The thunks from $\mathscr{T}$'s first argument are invoked in reverse order (which we indicate with the special notation $\cdots_r$) in order to erase the dynamic context in the opposite order from which it was created.

PART TWO
The Combined Reduction System

This part combines the techniques described in part 1 with standard techniques for modeling programming languages to define a semantics for the Scheme programming language of the Report. The semantics covers variable mutation, mutable lists, μlambda procedures (i.e., procedures that accept extra arguments as a list), *apply*, object identity based equivalence, and all of the features from part 1. There are a number of parts of the Report that are not covered, but most of them can either be defined in terms of existing features (e.g., **let**), are similar to features that are modeled (e.g., *eq?*), or would not add anything particularly interesting to the model (e.g., strings). Beyond those, we do not model Scheme's numeric tower, macros, or input and output, partly to keep the model a manageable size.

In general, the Report is not a complete specification, in order to give implementations freedom to behave differently, typically to allow optimizations. This underspecification shows up in a number of ways in our semantics, but the primary technique we use to model it is to have the single-step relation relate one program to multiple different programs, each corresponding to a legal transition that the abstract machine might take. Accordingly we use the transitive closure of the single step relation to define the semantics, $\mathcal{S}$, as a function from programs ($\mathcal{P}$) to sets of answers ($\mathcal{A}$):

$$\mathcal{S} : \mathcal{P} \to 2^{\mathcal{A}}$$
$$\mathcal{S}(\mathcal{P}) = \{ \mathcal{A} \mid \mathcal{P} \to^* \mathcal{A} \}$$

An implementation conforms to the semantics if, for every program $\mathcal{P}$, the implementation produces one of the results in $\mathcal{S}(\mathcal{P})$ or, if the implementation loops forever, then there is an infinite reduction sequence starting at $\mathcal{P}$. The precise definitions of $\mathcal{P}$ and $\mathcal{A}$ are given in section 9.

Our specification is executable, and the contents of all of the figures in this part were automatically generated from the source code that implements the specification, with the exception of the $\mathscr{R}$ metafunction, which was typeset by hand. Since an executable specification was an explicit goal of our work, we have made some modeling choices whose motivations may not be obvious at first. For example, there are many expressions whose return values are explicitly unspecified in the Report, such as the result of a **set!** expression. In part 1, we modeled them with a special value, called *unspecified*. A nonexecutable specification might not treat it as a value, and instead add the rule schema

$$\forall\ v.\ PC\,[\mathit{unspecified}] \to PC\,[v]$$

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

meaning that the unspecified value reduces to all possible values. In an executable specification, however, those reductions would overwhelm the ordinary reductions so we simply leave the value *unspecified* in our full semantics. Programs that attempt to inspect it by supplying it to other primitive operations will signal errors in our semantics, although conformance with the Report does not require these errors to be signaled (see section 18). Of course, programs that ignore it continue without incident. 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.

While we have not established any precise relationship between our semantics and the formal semantics in the Report, intuitively our semantics (when trimmed to the language in the Report's formal semantics) produces any result that the formal semantics might produce. The reverse is not true, because of the way we handle application expressions. As an example, our semantics produces *7*, *8*, *9*, and *10* for this program:

```
(define x 1) ((lambda (t) (t) (t))
               (lambda ()
                 ((lambda (a b) x)
                  (set! x (+ x 1))
                  (set! x (* x 2)))))
```

but the Report semantics can produce only *7* or *10*. Fundamentally, the difference is that our semantics can change the order of evaluation of the two **set!** expressions during evaluation, but the Report's semantics will pick only a single order both times the thunk is called. For a full discussion of the difference, see section 5.

9 Grammar

The grammar for programs in the Report is shown in Figure 10. In this figure, a program (given by the nonterminal *p*) consists of a store, a dynamic-wind stack, and a series of top-level definitions and expressions. The *sf* nonterminal generates bindings for the store. The *dws* nonterminal corresponds to one frame of dynamic-wind context information. The *d* nonterminal produces definitions (using **define**), top-level begin expressions (**begin**$^{\mathbf{D}}$), and expressions. The *e* nonterminal gives expressions, which in addition to standard Scheme core forms of application, **if**, **begin**, variables, **lambda** expressions and values, can be marked applications, as in section 4, and **throw**, **push**, and **pop**, as in section 8.

Values (*v*) are either procedures or nonprocedure's, but **lambda** terms are not values themselves—procedure values (*fun*) can be references to procedures in the store (*ufun*), or the built-in procedures, which are further refined into *fun1*, *fun2* and *aop*, in order to facilitate the error reductions. The **lambda** form places new procedure values into the store when evaluated so that we can specify the behavior of *eqv?* on procedures. As in section 6, we write dotted pairs with **dot** rather than a period to avoid meta-circular confusion in our PLT Redex implementation, and we take advantage of that to write arbitrary arity procedures as (**lambda** (**dot** *x*) *e e* ⋯).

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$$
\begin{array}{lcl}
p & ::= & (\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws\ \cdots)\ d\ d\ \cdots)) \\
sf & ::= & (x\ v)\ \mid\ (pp\ (\#\%cons\ v\ v))\ \mid\ (cp\ (\textbf{lambda}\ (x\ \cdots)\ e\ e\ \cdots)) \\
 & \mid & (mp\ (\textbf{lambda}\ (x\ x\ \cdots\ \textbf{dot}\ x)\ e\ e\ \cdots))\ \mid\ (mp\ (\textbf{lambda}\ x\ e\ e\ \cdots)) \\
dws & ::= & (x\ (v)\ (v)) \\
d & ::= & (\textbf{define}\ x\ e)\ \mid\ (\textbf{begin}^{\mathbf{D}}\ d\ \cdots)\ \mid\ e \\
e & ::= & (e\ e\ \cdots)\ \mid\ (e\ \cdots\ e^{\diamond}\ 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\ x\ \mid\ v\ \mid\ (\textbf{push}\ dws)\ \mid\ (\textbf{pop}) \\
 & \mid & (\textbf{throw}\ x\ dws\ \cdots\ D[x])\ \mid\ (\textbf{lambda}\ (x\ \cdots)\ e\ e\ \cdots) \\
 & \mid & (\textbf{lambda}\ (x\ x\ \cdots\ \textbf{dot}\ x)\ e\ e\ \cdots)\ \mid\ (\textbf{lambda}\ x\ e\ e\ \cdots) \\
v & ::= & nonfun\ \mid\ fun \\
nonfun & ::= & pp\ \mid\ \#\%null\ \mid\ 'sym\ \mid\ sqv\ \mid\ unspecified \\
fun & ::= & ufun\ \mid\ aop\ \mid\ fun1\ \mid\ fun2\ \mid\ \#\%list\ \mid\ \#\%dynamic\text{-}wind\ \mid\ \#\%apply\ \mid\ \#\%values \\
fun1 & ::= & \#\%null?\ \mid\ \#\%pair?\ \mid\ \#\%car\ \mid\ \#\%cdr\ \mid\ \#\%call/cc\ \mid\ \#\%eval \\
fun2 & ::= & \#\%cons\ \mid\ \#\%set\text{-}car!\ \mid\ \#\%set\text{-}cdr!\ \mid\ \#\%eqv?\ \mid\ \#\%call\text{-}with\text{-}values \\
ufun & ::= & cp\ \mid\ mp \\
aop & ::= & \#\%+\ \mid\ \#\%-\ \mid\ \#\%/\ \mid\ \#\%* \\
 & & \\
P & ::= & (\textbf{store}\ (sf\ \cdots)\ W) \\
W & ::= & (\textbf{dw}\ (dws\ \cdots)\ D\ d\ \cdots) \\
D & ::= & E^{\star}\ \mid\ (\textbf{define}\ x\ E^{\circ}) \\
E & ::= & [\,]\ \mid\ (e\ \cdots\ E^{\circ\diamond}\ e\ \cdots)\ \mid\ (\textbf{if}\ E^{\circ}\ e\ e)\ \mid\ (\textbf{if}\ E^{\circ}\ e)\ \mid\ (\textbf{set!}\ x\ E^{\circ}) \\
 & \mid & (\textbf{begin}\ E^{\star}\ e\ e\ \cdots)\ \mid\ (\#\%call\text{-}with\text{-}values\ (\textbf{lambda}\ ()\ E^{\star}\ e\ \cdots)\ v) \\
E^{\circ} & ::= & [\,]_{\circ}\ \mid\ E \\
E^{\star} & ::= & [\,]_{\star}\ \mid\ E \\
ds & ::= & es\ \mid\ (\textbf{define}\ x\ es)\ \mid\ (\textbf{begin}^{\mathbf{D}}\ ds\ \cdots) \\
es & ::= & 's\ \mid\ (\textbf{ccons}\ es\ es)\ \mid\ (es\ es\ \cdots)\ \mid\ (es\ \cdots\ es^{\diamond}\ es\ \cdots) \\
 & \mid & (\textbf{if}\ es\ es\ es)\ \mid\ (\textbf{if}\ es\ es)\ \mid\ (\textbf{set!}\ x\ es)\ \mid\ (\textbf{begin}\ es\ es\ \cdots)\ \mid\ x\ \mid\ v\ \mid\ (\textbf{push}\ dws) \\
 & \mid & (\textbf{pop})\ \mid\ (\textbf{throw}\ x\ dws\ \cdots\ D[x])\ \mid\ (\textbf{lambda}\ (x\ \cdots)\ es\ es\ \cdots) \\
 & \mid & (\textbf{lambda}\ (x\ x\ \cdots\ \textbf{dot}\ x)\ es\ es\ \cdots)\ \mid\ (\textbf{lambda}\ x\ es\ es\ \cdots) \\
s & ::= & (s\ \cdots)\ \mid\ (s\ \cdots\ s\ \textbf{dot}\ sqv)\ \mid\ (s\ \cdots\ s\ \textbf{dot}\ sym)\ \mid\ sqv\ \mid\ sym \\
sqv & ::= & n\ \mid\ \#t\ \mid\ \#f \\
SP & ::= & (\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws\ \cdots)\ d\ \cdots\ SD\ s\ \cdots)) \\
SD & ::= & S\ \mid\ (\textbf{define}\ x\ S)\ \mid\ (\textbf{begin}^{\mathbf{D}}\ d\ \cdots\ SD\ s\ \cdots) \\
S & ::= & [\,]\ \mid\ (e\ \cdots\ S\ s\ \cdots)\ \mid\ (\textbf{if}\ e\ e\ S)\ \mid\ (\textbf{if}\ e\ S\ s)\ \mid\ (\textbf{if}\ S\ s\ s)\ \mid\ (\textbf{if}\ e\ S)\ \mid\ (\textbf{if}\ S\ s) \\
 & \mid & (\textbf{set!}\ x\ S)\ \mid\ (\textbf{begin}\ e\ e\ \cdots\ S\ s\ \cdots)\ \mid\ (\textbf{begin}\ S\ s\ \cdots)\ \mid\ (\textbf{throw}\ x\ dws\ \cdots\ S) \\
 & \mid & (\textbf{push}\ (x\ e\ S))\ \mid\ (\textbf{push}\ (x\ S\ s))\ \mid\ (\textbf{lambda}\ (x\ \cdots)\ S\ s\ \cdots) \\
 & \mid & (\textbf{lambda}\ (x\ \cdots)\ e\ e\ \cdots\ S\ s\ \cdots)\ \mid\ (\textbf{lambda}\ (x\ x\ \cdots\ \textbf{dot}\ x)\ S\ s\ \cdots) \\
 & \mid & (\textbf{lambda}\ (x\ x\ \cdots\ \textbf{dot}\ x)\ e\ e\ \cdots\ S\ s\ \cdots)\ \mid\ (\textbf{lambda}\ x\ S\ s\ \cdots) \\
 & \mid & (\textbf{lambda}\ x\ e\ e\ \cdots\ S\ s\ \cdots)\ \mid\ (\textbf{ccons}\ v\ S)\ \mid\ (\textbf{ccons}\ S\ s)\ \mid\ S^{\diamond} \\
sym & ::= & [\text{variables except } \textbf{dot}] \\
x & ::= & [\text{variables except } \textbf{dot} \text{ and keywords}] \\
pp & ::= & [\text{pair pointers}] \\
cp & ::= & [\text{closure pointers}] \\
mp & ::= & [\mu\ \text{closure pointers}] \\
n & ::= & [\text{numbers}] \\
 & & \\
\mathcal{P} & ::= & (\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws\ \cdots)\ ds\ ds\ \cdots)) \\
\mathcal{A} & ::= & (\textbf{store}\ (sf\ \cdots)\ (\textbf{dw}\ (dws\ \cdots)\ (\#\%values\ v\ \cdots)))\ \mid\ \textbf{error:}\ error\ message
\end{array}
$$

Fig. 10. Grammar.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

instead of the traditional (**lambda** *x e e* ⋯). Nonprocedure values (*nonfun*) include pair pointers, numbers, the empty list (*null*), booleans, and the value *unspecified*.

Section 6 of the Report 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 primitives, and we bind those values to their *#%*-less names in an initial store:

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

We use four different kinds of contexts: program evaluation contexts (P), dynamic-wind contexts (W), definition contexts (D), and expression contexts (E, E°, and $E^{\star}$). Program contexts, dynamic-wind contexts, and definition contexts nest inside each other, and they all accept expressions in their holes. Evaluation takes place in expression contexts; they allow evaluation in marked subexpressions of an application, the test positions of **if** expressions, in **set!** expressions, in the first position in a **begin**, and in the body of thunks passed to *#%call-with-values*. The E° context expects a single value and the $E^{\star}$ context expects multiple values. Accordingly, top-level expressions may reduce to multiple values, but the expression on the right-hand side of a definition must be a single value.

Programs with quoted s-expressions are generated by the *ds* and *es* nonterminals. They are just like *d* and *e*, respectively, but include quoted expressions and **ccons** expressions. S-expressions are generated by the *s* and the *sqv* nonterminals. The *sqv* nonterminal is named for self-quoting values; i.e., those values where adding or removing a **quote** does not change the value. S-expression contexts are generated by the SP, SD, and S nonterminals and correspond to places where a **quote**d expression can be moved into the store. They are larger here than in section 6 because of the additional syntactic forms in this language.

The *sym* nonterminal represents symbols, 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.

Finally, the $\mathcal{P}$ and $\mathcal{A}$ nonterminals specify what complete programs are and help define the domain of the reduction relation. See section 19 for details.

10 Basic syntactic forms and arithmetic

Figure 11 displays the 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

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$$
\begin{array}{lll}
P[(\textbf{if}\ v_1\ e_1\ e_2)] & \rightarrow\ P[e_1] \quad (v_1 \neq \#f) & \text{[5if3t]} \\
P[(\textbf{if}\ \#f\ e_1\ e_2)] & \rightarrow\ P[e_2] & \text{[5if3f]} \\
P[(\textbf{if}\ v_1\ e_1)] & \rightarrow\ P[e_1] \quad (v_1 \neq \#f) & \text{[5if2t]} \\
P[(\textbf{if}\ \#f\ e_1)] & \rightarrow\ P[\textit{unspecified}] & \text{[5if2f]} \\
P[(\textbf{begin}\ (\#\%\textit{values}\ v \cdots)\ e_1\ e_2 \cdots)] & \rightarrow\ P[(\textbf{begin}\ e_1\ e_2 \cdots)] & \text{[5beginc]} \\
P[(\textbf{begin}\ e_1)] & \rightarrow\ P[e_1] & \text{[5beginl]}
\end{array}
$$

Fig. 11. Basic syntactic forms.

$$
\begin{array}{lll}
P[(\#\%+)] & \rightarrow\ P[0] & \text{[5+0]} \\
P[(\#\%+\ n_1\ n_2 \cdots)] & \rightarrow\ P[\ \ulcorner \Sigma\{n_1, n_2 \cdots\}\urcorner\ ] & \text{[5+]} \\
P[(\#\%-\ n_1)] & \rightarrow\ P[\ \ulcorner -\ n_1\urcorner\ ] & \text{[5u-]} \\
P[(\#\%-\ n_1\ n_2\ n_3 \cdots)] & \rightarrow\ P[\ \ulcorner n_1 - \Sigma\{n_2, n_3 \cdots\}\urcorner\ ] & \text{[5-]} \\
P[(\#\%*)] & \rightarrow\ P[1] & \text{[5*1]} \\
P[(\#\%*\ n_1\ n_2 \cdots)] & \rightarrow\ P[\ \ulcorner \Pi\{n_1, n_2 \cdots\}\urcorner\ ] & \text{[5*]} \\
P[(\#\%/\ n_1)] & \rightarrow\ P[(\#\%/\ 1\ n_1)] & \text{[5u/]} \\
P[(\#\%/\ n_1\ n_2\ n_3 \cdots)] & \rightarrow\ P[\ \ulcorner n_1 / \Pi\{n_2, n_3 \cdots\}\urcorner\ ] \quad (0 \notin \{\ n_2, n_3 \cdots\}) & \text{[5/]}
\end{array}
$$

Fig. 12. Arithmetic.

contexts defined in Figure 10 ensure that the first term of a **begin** expression is evaluated fully; then these rules rewrite **begin** expressions that consist of a value followed by other expressions to a new **begin** expression without the initial value. These rules also specifiy that a **begin** form with only a single expression reduces immediately to that expression even if that expression is not yet a value (or if it is a single value or multiple values).

Because our model does not take into account the Report's numeric tower, we express its numeric operations in terms of true mathematical functions, as shown in Figure 12. 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: + is modeled by summation on the represented numbers, * is modeled by product, and so on. The figures use the notation $\ulcorner n \urcorner$ to represent the mathematical number n's written form.

11 Lists

The rules for lists and operations on lists are given in Figure 13. Since all cons cells are mutable and therefore can be distinguished even when they hold identical values, the semantics must be able to reflect such distinctions. So, (*#%cons v v*) itself is not

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$(\textbf{store}\ (sf_1 \cdots)\ W_1[(\#\%cons\ v_1\ v_2)]) \rightarrow$ [5cons]
$(\textbf{store}\ (sf_1 \cdots (pp_i\ (\#\%cons\ v_1\ v_2)))\ W_1[pp_i])$ $(pp_i$ fresh$)$

$P[(\#\%list\ v_1\ v_2 \cdots)] \rightarrow$ [5listc]
$P[(\#\%cons\ v_1\ (\#\%list\ v_2 \cdots))]$

$P[(\#\%list)] \rightarrow$ [5listn]
$P[\#\%null]$

$(\textbf{store}\ (sf_1 \cdots (pp_i\ (\#\%cons\ v_1\ v_2))\ sf_2 \cdots)\ W_1[(\#\%car\ pp_i)]) \rightarrow$ [5car]
$(\textbf{store}\ (sf_1 \cdots (pp_i\ (\#\%cons\ v_1\ v_2))\ sf_2 \cdots)\ W_1[v_1])$

$(\textbf{store}\ (sf_1 \cdots (pp_i\ (\#\%cons\ v_1\ v_2))\ sf_2 \cdots)\ W_1[(\#\%cdr\ pp_i)]) \rightarrow$ [5cdr]
$(\textbf{store}\ (sf_1 \cdots (pp_i\ (\#\%cons\ v_1\ v_2))\ sf_2 \cdots)\ W_1[v_2])$

$P[(\#\%null?\ \#\%null)] \rightarrow$ [5null?t]
$P[\#t]$

$P[(\#\%null?\ v_1)] \rightarrow$ [5null?f]
$P[\#f]$ $(v_1 \neq \#\%null)$

$P[(\#\%pair?\ pp)] \rightarrow$ [5pair?t]
$P[\#t]$

$P[(\#\%pair?\ v_1)] \rightarrow$ [5pair?f]
$P[\#f]$ $(v_1 \notin pp)$

$(\textbf{store}\ (sf_1 \cdots (pp_1\ (\#\%cons\ v_1\ v_2))\ sf_2 \cdots)\ W_1[(\#\%set\text{-}car!\ pp_1\ v_3)]) \rightarrow$ [5setcar]
$(\textbf{store}\ (sf_1 \cdots (pp_1\ (\#\%cons\ v_3\ v_2))\ sf_2 \cdots)\ W_1[unspecified])$

$(\textbf{store}\ (sf_1 \cdots (pp_1\ (\#\%cons\ v_1\ v_2))\ sf_2 \cdots)\ W_1[(\#\%set\text{-}cdr!\ pp_1\ v_3)]) \rightarrow$ [5setcdr]
$(\textbf{store}\ (sf_1 \cdots (pp_1\ (\#\%cons\ v_1\ v_3))\ sf_2 \cdots)\ W_1[unspecified])$

Fig. 13. Lists.

a value; instead, the *#%cons* rule introduces a new pair into the store and reduces to a pointer to that new pair. The [5listc] and [5listn] rules rewrites *#%list* expressions to a sequence of calls to *#%cons*. The rules for *#%car* and *#%cdr* rewrite calls to either procedure to the appropriate field of a pair, extracting the field's value from the store.

The predicates in the figure are similarly straightforward. 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.

The *#%set-car!* and *#%set-cdr!* rules are similar to the *#%car* and *#%cdr* rules; they update the store and rewrite to the unspecified value.

12 Top-level and variables

Figure 14 gives the reduction rules for top-level definitions and variables in the store. Top-level definitions reduce either by extending the store with the new definition or, if the variable is already bound in the store, by updating the store with the new variable. The side condition on [5def] ensures that the rule applies only when x is

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

(**store** ($sf_1 \cdots$) (**dw** ($dws_1 \cdots$) (**define** x_1 v_1) $d_1 \cdots$)) → [5def]
(**store** ($sf_1 \cdots$ (x_1 v_1)) (**dw** ($dws_1 \cdots$) *unspecified* $d_1 \cdots$)) ($x_1 \notin \mathrm{dom}(sf_1 \cdots)$)

(**store** ($sf_1 \cdots$ (x_1 v_2) $sf_2 \cdots$) (**dw** ($dws_1 \cdots$) (**define** x_1 v_1) $d_1 \cdots$)) → [5redef]
(**store** ($sf_1 \cdots$ (x_1 v_1) $sf_2 \cdots$) (**dw** ($dws_1 \cdots$) *unspecified* $d_1 \cdots$))

(**store** ($sf_1 \cdots$) (**dw** ($dws_1 \cdots$) (**begin**$^{\mathbf{D}}$ d_1 $d_2 \cdots$) $d_3 \cdots$)) → [5tbegin]
(**store** ($sf_1 \cdots$) (**dw** ($dws_1 \cdots$) d_1 $d_2 \cdots$ $d_3 \cdots$))

(**store** ($sf_1 \cdots$) (**dw** ($dws_1 \cdots$) (**begin**$^{\mathbf{D}}$) $d_1 \cdots$)) → [5tbegine]
(**store** ($sf_1 \cdots$) (**dw** ($dws_1 \cdots$) *unspecified* $d_1 \cdots$))

(**store** ($sf_1 \cdots$) (**dw** ($dws_1 \cdots$) (*#%values* $v \cdots$) d_1 $d_2 \cdots$)) → [5tdrop]
(**store** ($sf_1 \cdots$) (**dw** ($dws_1 \cdots$) d_1 $d_2 \cdots$))

(**store** ($sf_1 \cdots$ (x_1 v_1) $sf_2 \cdots$) $W_1[x_1]$) → [5var]
(**store** ($sf_1 \cdots$ (x_1 v_1) $sf_2 \cdots$) $W_1[v_1]$)

(**store** ($sf_1 \cdots$ (x_1 v_1) $sf_2 \cdots$) W_1[(**set!** x_1 v_2)]) → [5set]
(**store** ($sf_1 \cdots$ (x_1 v_2) $sf_2 \cdots$) W_1[*unspecified*])

Fig. 14. Top level and variables.

not already bound in the store. Top-level **begin**$^{\mathbf{D}}$ expressions (see section 7 for a discussion of **begin**$^{\mathbf{D}}$) are erased, and their contents are flattened into the sequence of definitions at the top level. Expressions that have been reduced to values at the top level are dropped, as long as there are more definitions to evaluate. The values of the final definition are retained. A reference to a variable in the store is replaced by its value, and an assignment to a variable in the store updates the store with the new value.

13 Procedures

The rules for procedure fall into three categories: procedure introduction, application marking, and procedure application, as shown in Figure 15. Like cons cells, procedures are not values, but pointers to them are. Procedures are modeled this way so that we can model *eqv?*. The rule [5calloc] allocates fixed arity procedures. The allocation for μlambda procedures always puts two procedures into the store: a stub μlambda procedure whose body contains a call to an ordinary procedure, and an ordinary procedure that contains the original μlambda's body expressions. We put both procedures into the store so that when a μlambda 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 [5mark] and [5unmark] show how marks are placed into and removed from applications, almost as in section 4. The only difference is that thunks that appear as the first argument to *#%call-with-values* are not marked (and thus is not moved into the store) in order to support evaluation in the body of the thunk (see section 15).

Application of a procedure pointer to arguments is modeled by creating one new binding in the store per formal argument, replacing the formal parameters in the

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

(**store** ($sf_1 \cdots$) W_1[(**lambda** ($x_1 \cdots$) $e_1\ e_2 \cdots$)]) → [5calloc]
(**store** ($sf_1 \cdots$ (cp (**lambda** ($x_1 \cdots$) $e_1\ e_2 \cdots$))) W_1[cp]) (cp fresh)

(**store** ($sf_1 \cdots$) W_1[(**lambda** ($x_1\ x_2 \cdots$ **dot** x_r) $e_1\ e_2 \cdots$)]) → [5μcalloc]
(**store** ($sf_1 \cdots$
(mp (**lambda** ($x_1\ x_2 \cdots$ **dot** x_r) ($cp\ x_1\ x_2 \cdots\ x_r$)))
(cp (**lambda** ($x_1\ x_2 \cdots\ x_r$) $e_1\ e_2 \cdots$)))
W_1[mp]) (mp, cp fresh)

(**store** ($sf_1 \cdots$) W_1[(**lambda** $x_1\ e_1\ e_2 \cdots$)]) → [5μcalloc1]
(**store** ($sf_1 \cdots$
(mp (**lambda** x_1 ($cp\ x_1$)))
(cp (**lambda** (x_1) $e_1\ e_2 \cdots$)))
W_1[mp]) (mp, cp fresh)

P_1[($e_1 \cdots e_i\ e_{i+1} \cdots$)] → [5mark]
P_1[($e_1 \cdots e_i^{\diamond}\ e_{i+1} \cdots$)] ($e_i \notin v$, e_i is not a thunk in a *#%call-with-values* application)

P[($e_1 \cdots v_i^{\diamond}\ e_{i+1} \cdots$)] → [5unmark]
P[($e_1 \cdots v_i\ e_{i+1} \cdots$)]

(**store** ($sf_1 \cdots$ (cp_i (**lambda** ($x_1 \cdots$) $e_{body1}\ e_{body2} \cdots$)) $sf_2 \cdots$) [5app]
W_1[($cp_i\ v_1 \cdots$)]) →
(**store** ($sf_1 \cdots$
(cp_i (**lambda** ($x_1 \cdots$) $e_{body1}\ e_{body2} \cdots$))
$sf_2 \cdots$
($x_2\ v_1$) $\cdots$)
W_1[{ $x_1 \cdots \mapsto x_2 \cdots$ }(**begin** $e_{body1}\ e_{body2} \cdots$)]) ($\#x_1 = \#v_1$, $x_2 \cdots$ fresh)

(**store** ($sf_1 \cdots$ (mp_i (**lambda** ($x_1\ x_2 \cdots$ **dot** x_r) ($cp_1\ x_1\ x_2 \cdots\ x_r$))) $sf_2 \cdots$) [5μapp]
W_1[($mp_i\ v_1 \cdots\ v_2 \cdots$)]) →
(**store** ($sf_1 \cdots$ (mp_i (**lambda** ($x_1\ x_2 \cdots$ **dot** x_r) ($cp_1\ x_1\ x_2 \cdots\ x_r$))) $sf_2 \cdots$)
W_1[($cp_1\ v_1 \cdots$ (*#%list* $v_2 \cdots$))]) ($\#v_1 = \#x_2 + 1$)

(**store** ($sf_1 \cdots$ (mp_i (**lambda** x_1 ($cp_1\ x_1$))) $sf_2 \cdots$) [5μapp1]
W_1[($mp_i\ v_1 \cdots$)]) →
(**store** ($sf_1 \cdots$ (mp_i (**lambda** x_1 ($cp_1\ x_1$))) $sf_2 \cdots$)
W_1[(cp_1 (*#%list* $v_1 \cdots$))])

(**store** ($sf_1 \cdots$ (pp_i (*#%cons* $v_2\ v_3$)) $sf_2 \cdots$) [5applyc]
W_1[(*#%apply* $fun_1\ v_1 \cdots\ pp_i$)]) →
(**store** ($sf_1 \cdots$ (pp_i (*#%cons* $v_2\ v_3$)) $sf_2 \cdots$)
W_1[(*#%apply* $fun_1\ v_1 \cdots\ v_2\ v_3$)])

P[(*#%apply* $fun_1\ v_1 \cdots$ *#%null*)] → [5applyn]
P[($fun_1\ v_1 \cdots$)]

Fig. 15. Procedures.

body with the new variables, and binding these new variables in the store to the actual parameters. Application of a μlambda, [5μapp], allocates a list for its extra arguments, applies the initial portion of the arguments as usual, and constructs a list containing the rest of the arguments to be supplied to cp, the procedure that contains the actual body of the original procedure.

The last two rules in Figure 15 specify the behavior of Scheme's *apply* procedure. It 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

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

P_1[(#%*dynamic-wind* v_1 v_2 v_3)] → [5dw]
P_1[(**begin** (v_1)
(**push** (d (v_1) (v_3)))
(#%*call-with-values*
v_2
(**lambda** x
(**pop**) (v_3) (#%*apply* #%*values* x))))] (v_1, v_2, & v_3 arity 0, d, x, cp, mp fresh)

(**store** (sf_1 ⋯) (**dw** (dws_1 ⋯) D_1[(**push** dws_2)] d_1 ⋯)) → [5push]
(**store** (sf_1 ⋯) (**dw** (dws_1 ⋯ dws_2) D_1[*unspecified*] d_1 ⋯))

(**store** (sf_1 ⋯) (**dw** (dws_1 ⋯ dws_2) D_1[(**pop**)] d_1 ⋯)) → [5pop]
(**store** (sf_1 ⋯) (**dw** (dws_1 ⋯) D_1[*unspecified*] d_1 ⋯))

(**store** (sf_1 ⋯) (**dw** (dws_1 ⋯) D_1[(#%*call/cc* v_1)] d_1 ⋯)) → [5callcc]
(**store** (sf_1 ⋯) (**dw** (dws_1 ⋯)
D_1[(v_1 (**lambda** (**dot** x_1)
(**throw** x_2 dws_1 ⋯
D_1[(**begin** x_2 (#%*apply* #%*values* x_1))])))]
d_1 ⋯)) (x_1, x_2 fresh)

(**store** (sf_1 ⋯) (**dw** (dws_1 ⋯) D_1[(**throw** x_2 dws_2 ⋯ D_2[(**begin** x_2 e_1)])] d_1 ⋯)) → [5throw]
(**store** (sf_1 ⋯) (**dw** (dws_1 ⋯) D_2[(**begin** $\mathscr{T}$(((dws_1 ⋯) , (dws_2 ⋯)) e_1)] d_1 ⋯))

Fig. 16. Call/cc and dynamic-wind

as subsequent arguments. To specify this behavior, the two *#%apply* rules flatten out the argument list and, when the list is exhausted, reduce to a normal application.

14 Call/cc and dynamic-wind

Our technique for modeling *call/cc* and *dynamic-wind* in the full semantics, shown in Figure 16, is essentially the technique from section 8. Apart from the change of using all function values rather than just (**lambda** () e) expressions, the only substantial change concerns the continuation procedures. In the model for the Report language, they accept any number of arguments, which become multiple return values when the continuation is invoked. The trimming metafunction $\mathscr{T}$ is the same as the function defined in Figure 9.

The first rule rewrites *#%dynamic-wind* to an expression that invokes its first argument thunk, pushes the dynamic context, invokes the second thunk, pops the dynamic context, invokes the third thunk, and returns the result of the second thunk. To ensure that the three thunks are invoked in the proper order but that the value of the second thunk is returned, the rule uses an auxiliary procedure. The idea is that the procedure's argument holds the value of the second thunk while the third thunk is invoked. If *dynamic-wind* only allowed a single value to be returned from its second argument thunk, an expression similar to the one in the rule from section 8 would suffice.

(**begin** (v_1)
(**push** ⋯)
((**lambda** (x) (**pop**) (v_3) x)
(v_2)))

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$P_1[v_1]_\star \rightarrow$ [5promote]
$P_1[(\#\%values\ v_1)]$

$P_1[(\#\%values\ v_1)]_\circ \rightarrow$ [5demote]
$P_1[v_1]$

$P_1[(\#\%call\text{-}with\text{-}values\ (\textbf{lambda}\ ()\ (\#\%values\ v_2\ \cdots))\ v_1)] \rightarrow$ [5cwvd]
$P_1[(v_1\ v_2\ \cdots)]$

$P_1[(\#\%call\text{-}with\text{-}values\ (\textbf{lambda}\ ()\ (\#\%values\ v_1\ \cdots)\ e_1\ e_2\ \cdots)\ v_2)] \rightarrow$ [5cwvc]
$P_1[(\#\%call\text{-}with\text{-}values\ (\textbf{lambda}\ ()\ e_1\ e_2\ \cdots)\ v_2)]$

$P_1[(\#\%call\text{-}with\text{-}values\ v_1\ v_2)] \rightarrow$ [5cwvw]
$P_1[(\#\%call\text{-}with\text{-}values\ (\textbf{lambda}\ ()\ (v_1))\ v_2)]$

Fig. 17. Multiple values and call-with-values.

To cope with multiple values, however, v_2 must be invoked via *call-with-values*, as shown in the rule [5dw].

The second and third rules manipulate the dynamic context. The fourth rule handles *#%call/cc*; it builds a variable arity procedure whose body throws to a continuation and then passes that procedure to *#%call/cc*'s argument. The variable x_2 is used by the **throw** rule, [5throw]. That rule restores the definition evaluation context from the point where the continuation was captured and, using the $\mathscr{T}$ metafunction, inserts code at x_2 to adjust the dynamic context.

15 Multiple values and call-with-values

Figure 17 shows our treatment of multiple values in the full language. It is nearly identical to multiple values in section 5, and in particular the context arrangement, promotion and demotion rules are the same: rule [5promote] promotes a single value by wrapping it with a call to *#%values* in a multivalues context, and rule [5demote] demotes a single value that is wrapped with *#%values* to a single value when it occurs in a single-value context.

There is one twist, though, because **lambda** expressions in this semantics are not values, but are moved into the store. To support *#%call-with-values*, the rule for marking applications ([5mark] in figure 15) treats expressions of the form (**lambda** () *e e* $\cdots$) as values, when they appear as the second argument to *#%call-with-values*. In addition, the evaluation contexts from figure 10 allow evaluation in the body of such lambda expressions, meaning that the rule [5cwvd] handles the main job of *#%call-with-values*, *i.e.*, combining its second argument with the values returned by its first argument.

The next rule, [5cwvc] supports **lambda** expressions with multiple body expressions; once an intermediate body expression is evaluated, its result is discarded, and evaluation continues with the next one. The [5cwvw] handles the situation where *#%call-with-values*'s first argument is already a thunk in the store, or is a primitive procedure.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$P[(\#\%eqv?\ v_1\ v_1)]$	$\rightarrow$	$P[\#t]$	[5eqt]
$P[(\#\%eqv?\ v_1\ v_2)]$	$\rightarrow$	$P[\#f]$ $(v_1 \neq v_2)$	[5eqf]
$P[(\#\%eqv?\ ufun_1\ ufun_2)]$	$\rightarrow$	$P[\#t]$ ($ufun_1 \neq ufun_2$, $ufun_1$ & $ufun_2$ observably equivalent)	[5eqproof]

Fig. 18. Eqv and equivalence.

16 Eqv? and equivalence

The Report does not specify the entire behavior of *eqv?*, but does require conforming implementations of *eqv?* to satisfy these properties (when supplied with two arguments):

- *eqv?* must return *#t* if its arguments refer to the same locations in the store;
- *eqv?* must return *#f* it its arguments are functions that behave differently;
- *eqv?* may return *#t* or *#f* if its arguments are functions that behavfe the same way for all possible inputs, but have different locations; and
- *eqv?* must always return and always produce a boolean value.

We capture this behavior with the rules in Figure 18. The first rule corresponds directly to the first bullet. The second and third rules capture the second and third bullets above. In particular, when the two arguments are functions at different locations that behave identically, both [5eqf] and [5eqproof] apply, indicating that both *#t* and *#f* are legal results.

Since the Report does not specify how to determine whether two procedures are equivalent we leave this open as well, but a natural choice for our setting would be to build on the work of Mason and Talcott (1991) and Felleisen and Hieb (1992).

As a practical matter, our implementation always allows the [5eqproof] reduction but warns those using our implementation by coloring terms in red when they are only reachable by paths that use the [5eqproof] rule. For example, when this program

```
(define f (lambda (x) x))
(define g (lambda (y) y))
(#%eqv? f g)
```

is run in our semantics it produces both *#t* and *#f*, but *#t* is only legal if the procedures (**lambda** (*x*) *x*) and (**lambda** (*y*) *y*) have proven to behave identically. Accordingly, our tool colors the term with *#t* in red. Figure 19 shows a screenshot, starting from the program that the above reduces to, just before the [5eqproof] rule applies.

It is possible to have multiple paths in the reduction graph that lead to the same final answer, but where some paths require equivalence proofs and others do not.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

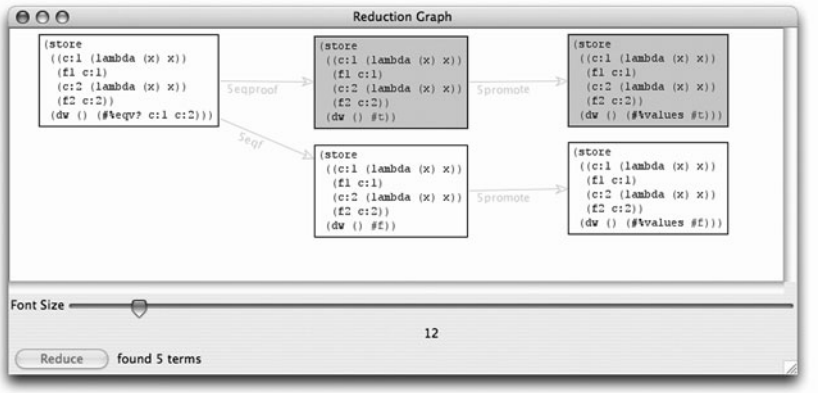

Fig. 19. PLT Redex GUI showing the terms requiring equivalence proofs.

For example, if we replace the final expression above with this one:

$$(\textbf{if}\ \ (\#\%eqv?\ f\ \ g)\ \ (\#\%+\ 2\ 2)\ \ (\#\%*\ 2\ 2))$$

then no matter which way the test of the **if** expression goes, the final answer is always *4*. In situations like that, the tool colors only those intermediate states on paths that require the extra proofs; where the paths converge, it drops the color.

17 Quote and eval

The rules for *#%eval* and **quote** in Figure 20 are essentially the same as the rules for *eval* and **quote** in section 6. The first rule rewrites a non-empty quoted list into a compile-time pair allocation and the second rule rewrites an empty quoted list into *#%null*. The third rule is a generalization of the rule that drops the quote around numbers from Figure 20, but here drops the quote around any self-quoting value. The [5ccons] performs an allocation, replacing a **ccons** expression with a pointer into the store. Finally, the [5eval] reifies its argument, leaving behind new program text to evaluate. Its side condition guarantees that it applies only when the result of reification is a well-formed expression. The $\mathscr{R}$ function is similar to the one in Figure 6; the only difference is that it operates on programs in this semantics that have *#%cons* and *#%null* instead of *cons* and *null*.

18 Errors

Figures 21 and 22 show all of the error reductions for this semantics. Each of the rules in these figures rewrites a program into an error message and discards the context, ensuring that no further reduction can take place. The Report, however, does not specify that any of these errors should be signaled. In fact, it leaves all of these situations completely unspecified. Accordingly, implementations that conform to the Report may signal the errors as shown in the figures, or may do anything else in these situations.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$SP_1[(\textbf{quote}\ (s_1\ s_2\ \cdots))] \rightarrow$ [5qcons]
$SP_1[(\textbf{ccons}\ (\textbf{quote}\ s_1)\ (\textbf{quote}\ (s_2\ \cdots)))]$

$SP_1[(\textbf{quote}\ ())] \rightarrow$ [5qnull]
$SP_1[\#\%null]$

$SP_1[(\textbf{quote}\ (s_1\ s_2\ s_3\ \cdots\ dot\ s_4))] \rightarrow$ [5qconsd]
$SP_1[(\textbf{ccons}\ (\textbf{quote}\ s_1)\ (\textbf{quote}\ (s_2\ s_3\ \cdots\ dot\ s_4)))]$

$SP_1[(\textbf{quote}\ (s_1\ dot\ s_2))] \rightarrow$ [5qdot]
$SP_1[(\textbf{ccons}\ (\textbf{quote}\ s_1)\ (\textbf{quote}\ s_2))]$

$SP_1[(\textbf{quote}\ sqv_1)] \rightarrow$ [5qsqv]
$SP_1[sqv_1]$

$(\textbf{store}\ (sf_1\ \cdots)\ (\textbf{dw}\ (dws_1\ \cdots)\ d_1\ \cdots\ SD_1[(\textbf{ccons}\ v_1\ v_2)]\ s_1\ \cdots)) \rightarrow$ [5ccons]
$(\textbf{store}\ (sf_1\ \cdots\ (pp_1\ (\#\%cons\ v_1\ v_2)))\ (\textbf{dw}\ (dws_1\ \cdots)\ d_1\ \cdots\ SD_1[pp_1]\ s_1\ \cdots))$ (pp_1 fresh)

$(\textbf{store}\ (sf_1\ \cdots)\ W_1[(\#\%eval\ v_1)]) \rightarrow$ [5eval]
$(\textbf{store}\ (sf_1\ \cdots)\ W_1[\mathscr{R}\ [\![(sf_1\ \cdots), v_1\ ]\!]])\quad (\mathscr{R}\ [\![(sf_1\ \cdots), v_1\ ]\!] \in es)$

$\mathscr{R} : (sequence\text{-}of\ sf) \times v \rightarrow s$
$\mathscr{R}\ [\![\ S, \#\%null\ ]\!] = ()$
$\mathscr{R}\ [\![\ S, 'sy\ ]\!] = sy$
$\mathscr{R}\ [\![\ S, pp\ ]\!] = \mathscr{C}\ [\![\ \mathscr{R}\ [\![\ v_a\ ]\!], \mathscr{R}\ [\![\ v_d\ ]\!]\ ]\!]$
$\mathscr{R}\ [\![\ S, v\ ]\!] = v$ (otherwise)

$\mathscr{C} : s \times s \rightarrow s$
$\mathscr{C}\ [\![\ s1, (s2\ \cdots)\ ]\!] = (s1\ s2\ \cdots)$
$\mathscr{C}\ [\![\ s1, s2\ ]\!] = (s1\ \textbf{dot}\ s2)$
(where S binds pp to $(\#\%cons\ v_a\ v_d)$)

Fig. 20. Quote and eval.

The first two rules in Figure 21 cover arithmetic errors. The second rule reports which argument is a nonnumber by counting the number of values in $(v_1 \ldots)$ and then adding one. The next four cover errors for the pair primitives. The rules [5errvar] and [5errset] cover free variable errors. The rule [5appe] covers application of a nonfunction. The rules [5applye] and [5applen] cover abuse of *#%apply*. The [5dwerr] covers bad arguments supplied to *#%dynamic-wind* and [5valerr] signals an error when a single-value context receives too few or too many values. The last two rules signal errors when *#%eval* gets the wrong number of arguments. The Report says that the argument to *eval* "must be a valid Scheme expression" but that "[i]mplementations may extend eval to allow non-expression programs (definitions) as the first argument." Rather than extending our *#%eval* to accept definitions, we merely identify two different errors, one when *#%eval*'s argument is a definition and one when it is ill-formed.

The rules in Figure 22 handle all of the arity error reductions, both for primitive procedures and for user-defined procedures. The *fun1* nonterminal contains all of the primitive procedures of arity one, and *fun2* contains all of the primitive procedures of arity two. The remaining primitives, *#%dynamic-wind*, *#%–*, *#%/*, and *#%apply*, are handled specially.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

$P[(\#\%/\ n_1\ n_2\ n_3\ \cdots)] \rightarrow$ [5/0]
error: divison by zero $(0 \in \{\ n_2, n_3\ \cdots\ \})$

$P[(aop\ v_1\ \cdots\ v_i\ v_{i+1}\ \cdots)] \rightarrow$ [5ae]
error: arith-op applied to non-number, arg $(\#v_1)+1$ (v_i is not a number)

$P_1[(\#\%car\ v_i)] \rightarrow$ [5care]
error: can't take car of non-pair $(v_i \notin pp)$

$P_1[(\#\%cdr\ v_i)] \rightarrow$ [5cdre]
error: can't take cdr of non-pair $(v_i \notin pp)$

$P_1[(\#\%set\text{-}car!\ v_1\ v_2)] \rightarrow$ [5scare]
error: can't set-car! on a non-pair $(v_1 \notin pp)$

$P_1[(\#\%set\text{-}cdr!\ v_1\ v_2)] \rightarrow$ [5scdre]
error: can't set-cdr! on a non-pair $(v_1 \notin pp)$

(**store** $(sf_1\ \cdots)$ (**dw** $(dws\ \cdots)\ D[x_1]\ d\ \cdots)) \rightarrow$ [5errvar]
error: reference to free identifier: x_1 $(x_1 \notin \mathrm{dom}(sf_1\ \cdots))$

(**store** $(sf_1\ \cdots)$ (**dw** $(dws\ \cdots)\ D[(\mathbf{set!}\ x_1\ v)]\ d\ \cdots)) \rightarrow$ [5errset]
error: attempt to set! free identifier: x_1 $(x_1 \notin \mathrm{dom}(sf_1\ \cdots))$

$P[(nonfun\ v\ \cdots)] \rightarrow$ [5appe]
error: can't apply non-function

$P[(\#\%apply\ fun\ v_1\ \cdots\ v_2)] \rightarrow$ [5applye]
error: apply's last argument non-list $(v_2 \notin pp, v_2 \neq \#\%null)$

$P[(\#\%apply\ nonfun\ v\ \cdots)] \rightarrow$ [5applynf]
error: can't apply non-function

$P[(\#\%dynamic\text{-}wind\ v_1\ v_2\ v_3)] \rightarrow$ [5dwerr]
error: dynamic-wind expects arity 0 procs (v_1, v_2, or v_3 does not have arity 0)

$P[(\#\%values\ v_1\ \cdots)]_\circ \rightarrow$ [5valerr]
error: context received wrong # of values $(\#v_1 \neq 1)$

(**store** $(sf_1\ \cdots)\ W[(\#\%eval\ v_1)]) \rightarrow$ [5evale]
error: malformed expression: $\mathscr{R}\ [\![(sf_1\ \cdots), v_1\ ]\!]$ $(\mathscr{R}\ [\![(sf_1\ \cdots), v_1\ ]\!] \notin ds)$

(**store** $(sf_1\ \cdots)\ W[(\#\%eval\ v_1)]) \rightarrow$ [5evald]
error: eval only takes expressions $(\mathscr{R}\ [\![(sf_1\ \cdots), v_1\ ]\!] \in ds, \mathscr{R}\ [\![(sf_1\ \cdots), v_1\ ]\!] \notin es)$

Fig. 21. Nonarity errors.

19 Consistency

In an effort to ensure that our semantics is sensible and defines the language we intend it to define, we have exploited its executable nature to build a test suite for it. The test suite contains 258 test expressions that together explore more than 14,000 distinct program states. The largest test case requires 1317 states and the test case with the most nondeterminism visits 71 different states that each have multiple next states. Each test expression is checked against its expected result (or results), and each intermediate state is checked to be sure it is an element of $\mathcal{P}$.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

(**store** (*sf* ⋯ (*cp*$_i$ (**lambda** (*x*$_1$ ⋯) *e e* ⋯)) *sf* ⋯) [5arity]
  W[(*cp*$_i$ *v*$_1$ ⋯)]) →
error: arity mismatch (#*x*$_1$ ≠ #*v*$_1$)

(**store** (*sf* ⋯ (*mp*$_i$ (**lambda** (*x*$_1$ *x*$_2$ ⋯ **dot** *x*) (*cp x* ⋯))) *sf* ⋯) [5μarity]
  W[(*mp*$_i$ *v*$_1$ ⋯)]) →
error: arity mismatch (#*v*$_1$ < #*x*$_2$ + 1)

P[(*fun1 v*$_1$ ⋯)] → [51arity]
error: arity mismatch (#*v*$_1$ ≠ 1)

P[(*fun2 v*$_1$ ⋯)] → [52arity]
error: arity mismatch (#*v*$_1$ ≠ 2)

P[(#%*dynamic-wind v*$_1$ ⋯)] → [5dwarity]
error: arity mismatch (#*v*$_1$ ≠ 3)

P[(#%−)] → [5-arity]
error: arity mismatch

P[(#%/)] → [5/arity]
error: arity mismatch

P[(#%*apply*)] → [5apparity0]
error: arity mismatch

P[(#%*apply v*)] → [5apparity1]
error: arity mismatch

Fig. 22. Arity errors.

As we worked on the semantics, we would often find that seemingly innocuous changes in one part of the semantics would disrupt other parts. As an example, when we improved the way that unspecified order of evaluation was handled, the rules for *#%call-with-values* broke, without us realizing it until we ran the test suite. Because we had test cases for *#%call-with-values*, however, the problem was quickly identified and fixed. Usually the initial term of the failed test case was enough to help us identify the problem and fix it. When that was not enough, we would first try to make the test case as small as possible and then use PLT Redex to visualize a small graph that had the problem. Understanding the problem via small examples inevitably led to a solution.

The test suite is the source of most of our confidence that this semantics behaves as we expect. But, as a further guarantee that the semantics is sensible, we also prove that well-formed programs cannot get stuck.

Definition 1
A program $\mathcal{P}$ is *well-formed* if every *pp*, *cp*, *mp* that appears in the body of $\mathcal{P}$ is bound in the store.

Theorem 1
For any well-formed program $\mathcal{P}$, either there exists at least one $\mathcal{A}$ such that $\mathcal{P} \rightarrow^* \mathcal{A}$, or for every $\mathcal{P}'$ such that $\mathcal{P} \rightarrow^* \mathcal{P}'$, there exists a well-formed $\mathcal{P}''$ such that $\mathcal{P}' \rightarrow \mathcal{P}''$.

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

Proof
The proof of this theorem is structured like a standard proof of type soundness, but instead of guaranteeing that programs are well-typed, we only guarantee that programs are well-formed. Lemma 1 plays the role of the preservation lemma and lemma 2 plays the role of the progress lemma. Together they establish the theorem. □

Lemma 1
If there exists some well-formed $\mathcal{P}'$ that reduces to $\mathcal{P}$, then $\mathcal{P}$ is well-formed.

Proof
Follows by inspection of the reduction rules. □

Lemma 2
For any well-formed $\mathcal{P}$, at least one of the following is true:

- $\mathcal{P} \rightarrow \mathcal{P}'$
- $\mathcal{P} \rightarrow$ **error:** *str* for some error message *str*, or
- $\mathcal{P} =$ (**store** (*sf* $\cdots$) (**dw** (*dws* $\cdots$) (*#%values v* $\cdots$)))

Note that both the first and the second cases might apply to the same term, due to the way we model unspecified order of evaluation, as discussed in section 4.

Proof
We proceed by cases on the structure of $\mathcal{P}$. First, assume that $\mathcal{P}$ contains some quoted subexpression or some **ccons** subexpression. Inspection of the S contexts shows that, no matter where these expressions might occur, they will reduce either by [5qcons], [5qnull], or [5qsqv] in the case of a quoted expression or by [5ccons] in the case of a **ccons** expression.

To show that the remaining cases of $\mathcal{P}$ satisfy the lemma, we rely on lemma 3. Examination of the top-level program context P and that lemma tells us that an expression that appears at the top level of a well-formed program reduces or is (*#%values v* $\cdots$). The only other cases are top-level definitions and **begin**$^{\mathbf{D}}$ expressions, which are covered by the first five reduction rules in Figure 14 and [5valerr]. □

Lemma 3
Every e is at least one of

- v,
- (*#%values v* $\cdots$),
- $E[v]_{\star}$,
- $E[(\#\%values\ v\ \cdots)]_{\circ}$, or
- $E[i]_{\circ}$

where i is one of the expressions that appears in the context in one of the reduction rules, excluding [5promote], [5demote], or [5valerr].

Proof
This lemma follows directly from a straightforward inductive argument on the structure of e, but is the key lemma in the proof of the theorem. It is complicated by multiple values, but is analogous to a lemma that guarantees that each expression e is either $E[i]$ or v in more standard reduction systems. □

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

20 Conclusions

Our journey into the Report has once again revealed the value of mechanizing a semantics. In addition to improving our own understanding of operational semantics through the process of coaxing a machine to behave as the Report decrees, we have also learned two lessons worth sharing.

First, we learned that the underspecification of the Report goes deep. It is common knowledge that the Report specifies a family of programming languages rather than just a single programming language, since implementations can vary on a myriad of details: the order of evaluation of a function's arguments, the results of particular primitives, which errors to report and which to ignore, etc. There are so many possibilities that writing implementation-independent Scheme code requires tool support (Sitaram 2003). It is less well-known, however, that the Report is actually even less specific than that — it specifies an entire family of semantics, due to the specification of *eqv?*. The Report specifies that, when *eqv?*'s arguments are procedures with different tags, an implementation may produce *#t* if it can prove that procedures behave identically when presented with identical inputs (rule [5eqproof] from Figure 18 in our semantics). Since the truth of this statement is as difficult to prove as the observational equivalence of two phrases and the Report does not suggest a specific proof system, the semantics itself must be parameterized over the possible proof systems to use with this rule.

Second, and perhaps more importantly, we learned the importance of building a test suite of programs and their expected behavior. Not only does a test suite ensure that the semantics models the intended behavior, it also cuts down the number of errors in the semantics. Much of the effort in mechanizing semantics today is focused on automatically generating or verifying proofs of the meta-theory of a semantics. While this effort is also clearly important, such proofs do not substitute for the ability of test suites to ensure that the semantics is modeling what we expect it to model, nor do they substitute for the understanding gained by exploring the behavior of examples. This lesson is what has led us to spend significant efforts on PLT Redex, and we believe its use will benefit the use of other tools for the mechanization of semantic specifications.

Indeed, a test suite might have helped the authors of the Report discover an inconsistency in their specification. In section 6.4, The Report's informal semantics says that "except for continuations created with the *call-with-values* procedure, all continuations take exactly one value." The formal semantics does not enforce this restriction on expressions evaluated for their effects. These expressions are evaluated by $\mathcal{C}$, which does not use *single*; in contrast, $\mathcal{E}^*$ also evaluates sequences of expressions but does use *single*. For that reason, the two definitions of Scheme's **begin** given in Section 7.3 of the Report are subtly incompatible with each other. The first requires (**begin** (*values*) *1*) to evaluate to the implementation-specific result of the *wrong* metafunction, which means that an implementation should signal an error. The second requires it to evaluate to 1.[7]

[7] Although the Report does not explicitly specify whether the *wrong* metafunction corresponds to the "is an error" situation or the "signals an error" situation in the terminology of the informal semantics,

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

Beyond test suites, the mechanization of the semantics also makes calculations simple. As an example, the Report defines *values* using *call/cc*, but our semantics models *values* directly (as *#%values*). The definition in the Report is

```
(define values
  (lambda things
    (call/cc
      (lambda (cont)
        (apply cont things)))))
```

To prove that this definition and the *#%values* in our semantics are observably equivalent, one needs a standard lemma about contexts (Felleisen *et al.*, 1987; Mason & Talcott 1991) to conclude that the only interesting case is when the two versions of *values* appear in an application context. So, we plugged the lambda expression above into PLT Redex in an application context and voilà: 77 steps later, out popped *#%values* in that same application context.

Overall, we hope that our experience leads others to provide good support for experimenting with examples and maintaining test suites.

Acknowledgments

Thanks to Kent Dybvig and Matthew Flatt for helpful discussions and tips concerning the inner workings of Chez Scheme (Dybvig 2005) and MzScheme (Flatt 2006). Thanks to Mike Sperber for pointing out a flaw in our dynamic-wind semantics and providing us with an example that demonstrated the problem, in addition to other helpful comments about the draft. Thanks to Will Clinger for enlightening discussions of semantics, Scheme, the Report in general, and the Report's specification of *eqv?* in particular. Thanks also to John Reppy, Dave MacQueen, Matthias Felleisen, and the anonymous reviewers from ICFP 2005, the Scheme and Functional Programming Workshop 2005, and JFP for their comments on earlier versions of this work.

References

Clinger, W. D. (1998, June) Proper tail recursion and space efficiency. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pp 66–67.

Clinger, W. D. & Hansen, L. T. (1994) Lambda, the ultimate label, or a simple optimizing compiler for Scheme. *ACM symposium on Lisp and Functional Programming*. SIGPLAN Lisp Pointers 7(3) (July–September 1994). Available at: `http://www.ccs.neu.edu/home/will/Larceny/`. Accessed date: July 6, 2007.

Dybvig, K., Clinger, W., Flatt, M., Sperber, M. & van Straaten, A. (2006, June). The R6RS status report. Available at: `http://www.schemers.org/Documents/Standards/Charter/`. Accessed date: July 6, 2007.

the fact that it does not accept a continuation argument means it must actually be signaling an error. The inconsistency between the two specifications of **begin** has been resolved in the working draft of the Revised[6] Report on Scheme, which states that intermediate expressions in **begin** statements should be allowed to return multiple values (Dybvig *et al.* 2006).

https://doi.org/10.1017/S0956796807006478 Published online by Cambridge University Press

Dybvig, R. K. (2005) Chez Scheme version 7 user's guide. Cadence Research Systems. Available at: http://www.scheme.com/. Accessed date: July 6, 2007.

Feeley, M. (2006) Gambit-C, version 4.0 beta 17. Available at: http://www.iro.umontreal.ca/~gambit/. Accessed date: July 6, 2007.

Felleisen, M. (1987) *The Calculi of lambda-v-cs Conversion: A Syntactic Theory of Control and State in Imperative Higher-Order Programming Languages*. Ph.D. thesis, Indiana University.

Felleisen, M. (1988) Lambda-v-CS: and extended lambda-calculus for Scheme. In *Proceedings of the Conference on LISP and Functional Programming*.

Felleisen, M. & Flatt, M. (2006) Programming languages and lambda calculi. Unpublished manuscript. Available at: http://www.cs.utah.edu/plt/publications/pllc.pdf. Accessed date: July 6, 2007.

Felleisen, M. & Hieb, R. (1992) The revised report on the syntactic theories of sequential control and state. *Theor. Comput. Sci.*, **102**, 235–271. Original version in: Technical Report 89-100, Rice University, June 1989.

Felleisen, M., Friedman, D. P., Kohlbecker, E. & Duba, B. (1987) A syntactic theory of sequential control. *Theor. Comput. Sci.*, **52**(3) 205–237.

Flanagan, C. & Felleisen, M. (1999) The semantics of future and an application. *J. Funct. Program.* **9**, 1–31.

Flatt, M. (2006). *PLT MzScheme: Language manual*. Technical Report PLT-TR2006-1-v352. PLT Scheme Inc. Available at: http://www.plt-scheme.org/techreports/. Accessed date: July 6, 2007.

Flatt, M., Krishnamurthi, S. & Felleisen, M. (1999) A programmer's reduction semantics for classes and mixins. *Formal Syntax Semant. Java*, **1523**, 241–269. Preliminary version appeared in Proceedings of *Principles of Programming Languages*, 1998. Revised version is Rice University technical report TR 97-293, June 1999.

Gasbichler, M., Knauel, E., Sperber, M. & Kelsey, R. A. (2003) How to add threads to a sequential language without getting tangled up. In *Workshop on Scheme and Functional Programming*.

GNU. (2006). *MIT/GNU Scheme 7.7.90+*. Available at: http://www.gnu.org/software/mit-scheme/. Accessed date: July 6, 2007.

Harper, R. & Lillibridge, M. (1993) Explicit polymorphism and CPS conversion. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Harper, R. & Stone, C. (1996) *A Type-Theoretic Account of Standard ml 1996 (version 2)*. Tech. rept. CMU-CS-96-136R. School of Computer Science, Carnegie Mellon University.

Haynes, C. T. & Friedman, D. P. (1987) Embedding continuations in procedural objects. In *ACM Transactions on Programming Languages and Systems*, **9**(4), 582–598.

Herman, D. & Meunier, P. (2004) Improving the static analysis of embedded languages via partial evaluation. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*. New York: ACM Press, pp. 16–27.

Kelsey, R., Rees, J. & Sperber, M. (2005) *Scheme 48*. Available at: http://s48.org/. Accessed date: July 6, 2007.

Kelsey, R., Clinger, W. & (Editors), Jonathan R. (1998) Revised^5 report of the algorithmic language Scheme. *ACM SIGPLAN Notices*, **33**(9), 26–76.

Lee, D. K., Crary, K. & Harper, R. (2006) *Mechanizing the Metatheory of Standard ML*. Tech. rept. CMU-CS-06-138. Carnegie Mellon University. Available at: http://www.cs.cmu.edu/~crary/papers/2006/tslf.pdf. Accessed date: July 6, 2007.

Mason, I. & Talcott, C. (1991) Equivalence in functional languages with effects. *J. Funct. Program.* **1**(July), 287–327.

Matthews, J. (2005) *Operational semantics for Scheme via term rewriting*. Tech. rept. TR-2005-02. University of Chicago.

Matthews, J. & Findler, R. B. (2005) An operational semantics for R5RS Scheme. In *Workshop on Scheme and Functional Programming*.

Matthews, J. & Findler, R. B. (2007) Operational semantics for multi-language programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*.

Matthews, J., Finder, R. B., Flatt, M. & Felleisen, M. (2004) A visual environment for developing context-sensitive term rewriting systems. In *Proceedings of the International Conference on Rewriting Techniques and Applications (RTA)*.

Miller, S. G. & Radestock, M. (2006) *SISC for seasoned schemers*. Available at: `http://sisc.sourceforge.net/`. Accessed date: July 6, 2007.

Muller, R. (1992) M-LISP: A representation-independent dialect of LISP with reduction semantics. *ACM Transact. Program. Lang. Syst.* **14**(4).

Neubauer, M. & Sperber, M. (2001) Down with Emacs Lisp: Dynamic scope analysis. In *ACM SIGPLAN International Conference on Functional Programming (ICFP)*.

Oliva, D. P., Ramsdell, J. D. & Wand, M. (1995) The VLISP verified prescheme compiler. *Lisp and Symbol. Comput.* **8**(1/2).

Project GNU. (2005) Guile reference manual. Available at: `http://www.gnu.org/software/guile/`. Accessed date: July 6, 2007.

Ramsdell, J. D. (1992) An operational semantics for Scheme. *Lisp Point.* **2**(April–June).

Reppy, J. (1999) *Concurrent Programming in ML*. Cambridge University Press.

Serrano, M. (2006) *Bigloo: A Practical Scheme compiler*. Available at: `http://www-sop.inria.fr/mimosa/fp/Bigloo/`. Accessed date: July 6, 2007.

Sitaram, D. (2003) Porting Scheme programs. In *Scheme and Functional Programming Workshop*.

Sussman, G. J. & Guy, L. S., Jr. (1975) Scheme: An interpreter for extended lambda calculus. Tech. rept. AI Lab Memo AIM-349. MIT AI Lab.

van Straaten, A. (2002) An executable denotational semantics for scheme. Available at: `http://www.appsolutions.com/SchemeDS`. Accessed date: July 6, 2007.

Winkelmann, F. L. (2006) Chicken: A practical and portable scheme system. Available at: `http://www.call-with-current-continuation.org/`. Accessed date: July 6, 2007.

Wright, A. & Felleisen, M. (1994) A syntactic approach to type soundness. *Inform. Comput.* 38–94. First appeared as Technical Report TR160, Rice University, 1991.