Massachusetts Institute of Technology
6.844, Spring ’03: Computability Theory of and with Scheme
Prof. Albert Meyer

Course Notes 3
March 6
revised May 13, 2003, 1203 minutes

A Substitution Model for Scheme

1 Introduction

These notes describe a Scheme Substitution Model: an accurate, simple mathematical model of Scheme evaluation based on rules for rewriting one Scheme expression into another. The model captures a significant portion of Scheme, including asssignment (`set!`) and control abstraction (`call/cc`).

We assume the reader already has an understanding of Scheme at the level taught in an introductory Scheme programming course. In particular, we assume the concepts of free and bound variables, and the scoping rules for `lambda` and `letrec`, are understood.

The rules of the game in a Substitution Model are that the only objects manipulated in the Model are Scheme expressions: no separate data structures for environments, `cons`-cells, or continuations. Evaluation of an expression, M, is modelled by successive application of rewrite rules starting with M. Each rule transforms an expression into a new Scheme expression. Rewriting continues until an expression is reached for which no rule is applicable. This final expression, if any, gives a direct representation either of the value returned by the expression, or of the kind of dynamic error that first occurs in the evaluation.

At most one rule is applicable to each expression, reflecting the deterministic character of Scheme evaluation. The way an expression rewrites is determined solely by the expression, not the sequence of prior rewrites that may have led to it. If M' is an expression reached at any point by rewriting starting at M, then evaluating M' in Scheme’s initial environment will result in the same final value, the same kind of error, or the same “runaway” behavior (divergence) as evaluation of M.

The environment in which an expression is to be evaluated will be represented by surrounding an expression with an outermost `letrec` that binds variables in the environment to the expressions representing their values. Immutable lists and pairs are represented as combinations with operators `list` or `cons`.

Mutable lists do not fit well into a Substitution Model. They could be shoe-horned in, but we haven’t found a tasteful way to do it. The problem is that we haven’t found a reasonable class of Scheme expressions that evaluate directly to circular list structures and that could serve as canonical forms for these structures. So we have omitted mutable lists from this Substitution Model; vectors are omitted for similar reasons. We have omitted characters altogether, ensuring that strings are immutable.

Side-effects involving input/output also don’t fit and have been omitted. So procedures such as `set-car!`, `string-set!`, `display`, or `read` are not included in the Model.

2 Control Syntax for the Substitution Model

A simplified Backus-Naur Form (BNF) grammar for Scheme is given in an Appendix. According to the official Scheme specification, the standard builtin operators such as `+`, `symbol?`, `cons`, `apply` are variables that can be reassigned. This is a regrettable design decision, because with few exceptions, it is a really bad idea for a programmer to redefine the builtins. In this Substitution Model, these identifiers are treated as *constants* rather than variables.

The Substitution Model requires some additional syntactic concepts, namely, *syntactic values* and *control contexts*.

2.1 Syntactic Values

Scheme's values are numbers, Booleans, symbols, and similar atomic types; procedures; and lists and pairs of values. Each value will be represented by a canonical expression called a *syntactic value*. In particular, compound procedures are represented as `lambda`-expressions. Here is the grammar[1]:

$$\begin{array}{rcl}
\langle\text{syntactic-value}\rangle & ::= & \langle\text{immediate-value}\rangle \mid \langle\text{list-value}\rangle \mid \langle\text{pair-value}\rangle \\
\langle\text{immediate-value}\rangle & ::= & \langle\text{self-evaluating}\rangle \mid \langle\text{symbol}\rangle \mid \langle\text{procedure}\rangle \\
\langle\text{list-value}\rangle & ::= & \texttt{(list}\ \ \langle\text{syntactic-value}\rangle^{*}\texttt{)} \\
\langle\text{pair-value}\rangle & ::= & \texttt{(cons}\ \ \langle\text{syntactic-value}\rangle\,\langle\text{immediate-value}\rangle\texttt{)} \\
 & & \mid \texttt{(cons}\ \ \langle\text{syntactic-value}\rangle\,\langle\text{pair-value}\rangle\texttt{)}
\end{array}$$

Note that syntactic values may contain `letrec`'s only within procedure bodies.

2.2 Control Contexts for Kernel Scheme

An important technical property of the Substitution Model is that the rewrite rule to apply at any evaluation step wiil be uniquely determined. The order in which subexpressions are evaluated is formalized in terms of *control contexts*. A control context is an expression with a "hole," [ ], indicating the subexpression that an evaluator would begin working on. We'll illustrate with an example before giving the formal definitions.

Definition 2.1. If R is an expression with a hole, and M is an expression, we write $R[M]$ to denote the result of replacing the hole in R by M *without any renaming* of bound variables.

Example 2.2. Let

$$M = \texttt{(+ 1 (if (pair? (list (list) 'a)) 2 3) (* 4 5))}.$$

[1]In these grammars, superscript "*" indicates zero or more occurrences of a grammatical phrase, and superscript "$^+$" indicates one or more occurrences.

M is a combination, and not all the operands are values, so Scheme would start to evaluate one of the operands. Using left-to-right evaluation, the operand

```
(if (pair? (list (list) 'a)) 2 3)
```

would be the one to start evaluating, since `+` and `1` represent final values. This corresponds to parsing M as

$$R_1[\texttt{(if (pair? (list (list) 'a)) 2 3)}]$$

where R_1 is the control context

$$R_1 ::= \texttt{(+ 1 } [\ ] \texttt{ (* 4 5))}.$$

The test of this `if` expression is a combination

$$P ::= \texttt{(pair? (list (list) 'a))}$$

whose operator and operand are values, so next, Scheme would actually apply the operator `pair?` to the operand `(list (list) 'a)`. This corresponds parsing M as $R[P]$ where

$$R ::= \texttt{(+ 1 (if } [\ ] \texttt{ 2 3) (* 4 5))}.$$

The fact that Scheme will apply the operator `pair?` is captured by the fact that P is an *immediate redex*.

The fact that the rewrite rule to apply at any evaluation step is uniquely determined follows from the fact that every nonvalue Scheme expression parses in a unique way as a control context with an immediate redex in its hole.

Formally, we specify control contexts and immediate redexes by the following grammars:

$$
\begin{array}{rcl}
\langle\text{control-context}\rangle & ::= & \langle\text{hole}\rangle \\
 & & \mid \texttt{(if } \langle\text{control-context}\rangle\ \langle\text{expression}\rangle\ \langle\text{expression}\rangle\texttt{)} \\
 & & \mid \texttt{(begin } \langle\text{control-context}\rangle\ \langle\text{expression}\rangle^{+}\texttt{)} \\
 & & \mid \texttt{(set! } \langle\text{variable}\rangle\ \langle\text{control-context}\rangle\texttt{)} \\
 & & \mid \texttt{(}\langle\text{let-keyword}\rangle \\
 & & \quad\quad \texttt{(}\langle\text{value-binding}\rangle^{*}\ \texttt{(}\langle\text{variable}\rangle\ \langle\text{control-context}\rangle\texttt{)}\ \langle\text{binding}\rangle^{*}\texttt{)} \\
 & & \quad\quad\quad \langle\text{expression}\rangle\texttt{)} \\
 & & \mid \texttt{(}\langle\text{syntactic-value}\rangle^{*}\ \langle\text{control-context}\rangle\ \langle\text{expression}\rangle^{*}\texttt{)} \\
\langle\text{hole}\rangle & ::= & [\,] \\
\langle\text{value-binding}\rangle & ::= & \texttt{(}\langle\text{variable}\rangle\ \langle\text{syntactic-value}\rangle\texttt{)}
\end{array}
$$

Note that `letrec`'s all of whose ⟨init⟩'s are syntactic values may only appear in control contexts when they are within procedure bodies.

$$
\begin{array}{lll}
\langle\text{immediate-redex}\rangle & ::= & \langle\text{variable}\rangle \\
 & & \mid \texttt{(if } \langle\text{syntactic-value}\rangle\ \langle\text{expression}\rangle\ \langle\text{expression}\rangle\texttt{)} \\
 & & \mid \texttt{(}\langle\text{let-keyword}\rangle\ \texttt{(}\langle\text{value-binding}\rangle^*\texttt{)}\ \langle\text{expression}\rangle\texttt{)} \\
 & & \mid \texttt{(}\langle\text{nonpairing-procedure}\rangle\ \langle\text{syntactic-value}\rangle^*\texttt{)} \\
 & & \mid \texttt{(begin } \langle\text{expression}\rangle\texttt{)} \\
 & & \mid \texttt{(begin } \langle\text{syntactic-value}\rangle\ \langle\text{expression}\rangle^*\texttt{)} \\
 & & \mid \texttt{(set! } \langle\text{variable}\rangle\ \langle\text{syntactic-value}\rangle\texttt{)}
\end{array}
$$

Definition 2.3. An outermost `letrec` binding variables to values—used to model a Scheme environment—is called the *environment* `letrec`. An expression is said to be in *environment form* when it has an environment `letrec`, namely, it is of the form

$$\texttt{(letrec (}\langle\text{value-binding}\rangle^*\texttt{)}\ N\texttt{)}$$

for some ⟨expression⟩, N. We use $\text{Env}(N)$ as an abbreviation for this form.

Definition 2.4. Let M be a Scheme expression, R be a control context, and P be an immediate redex. Then M is said to *control-parse into R and P* iff either

1. M is of the form $\text{Env}(R[P])$, or
2. $M = R[P]$ for $R \neq \langle\text{hole}\rangle$, or
3. $M = P$, $R = \langle\text{hole}\rangle$, and P is not in environment form .

Definition 2.4.3 reflects that fact that a non-outermost `letrec` binding of variables to values will be the redex of a rule to incorporate the bindings into the outermost environment `letrec`. On the other hand, we do not want to parse the environment `letrec` in this way. For example, consider the expression

$$\texttt{(letrec ((x 1)) (letrec ((y 2)) (+ x y)))}. \qquad (1)$$

Expression (1) control parses into

$$
\begin{aligned}
R &= \texttt{(letrec ((x 1))}\ [\ ]\texttt{)},\\
P &= \texttt{(letrec ((y 2)) (+ x y))}.
\end{aligned}
$$

On the other hand, (1) is itself is an ⟨immediate-redex⟩ according to the grammatical rules, so it could also be parsed as $R'[P']$, where $R' = \langle\text{hole}\rangle$, and P' is (1) itself. However, Definition 2.4.3 disallows this second parse as a control parse because P' is in environment form.

Lemma 2.5. ***(Unique Control Parsing)*** *If a Scheme expression, M, is not a syntactic value, then there is a unique control context, R, and immediate redex, P, such that $M = R[P]$. If M is a syntactic value, then it is not control-parsable.*

Proof. By structural induction on M. If M is:

- [⟨self-evaluating⟩, ⟨symbol⟩, or ⟨procedure⟩] In this case M is a ⟨syntactic-value⟩, and we must show that it cannot be parsed as $R[P]$. But it follows immediately from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, that the only control context R such that $M = R[N]$ for some N, is $R = [\,]$, in which case $N = M$. But since $N = M$ is an ⟨syntactic-value⟩, it is not an ⟨immediate-redex⟩, proving that M is not control-parsable.
- [a variable] In this case $R =$ ⟨hole⟩ and $P = M$.
- [a combination] Then if:
  - [the operator or some operand is not a ⟨syntactic-value⟩] Then M is of the form

    $$\texttt{(}V_0 \ldots\ N\ M_0 \ldots\texttt{)}$$

    where $V_0 \ldots$ is a (possibly empty) sequence of syntactic values, N is not a syntactic value, and $M_0 \ldots$ is a (possibly empty) sequence of expressions. In this case, M is neither a syntactic value nor an immediate redex. Now it follows immediately from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, that any control context R such that $M = R[M']$ for some M', must be of the form $R = \texttt{(}V_0 \ldots\ R'\ M_0 \ldots\texttt{)}$ for some ⟨control-context⟩, R' such that $R'[M'] = N$. But by induction, $N = R'[P]$ for a unique ⟨control-context⟩, R', and ⟨immediate-redex⟩, P. Hence, $M = R[P]$ for these uniquely determined R and P.
  - [the operator and all operands are ⟨syntactic-value⟩'s] Then M is of the form $\texttt{(}\mathit{op}\ V_0 \ldots\texttt{)}$ where $V_0 \ldots$ is a (possibly empty) sequence of syntactic values and *op* is a syntactic value. By induction, there is no ⟨control-context⟩, R' such that $R'[P] = \mathit{op}$ or $R'[P] = V_i$ for some immediate redex P and operand V_i. Now it follows immediately from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, that the only control context, R, such that $M = R[P]$ for some immediate redex, P, must be with $R =$ ⟨hole⟩ and $P = M$. If *op* is not a ⟨pairing-operator⟩, then $M = P$ is an immediate redex, and R and P are uniquely determined as required. On the other hand, if *op* is a ⟨pairing-operator⟩, then M is a value, not an immediate redex, so M is not control-parsable, as required in this case.
- [etc.] The remaining cases are similar.

□

Problem 1. Verify that if R_1 and R_2 are control contexts, then so is $R_1[R_2]$.

3 Scheme Rewrite Rules

This section contains all the rewrite rules necessary to specify the evaluation of kernel Scheme expressions.

We will not consider rewrite rules for the derived expressions. The Revised^5 Scheme Manual describes how to translate ("desugar") expressions using derived syntax into Kernel Scheme. These translations can easily be described with rewrite rules, and these desugaring rules could be incorporated directly into a Substitution Model. The desugaring rules raise no new issues beyond those we consider for the kernel rules, so we have omitted them.

In the following sections, R denotes a control context, B denotes a sequence of zero or more ⟨value-binding⟩'s of distinct variables, V denotes a syntactic value, $V_1 \ldots$ a sequence of one or more syntactic values, and x denotes a variable.

3.1 Simple Control Rules

Definition 3.1. A *simple control rule* is a rewrite rule of the form

$$R[P] \rightarrow R[T]$$

or

$$\texttt{(letrec (}B\texttt{)}\ R[P]\texttt{)} \rightarrow \texttt{(letrec (}B\texttt{)}\ R[T]\texttt{)}.$$

Such a pair of simple control rules may be abbreviated as $P \rightarrow T$, showing only the subexpressions that are changed by the rule. In this case, P will be an immediate redex and is called the *immediate redex of the rule*, and T is called the *immediate contractum*.

3.1.1 Rules for Kernel Scheme

The redexes and contracta for the kernel Scheme simple control rules are:

- `if`:

$$\texttt{(if \#f}\ M\ N\texttt{)} \rightarrow N$$
$$\texttt{(if}\ V\ M\ N\texttt{)} \rightarrow M, \qquad \text{for } V \neq \texttt{\#f}$$

- `lambda` *no args:*

$$\texttt{((lambda ()}\ M\texttt{))} \rightarrow M$$

- `begin`:

$$\texttt{(begin}\ M\texttt{)} \rightarrow M$$
$$\texttt{(begin}\ V\ \langle\text{expression}\rangle^{+}\texttt{)} \rightarrow \texttt{(begin}\ \langle\text{expression}\rangle^{+}\texttt{)}$$

- `procedure?`:

$$\texttt{(procedure?}\ V\texttt{)} \rightarrow \texttt{\#t}, \qquad \text{for } V \text{ a } \langle\text{procedure}\rangle$$
$$\texttt{(procedure?}\ V\texttt{)} \rightarrow \texttt{\#f}, \qquad \text{for other } V.$$

- *builtin operations:*

$$\begin{aligned}
\texttt{(+ 2 3)} &\rightarrow \texttt{5}\\
\texttt{(string-append "ab" "cde")} &\rightarrow \texttt{"abcde"}\\
\texttt{(boolean? "ab")} &\rightarrow \texttt{\#f}\\
&\vdots
\end{aligned}$$

- *symbols:*

$$\begin{aligned}
\texttt{(symbol? (quote } S\texttt{))} &\rightarrow \texttt{\#t}\\
\texttt{(symbol? } V\texttt{)} &\rightarrow \texttt{\#f}, \quad \text{if } V \text{ is not } \texttt{(quote } S\texttt{)}\\
\texttt{(eq? (quote } S\texttt{) (quote } S\texttt{))} &\rightarrow \texttt{\#t}\\
\texttt{(eq? (quote } S_1\texttt{) (quote } S_2\texttt{))} &\rightarrow \texttt{\#f}, \quad \text{if } S_1 \neq S_2,
\end{aligned}$$

where S is an $\langle$identifier$\rangle$.

- *lists:*

$$\begin{aligned}
\texttt{(cons } V \ \langle\text{nil}\rangle\texttt{)} &\rightarrow \texttt{(list } V\texttt{)}\\
\texttt{(cons } V \texttt{ (list } V_1 \ldots\texttt{))} &\rightarrow \texttt{(list } V \ V_1 \ldots\texttt{)}\\
\texttt{(car (list } V_1 \ldots\texttt{))} &\rightarrow V_1\\
\texttt{(cdr (list } V_1 \ V_2 \ \ldots\texttt{))} &\rightarrow \texttt{(list } V_2 \ldots\texttt{)}\\
\texttt{(null? } \langle\text{nil}\rangle\texttt{)} &\rightarrow \texttt{\#t}\\
\texttt{(null? } V\texttt{)} &\rightarrow \texttt{\#f}, \quad \text{if } V \neq \langle\text{nil}\rangle\\
\texttt{(pair? (list } \langle\text{syntactic-value}\rangle^{+}\texttt{))} &\rightarrow \texttt{\#t}\\
\texttt{(apply } V \ \langle\text{nil}\rangle\texttt{)} &\rightarrow \texttt{(}V\texttt{)}\\
\texttt{(apply } V \texttt{ (list } V_1 \ldots\texttt{))} &\rightarrow \texttt{(}V \ V_1 \ldots\texttt{)}
\end{aligned}$$

- *pairs:*

$$\begin{aligned}
\texttt{(pair? (cons } V_1 \ V_2\texttt{))} &\rightarrow \texttt{\#t}\\
\texttt{(pair? } V\texttt{)} &\rightarrow \texttt{\#f},\\
&\quad V \text{ not } \texttt{(list } \langle\text{syntactic-value}\rangle^{+}\texttt{)} \text{ or } \texttt{(cons } \ldots \texttt{)}\\
\texttt{(car (cons } V_1 \ V_2\texttt{))} &\rightarrow V_1\\
\texttt{(cdr (cons } V_1 \ V_2\texttt{))} &\rightarrow V_2
\end{aligned}$$

3.2 Environment Rules

The following rules for Kernel Scheme update the environment `letrec`.

- `lambda` *bind an arg:*

$$\texttt{(letrec (}B\texttt{)}\ R[\texttt{((lambda (}x_1\ \ldots\texttt{)}\ M\texttt{)}\ V_1\cdots\texttt{)}]\texttt{)}$$
$$\to\ \texttt{(letrec (}B\ \texttt{(}x_1\,V_1\texttt{))}\ R[\texttt{((lambda (}\ldots\texttt{)}\ M\texttt{)}\ \cdots\texttt{)}]\texttt{)},$$

$$R[\texttt{((lambda (}x_1\ \ldots\texttt{)}\ M\texttt{)}\ V_1\cdots\texttt{)}]$$
$$\to\ \texttt{(letrec ((}x_1\,V_1\texttt{))}\ R[\texttt{((lambda (}\ldots\texttt{)}\ M\texttt{)}\ \cdots\texttt{)}]\texttt{)}.$$

$$\texttt{(letrec (}B\texttt{)}\ R[\texttt{((lambda}\ x\ M\texttt{)}\ V_1\ldots\texttt{)}]\texttt{)}$$
$$\to\ \texttt{(letrec (}B\ \texttt{(}x\ \texttt{(list}\ V_1\ldots\texttt{)))}\ R[M]\texttt{)},$$

$$R[\texttt{((lambda}\ x\ M\texttt{)}\ V_1\ldots\texttt{)}]$$
$$\to\ \texttt{(letrec ((}x\ \texttt{(list}\ V_1\ldots\texttt{)))}\ R[M]\texttt{)}.$$

- *nested* `letrec`*:*

$$R[\texttt{(letrec (}B_2\texttt{)}\ M\texttt{)}]\ \to\ \texttt{(letrec (}B_2\texttt{)}\ R[M]\texttt{)}\qquad \text{for } R \neq \langle\text{hole}\rangle,$$
$$\texttt{(letrec (}B_1\texttt{)}\ R[\texttt{(letrec (}B_2\texttt{)}\ M\texttt{)}]\texttt{)}\ \to\ \texttt{(letrec (}B_1\,B_2\texttt{)}\ R[M]\texttt{)}.$$

- *instantiation:*

$$\texttt{(letrec (}B_1\ \texttt{(}x\,V\texttt{)}\ B_2\texttt{)}\ R[x]\texttt{)}\ \to\ \texttt{(letrec (}B_1\ \texttt{(}x\,V\texttt{)}\ B_2\texttt{)}\ R[V]\texttt{)}$$

- *assignment:*

$$\texttt{(letrec (}B_1\ \texttt{(}x\,V_1\texttt{)}\ B_2\texttt{)}\ R[\texttt{(set!}\ x\,V_2\texttt{)}]\texttt{)}$$
$$\to\ \texttt{(letrec (}B_1\ \texttt{(}x\,V_2\texttt{)}\ B_2\texttt{)}\ R[\texttt{(quote set!-done)}]\texttt{)}$$

3.3 Unique Rewriting

Definition 3.2. For Scheme expressions M, N, we write $M \to N$ to indicate that M rewrites to N by *one application* of a Scheme Substitution Model rewrite rule. $M \not\to$ means that no rewrite rule applies to M.

From the form of the Substitution Model rewrite rules and the Unique Control Parsing Lemma 2.5, we can straightforwardly conclude:

Corollary 3.3. *(**Unique Rewriting**) There is at most one Scheme Substitution Model rewrite rule whose pattern matches an expression M, and if there is such a rewrite rule, its match is unique. Hence, for every Scheme expression, M, there is at most one N such that $M \to N$.*

Scheme's evaluation behavior is "sequential." Namely, if in the process of evaluation, a subexpression starts to be evaluated, then evaluation continues "at that subexpression" until a value for the subexpression is returned. This happens regardless of how evaluation would proceed once a value is returned. In particular, no Scheme evaluation would switch back and forth between disjoint subexpressions to evaluate them in parallel. The Control-context Independence Corollary makes this precise.

Corollary 3.4. *(Control-context Independence)*

1. *If M_2 is* not *in environment form then*

$$M_1 \to M_2 \quad \textit{implies} \quad R[M_1] \to R[M_2].$$

2. *If*

$$\texttt{(letrec (}B_1\texttt{)}\ M_1\texttt{)} \to \texttt{(letrec (}B_2\texttt{)}\ M_2\texttt{)}$$

then

$$\texttt{(letrec (}B_1\texttt{)}\ R[M_1]\texttt{)} \to \texttt{(letrec (}B_2\texttt{)}\ R[M_2]\texttt{)}.$$

3. *If M_1 is not in environment form and*

$$M_1 \to \texttt{(letrec (}B\texttt{)}\ M_2\texttt{)},$$

then

$$R[M_1] \xrightarrow{\leq 2} \texttt{(letrec (}B\texttt{)}\ R[M_2]\texttt{)}$$

where $\xrightarrow{\leq 2}$ indicates successive application of at most two rewriting rules.

An example of Corollary 3.4.3 where two rule applications are needed is when

$$\begin{aligned} M_1 &= \texttt{(letrec ((x 1)) (+ x x))}, \\ R &= \texttt{(- [ ])}. \end{aligned}$$

In this case, for

$$\begin{aligned} B &= \texttt{(x 1)}, \\ M_2 &= \texttt{(+ 1 x)}, \end{aligned}$$

we have $M_1 \to \texttt{(letrec (}B\texttt{)}\ M_2\texttt{)}$, but it takes 2 steps to rewrite $R[M_1]$ to the desired form:

$$\begin{aligned} R[M_1] &= \texttt{(- (letrec ((x 1)) (+ x x)))} \\ &\to \texttt{(letrec ((x 1)) (- (+ x x)))} \qquad \text{(by the nested \texttt{letrec} rule)} \\ &\to \texttt{(letrec ((x 1)) (- (+ 1 x)))} \\ &= \texttt{(letrec (}B\texttt{)}\ R[M_2]\texttt{)}. \end{aligned}$$

Problem 2. Prove Corollary 3.4. *Hint:* Use Problem 1 and Unique Control Parsing.

4 The Variable Convention

In this and subsequent sections, we restrict ourselves to expressions from Kernel Scheme.

Definition 4.1. A Scheme expression satisfies the *Variable Convention* iff no variable identifier is bound more than once, and no identifier has both by bound and free occurrences.

Problem 3. A Substitution Model rewrite rule *preserves the Variable Convention*, if when M satisfies the Variable Convention and rewrites to N by one application of the rule, then N also satisfies it. Most of the rules preserve the Variable Convention; which do not?

Note that if $M \to N$ but M does not satisfy the Variable Convention, then N may not be a well-formed Scheme expression because the same variable may have two bindings in the outermost, "environment," `letrec` of N. For example,

Example 4.2.

```
(letrec ((x 1)) ((lambda (x) x) 2))
  →  (letrec ((x 1) (x 2)) ((lambda () x)))
```

So we want to ensure that each expression satisfies the Variable Convention before application of a rewriting rule.

It is possible to choose "fresh" names for the bound variables in any Scheme expression and thereby obtain an expression satisfying the Variable Convention. The new expression is equivalent to the original one *up to renaming*. (The Scheme Substitution Model implementation on the course web page has a procedure `enforce` that performs such a renaming.) For historical reasons, this equivalence up to renaming is called *α-equivalence*. So to ensure that the Substitution Model rewriting rules correctly model Scheme behavior, we henceforth assume that the Variable Convention will, if necessary, be enforced on Scheme expressions before they are rewritten by a Substitution Model rule. A consequence of this assumption is that rewritten expressions are no longer determined uniquely; instead, they are only determined up to α-equivalence.

To give a precise definition of α-equivalence we first have to define the notion of substitution for a variable in a Scheme expression. Because Scheme has binding constructs, simple substitution as we defined it for arithmetic expressions will not do. First, when we substitute N for a variable x in M, in symbols $M[x := N]$, we want to replace by N only the "free"occurrences of x in M. Second, we don't want any free variables in N to be "accidentally" bound because they happen to fall within the scope of a binding construct in M.

To avoid this, we may have to rename some bound variables of M to "fresh" variables. A variable is "fresh" with respect to a given finite set of expressions if it does not occur in any of the expressions. There are many ways to find such fresh variables, and there is no need to go into the details of a specific method for finding them.

It turns out that to define the substitution of a term for a variable, we have to define the more general notion of a *simultaneous substitution*, $M[\mathbf{x}_k := \mathbf{N}_k]$, of a sequence $\mathbf{N}_k$ of expressions $N_1, \ldots, N_k$ for a sequence, $\mathbf{x}_k$ of distinct variables $x_1, \ldots, x_k$ in M. The definition is by induction on the structure of M:

Definition 4.3. $M[\mathbf{x}_k := \mathbf{N}_k] ::= P$, where, if M is

- $[x_i]$, then $P ::= N_i$,
- $[y]$ for some variable $y \neq x_i$, then $P ::= M$,
- [⟨self-evaluating⟩, ⟨symbol⟩, ⟨procedure-constant⟩, or ⟨pairing-operator⟩], then $P ::= M$,
- [`(if` $T\ C\ A$`)`], then $P ::=$ `(if` $T[\mathbf{x}_k := \mathbf{N}_k]\ \ C[\mathbf{x}_k := \mathbf{N}_k]\ \ A[\mathbf{x}_k := \mathbf{N}_k]$`)`,
- [`(`$M_1\ \ldots\ M_m$`)`], then $P ::=$ `(`$M_1[\mathbf{x}_k := \mathbf{N}_k]\ \ldots\ M_m[\mathbf{x}_k := \mathbf{N}_k]$`)`,
- [`(lambda (`$\mathbf{y}_m$`)` N`)`], then
  - if $\{\mathbf{x}_k\} \subseteq \{\mathbf{y}_m\}$, then $P ::= M$,
  - letting $x_{i_1}, \ldots, x_{i_l}$ be the subsequence of the variables $\mathbf{x}_k$ that do *not* appear in $\{\mathbf{y}_m\}$ and $\mathbf{z}_m$ be fresh variables, then

    $$P ::= \texttt{(lambda (}\mathbf{z}_m\texttt{)}\ N[x_{i_1}, \ldots, x_{i_l}, \mathbf{y}_m := N_{i_1}, \ldots, N_{i_l}, \mathbf{z}_m]\texttt{)}$$

- [`(letrec ((`$y_1\ M_1$`)` $\ldots$ `(`$y_m\ M_m$`))` N`)`], then
  - if $\{\mathbf{x}_k\} \subseteq \{\mathbf{y}_m\}$, then $P ::= M$,
  - let $x_{i_1}, \ldots, x_{i_l}$ be the subsequence of the variables $\mathbf{x}_k$ that do *not* appear in $\{\mathbf{y}_m\}$, and $\mathbf{z}_m$ be fresh variables, then

    $$P ::= \texttt{(letrec ((}z_1\ \ M'_1\texttt{)}\ \ldots\ \texttt{(}z_m\ \ M'_m\texttt{))}\ \ N'\texttt{)},$$

    where

    $$\begin{aligned} M'_i &::= M_i[x_{i_1}, \ldots, x_{i_l}, \mathbf{y}_m := N_{i_1}, \ldots, N_{i_l}, \mathbf{z}_m], \\ N' &::= N[x_{i_1}, \ldots, x_{i_l}, \mathbf{y}_m := N_{i_1}, \ldots, N_{i_l}, \mathbf{z}_m]. \end{aligned}$$

Definition 4.4. A *context*, C, is a Scheme expression except that the hole token, ⟨hole⟩, may serve as a free variable; the hole may only occur once. We write $C[M]$ to denote the result of replacing the hole in C by M *without any renaming* of bound variables.

Problem 4. Write a BNF grammar for ⟨context⟩.

Definition 4.5. *α-equivalence* is the smallest equivalence relation on expressions such that

- `(lambda (`$\mathbf{x}_m$`)` N`)` $=_\alpha$ `(lambda (`$\mathbf{z}_m$`)` $N[\mathbf{x}_m := \mathbf{z}_m]$`)`,
  for any sequence $\mathbf{z}_m$ of fresh variables,
- `(letrec ((`$x_1\ M_1$`)`$\ldots$`(`$x_m\ M_m$`))` N`)` $=_\alpha$ `(letrec ((`$z_1\ M'_1$`)`$\ldots$`(`$z_m\ M'_m$`))` N'`)`,
  where $M'_i ::= M_i[\mathbf{x}_m := \mathbf{z}_m]$, $N' ::= N[\mathbf{x}_m := \mathbf{z}_m]$, and $\mathbf{z}_m$ are fresh variables,

- if $M =_\alpha N$, then $C[M] =_\alpha C[N]$ for any context C.

Problem 5. **(a)** Prove that α-equivalence is an equivalence relation.

(b) Prove that if $M_1 =_\alpha M_2$, then M_1 is a $\langle$syntactic-value$\rangle$ iff M_2 is also; likewise M_1 is an $\langle$immediate-redex$\rangle$ iff M_2 is also.

(c) Prove that if $M_1 =_\alpha M_2$ and $M_1 \rightarrow N_1$, then there is an N_2 such that $N_2 =_\alpha N_1$ and $M_2 \rightarrow N_2$.

Problem 6. Write a two argument Scheme procedure `alpha=?` that determines whether its arguments are α-equivalent Scheme expressions. That is, if M, N are α-equivalent Scheme expressions, then `(alpha=? '`M` '`N`)` returns `#t`, otherwise it returns `#f`.

We observed that renaming bound variables to ensure expressions satisfy the Variable Convention implies that expressions are determined only up to α-equivalence. To avoid constant reference to α-equivalence in subsequent sections, we will implicitly identify α-equivalent expressions: from now on, when we say two expressions are "equal", we actually will mean only that they are α-equivalent, and when we say an expression is "uniquely determined", we mean it is determined up to α-equivalence.

5 Repeated Rewriting

Starting with a Scheme expression and successively applying Substitution Model rewrite rules leads to a sequence of expressions that correspond to the steps that a standard interpreter would perform in evaluating the starting expression. When the rules no longer apply, the evaluation is complete—either successfully with the return of the value of the expression, or unsuccessfully because of an error of some type. We'll distinguish errors caused by lookup of an undefined variable from other types of errors, because an expression that causes a lookup error may do something interesting in an extended environment where the undefined variable is assigned a value.

Definition 5.1. An *error combination* is

- an expression, M, of the form `(`$\langle$procedure-constant$\rangle$ $\langle$syntactic-value$\rangle^*$`)` such that $M \not\rightarrow$, or of the form `(`V $\langle$syntactic-value$\rangle^*$`)` where V is a $\langle$syntactic-value$\rangle$ but not a $\langle$procedure$\rangle$.
- a $\langle$lambda-expression$\rangle$ applied to the wrong number of arguments, namely, an expression of the form

  `((lambda ()` $\langle$expression$\rangle$`)` $\langle$syntactic-value$\rangle^+$`)`,

  or

  `((lambda (`x $\langle$variable$\rangle^+$`)` $\langle$expression$\rangle$`))`.

- `(cons` $\langle$syntactic-value$\rangle^*$`)` if there are not exactly two values to which the `cons` is applied.

Error combinations cause immediate dynamic errors in Scheme, *e.g.*,

```
           (+ 'a 0),
             (/ 1 0)
           ('+ 1 0),
((lambda () (f 1)) 2),
            (cons 1)
```

are error combinations.

Definition 5.2. An *error* `letrec` is an expression of the form

$$\texttt{(letrec ((}\langle\text{value-binding}\rangle^* \texttt{ (}\langle\text{variable}\rangle\ R[x]\texttt{)}\ \langle\text{binding}\rangle^*\texttt{)}\ \langle\text{expression}\rangle\texttt{)}$$

where the indicated occurrence of x is bound by one of the `letrec` bindings[2]. An arbitrary Scheme expression, M, is *an immediate error* if it is of the form $R[N]$ or Env$(R[N])$ for some control context, R, and expression N that is an error combination or error `letrec`.

An expression, M is a *lookup error* if it has a free variable, x, and is of one of the forms $R[x]$, Env$(R[x])$, $R[$`(set!` x $\langle$syntactic-value$\rangle$`)`$]$, or Env$(R[$`(set!` x $\langle$syntactic-value$\rangle$`)`$])$.

Definition 5.3. A *final value* is a $\langle$syntactic-value$\rangle$ or an expression of the form Env($\langle$syntactic-value$\rangle$).

Corollary 5.4. *Let M be a Scheme expression. Then $M \not\to$ if and only if*

1. *M is a final value, or*
2. *M is an immediate error or a lookup error.*

Note that Corollary 5.4.2 describes the expressions that cause an immediate dynamic error in Scheme.

Definition 5.5. The notation $M \xrightarrow{n} N$ means that M rewrites to N by n successive applications of Substitution Model rewrite rules; $M \xrightarrow{*} N$ means that $M \xrightarrow{n} N$ for some $n \in \mathbb{N}$.

M converges to N when $M \xrightarrow{*} N$ and N is a final value, called the *final value of M*. The notation $M \downarrow N$ indicates that M converges to N, and $M\downarrow$ indicates that M converges to some final value.

M diverges when there is an immediate error, N, such that $M \xrightarrow{*} N$, or there is no N such that $M \xrightarrow{*} N$ and $N \not\to$. The notation $M\uparrow$ indicates that M diverges.

M leads to a lookup error when $M \xrightarrow{*} N$ for some lookup error N.

The Unique Rewriting Corollary 3.3 implies that if $M \xrightarrow{*} N$ and $N \not\to$, then N is uniquely determined. So we have:

[2]Error `letrec`'s don't always cause errors in all Scheme implementations, but it is consistent with the Revised^5 Scheme Manual for all of them to cause errors.

Lemma 5.6. *For every expression, M, exactly one of the following holds: $M\downarrow$, $M\uparrow$, or M leads to a lookup error.*

Note that a *closed* expression, that is, one with no free variables, cannot lead to a lookup error, so it diverges iff it does not converge.

Lemma 5.7. *If M diverges, then so does* `(letrec (`B`)` $R[M]$`)`.

Problem 7. Prove Lemma 5.7

Problem 8. **(a)** Prove that

$$M \downarrow \texttt{(letrec (}B_1\texttt{)}\ V\texttt{)} \quad \text{implies} \quad \texttt{(letrec (}B_2\texttt{)}\ M\texttt{)} \downarrow \texttt{(letrec (}B_2\, B_1\texttt{)}\ V\texttt{)}.$$

(b) Prove that

$$M \downarrow \texttt{(letrec (}B\texttt{) 3)} \quad \text{implies} \quad \texttt{(+}\ M\ \ M\texttt{)} \downarrow \texttt{(letrec (}B'\texttt{) 6)},$$

for some value bindings, B'.

(c) Exhibit value bindings, B, and an expression, M, such that

$$\texttt{(letrec (}B\texttt{)}\ M\texttt{)}\downarrow \texttt{(letrec (}B'\texttt{) 3)},$$

but

$$\texttt{(letrec (}B\texttt{) (+}\ M\ \ M\texttt{))}\uparrow .$$

Hint: `set!` will have to appear in M.

6 Garbage Collection

Garbage collection refers to the process whereby Lisp-like programming systems recapture inaccessible storage space. An attraction of Lisp-like languages is that garbage colection occurs behind the scenes, freeing the programmer from responsibility for explicit allocation and deallocation of storage blocks.

There are two rules of the Substitution Model corresponding to garbage collection. These garbage collection rules are distinguished from the other rewrite rules because they can be applied at any time—just as garbage collection can occur at any time during a computation. In particular, both a garbage collection rule and a regular Substitution Model rewrite rule may be applicable to the same expression, so the Unique Rewriting Corollary 3.3 will need to be qualified. We'll explain how to reformulate the Unique Rewriting property in Section 6.2 below.

6.1 Environment Garbage Collection

In our Substitution Model, garbage collectable storage in a Scheme computation corresponds to unneeded bindings in the environment `letrec` of an expression. The *environment garbage collection rule* is

$$\texttt{(letrec (}B\texttt{)}\ N\texttt{)} \to \texttt{(letrec (}B'\texttt{)}\ N\texttt{)},$$

where B' is a subsequence of the value bindings B, and none of the free variables in the contractum `(letrec (`B'`)` N`)` were bound by the omitted bindings, namely, the bindings in B that do not appear in B'. For example,

```
(letrec ((a (lambda () b))
         (b 3)
         (c (lambda () (* b (f))))
         (d (lambda () f))
         (f 4))
 (+ 1 (a) ((lambda (c) (c 5 6)) -)))
```

can rewrite by this garbage collection rule to

```
(letrec ((a (lambda () b))
         (b 3)
         (f 4))
 (+ 1 (a) ((lambda (c) (c 5 6)) -)))
```

because there are no free occurrences of `c` or `d` in the rewritten expression. This second expression could in turn be rewritten by the garbage collection rule to

```
(letrec ((a (lambda () b))
         (b 3))
 (+ 1 (a) ((lambda (c) (c 5 6)) -)))
```

Of course, the garbage collection rule would also have allowed the first expression to rewrite directly to this last.

An efficient way to apply the garbage collection rule is to identify all the variables which are "needed" by the body of the `letrec` and erase the bindings for the rest of the variables. Here is a recursive way to find these *needed variables* in an expression `(letrec (`B`)` N`)`:

- All free variables of N are needed.
- If x is a needed variable and `(`$x\ V$`)` is a binding in B, then the free variables of V are also needed.

A rule that collects all the garbage in an environment can now be described as

$$\texttt{(letrec (}B\texttt{)}\ N\texttt{)} \to \texttt{(letrec (}B'\texttt{)}\ N\texttt{)},$$

where B' is the subsequence of B consisting of the bindings of the needed variables in `(letrec (`B`)` N`)`.

6.2 Equivalence up to Garbage Collection

The garbage collection rules allow an expression to be rewritten in different ways. For example, we saw above that

```
(letrec ((a (lambda () b))
         (b 3)
         (c (lambda () (* b (f))))
         (d (lambda () f))
         (f 4))
 (+ 1 (a) ((lambda (c) (c 5 6)) -)))
```

can be rewritten to

```
(letrec ((a (lambda () b))
         (b 3))
 (+ 1 (a) ((lambda (c) (c 5 6)) -)))
```

by the environment garbage collection rule, but it can also be rewritten in a completely different way by instantiating `a`:

```
(letrec ((a (lambda () b) )
         (b 3)
         (c (lambda () (* b (f))))
         (d (lambda () f))
         (f 4))
 (+ 1 ((lambda () b)) ((lambda (c) (c 5 6)) -)))
```

So Unique Rewriting Corollary 3.3 no longer holds, forcing us to consider the possibility that expressions may no longer rewrite to a unique final form. But they do:

Theorem 6.1. *If M is a Scheme expression and $M \xrightarrow{*} N$ for some N such that $N \not\rightarrow$, then this N is uniquely determined.*

A simple way to prove this result is to observe that the full set of Substitution Model rules—including both the regular and garbage collection rules—satisfies the Diamond Lemma, also known as the Strong Confluence property (cf. Mitchell's text, p. 224). We can even recover a Unique Rewriting Corollary by changing uniqueness up to α-equivalence into uniqueness up to garbage collection.

Definition 6.2. *Garbage-collection equivalence*, $=_{\text{gc}}$, is the smallest equivalence relation on expressions such that

- if $M =_\alpha N$, or if M rewrites to N by application of a garbage collection rule, then $M =_{\text{gc}} N$,
- if $M =_{\text{gc}} N$, then $C[M] =_{\text{gc}} C[N]$ for any context C.

Now we can recover the Unique Rewriting Corollary above:

Corollary 6.3. *(**Unique Rewriting up to Garbage Collection**) If M is a Scheme expression, and $M \rightarrow N_1$ and $M \rightarrow N_2$ for some Scheme expressions N_1, N_2, then $N_1 =_{gc} N_2$.*

7 Observational Equivalence

Revising a program to improve performance is a familiar programming activity. A trivial example would be to replace an occurrence of a subexpression of the form `(+ 1 2)` with the subexpression `3`. The revised program would then perform fewer additions, but would otherwise yield the same results as the original. Well not quite "the same"—the expressions

```
(lambda (x) (* (+ 1 2) x))
```

and

```
(lambda (x) (* 3 x))
```

obviously describe *different* procedures—applying the first will lead to an addition operation that is not performed by the second. But we would observe the same numerical result if we applied either of these procedures to the same numerical argument. Also, we would observe an error if we applied either of them to a nonnumerical value. As long as all we care to observe about an evaluation is the number, if any, that results from an evaluation, these two procedure values will be indistinguishable. This is the sense in which they are "the same."

An observable property of an expression evaluation that is even more fundamental than its numerical value, is whether any value is returned at all. We choose this more general observation of convergence as the basis for our definition of distinguishability.

Definition 7.1. Two Scheme expressions, M and N, are said to be *observationally distinguishable* iff there is a context, C, such that exactly one of $C[M]$ and $C[N]$ converges. Such a context is called a *distinguishing context* for M and N. If M and N are not observationally distinguishable, they are said to be *observationally equivalent*, written $M \equiv N$.

It is easy to verify that observational equivalence is actually an equivalence relation. Moreover, it follows immediately from its definition that it is a congruence relation[3], namely,

$$M \equiv N \quad \text{implies} \quad C[M] \equiv C[N]$$

for all contexts C. So the sense in which it is always OK to replace `(+ 1 2)` by `3` is captured by noting that `(+ 1 2)` $\equiv$ `3`.

The following problems show that observing numerical results, rather than just observing convergence, yields the same observational equivalence relation on Scheme expressions.

Problem 9. **(a)** Show that if $M \equiv N$, and $M \downarrow k$ for some integer k, then, $N \downarrow k$.

(b) Show that if M and N are distinguishable, and k is an integer, then there is a context, C, such that the one of $C[M]$ and $C[N]$ has final value k and the other one diverges.

(c) Conclude that $M \equiv N$ if and only if

$C[M]$ has a numerical value iff $C[N]$ has the same numerical value,

[3]For this reason, observational equivalence is sometimes called observational *congruence*.

for all contexts, C.

It is implicit in the definition of context that *the hole may not be quoted*. So, for example,

```
(eq? (quote [ ]) (quote hello))
```

is not a context—because when an ⟨identifier⟩ is quoted, it parses as a ⟨symbol⟩ not a ⟨hole⟩.[4] This is a minor, but necessary, technicality as indicated in the next problem.

Problem 10. Explain why no two expressions would be observationally equivalent if contexts could have a quoted hole.

In Scheme, it doesn't matter in what order the bindings appear in a `letrec` as long as all the variables are bound to syntactic values. "It doesn't matter" more precisely means observational equivalence:

Lemma 7.2. *If M_i, M_j are syntactic values, then*

$$\begin{aligned}&\texttt{(letrec ((}x_1\,M_1\texttt{)}\ \ldots\ \texttt{(}x_i\,M_i\texttt{)}\ \ldots\ \texttt{(}x_j\,M_j\texttt{)}\ \ldots\ \texttt{)}\,M\texttt{)}\\ \equiv\ &\texttt{(letrec ((}x_1\,M_1\texttt{)}\ \ldots\ \texttt{(}x_j\,M_j\texttt{)}\ \ldots\ \texttt{(}x_i\,M_i\texttt{)}\ \ldots\ \texttt{)}\,M\texttt{)}\end{aligned}$$

The proof of Lemma 7.2 is similar to the proof that rewriting preserves α-equivalence in Problem 5.

Problem. Prove Lemma 7.2.

8 Context Rewriting

A direct approach to proving observational equivalences involves examining how the context of an expression can be rewritten, given some limited information about the kind of expression that is in the hole, but without knowing the exact expression.

For example, suppose E_1 is the context:

```
(letrec
  ((cpn
    (lambda (v n)
     (if (zero? n) (list) (cons v (cpn (list v) (- n 1)))))))
 (cpn [ ] 2))
```

The Instantiation Rule allows the operator `cpn` to be replaced by the `lambda` expression:

[4]As a matter of fact, in real Scheme, delimiters like "]" and "[" cannot appear as characters in ⟨identifier⟩'s, so technically, `(quote [ ])` is not an ⟨expression⟩.

```
(letrec
  ((cpn
    (lambda (v n)
     (if (zero? n) (list) (cons v (cpn (list v) (- n 1)))))))
  ((lambda (v n)
    (if (zero? n) (list) (cons v (cpn (list v) (- n 1)))))
      [ ] 2))
```

This resulting expression does not rewrite because it is a lookup error of the hole variable. But suppose we can assume that the hole will be replaced by some unknown expression that is guaranteed to be a syntactic value. So we can treat the hole as a value, and the rules for `lambda` can be applied. Now, in a few steps, E_1 rewrites to:

```
(letrec ((cpn ...) (n 2) (v [ ]))
  (cons [ ]
        ((lambda (n)
          (if (zero? n) (list) (cons v (cpn (list v) (- n 1)))))
         1)))
```

Continuing in this way, we can find value bindings, B, such that E_1 converges to

$$F_1 ::= \texttt{(letrec (}B\texttt{) (list [ ] (list [ ])))}.$$

That is, there is a context F_1 such that for any syntactic value, V, $E_1[V] \downarrow F_1[V]$. Notice that F_1 is technically not a context because it has more than one occurrence of a hole; we'll call F_1 a *multihole* context.

Definition 8.1. A *multihole context*, C, is a Scheme expression except that the ⟨hole⟩, may serve as a free variable; it may have any number of occurrences.

If C is a multihole context, we write $C[M]$ for the result of replacing all occurrences of ⟨hole⟩ in C by M, without any renaming of bound variables. More generally, if C has n occurrences of holes, then for a sequence $\mathbf{M}_n ::= M_1, \ldots, M_n$, of expressions, we write

$$C[M_1]_1 \ldots [M_n]_n, \text{ abbreviated } C[\mathbf{M}_n],$$

to denote the result of replacing the ith occurrence of ⟨hole⟩ in C by M_i, without any renaming of bound variables.

Now let V be the context `(lambda (x) (+ x [ ] (* 3 4)))`, and let $E_2 ::= E_1[V]$ and $F_2 ::= F_1[V]$. That is, E_2 is

```
(letrec
    ((cpn
       (lambda (v n)
         (if (zero? n) (list) (cons v (cpn (list v) (- n 1)))))))
  (cpn (lambda (x) (+ x [ ] (* 3 4))) 2)),
```

and F_2 is

```
(letrec (B) (list (lambda (x) (+ x [ ] (* 3 4)))
                    (list (lambda (x) (+ x [ ] (* 3 4)))))).
```

Since V is a syntactic value, we know that $E_2 = E_1[V] \downarrow F_1[V] = F_2$. Since we concluded this without any assumptions about what might be in the hole in V, it follows that $E_2[M] \downarrow F_2[M]$ for every expression, M.

For a second example, let E_3 be the same as the context E_2 except that `caadr` of the body of E_2 is applied to `5`. That is, E_3 is

```
(letrec
    ((cpn
       (lambda (v n)
        (if (zero? n) (list) (cons v (cpn (list v) (- n 1)))))))
  ((caadr (cpn (lambda (x) (+ x [ ] (* 3 4))) 2))
   5))
```

Now let F_3 be the corresponding modification of F_2, namely, F_3 is

```
(letrec (B)
  ((caadr (list (lambda (x) (+ x [ ] (* 3 4)))
                (list (lambda (x) (+ x [ ] (* 3 4))))))
   5)).
```

Now by Lemma 5.7, with $R =$ `((caadr [ ]) 5)`, we can conclude that $E_3 \xrightarrow{*} F_3$. But F_3 can be further rewritten using rules for `car` and `cdr` until its body is

```
((lambda (x) (+ x [ ] (* 3 4))) 5)
```

which will rewrite in a few steps to `(+ 5 [ ] (* 3 4))`. That is, there is an F_4 of the form

`(letrec (`B'`) (+ 5 [ ] (* 3 4)))`

such that $E_3 \xrightarrow{*} F_4$. Notice that the body of F_4 is a control context, and no further rewriting is possible at this point because F_4 is a lookup error of the hole variable.

So we can say that $E_3[M] \xrightarrow{*} F_4[M]$ for every expression M, and moreover F_4 is of the form `(letrec (`B'`)` $R[\ ]$`)` for the control context $R =$ `(+ 5 [ ] (* 3 4))`. It follows, for example, that the numerical value of $E_3[6]$ is 23. It also follows that if $M\uparrow$, then also $F_4[M]\uparrow$, and hence $E_3[M]\uparrow$.

These examples illustrate how any multihole context can be rewritten, based on partial information about the expressions that may appear in its holes. The partial information that is often available is the *kind* of expressions to be proved equivalent, namely:

Definition 8.2. An expression is of *nonvalue kind* if it is not a ⟨syntactic-value⟩. An expression is of *procedure kind* if it is a ⟨nonpairing-procedure⟩. An expression is of *constant kind* if it is either ⟨self-evaluating⟩ or a ⟨symbol⟩. An expression is of *structured kind* if it is either a ⟨pair-value⟩ or a ⟨list-value⟩.

If $\mathbf{M}_n$ is a sequence of expressions of various kinds, then the *kind pattern* of $\mathbf{M}_n$, is the sequence $\mathbf{k} ::= k_1, \ldots, k_n$ such that $k_i \in \{$*nonval, proc, constant, struct*$\}$ indicates the kind of M_i, for $1 \le i \le n$.

Except for the pairing operators `list` and `cons`, every Scheme expression is of exactly one of these four kinds. The pairing operators play a special role in the Substitution Model, because applications of ⟨pairing-operator⟩'s to values are themselves values, rather than combinations that are immediate-redexes. It's convenient to designate these operators as not having a kind, ensuring that `list` and `cons` will not by themselves be expressions in set of expressions "of various kinds." (But `list` and `cons` may certainly appear as *sub*expressions of expressions of various kinds).

The examples above illustrate how to rewrite a context until it is guaranteed to converge or gets to a point where more information than the kind of expressions to be placed in the holes is needed to continue. A further example is the context

`(if (list [ ]) 'yes 'no)`.

Given that the hole will be replaced with a syntactic value, the body can be rewritten to the ⟨symbol⟩ `'yes`. But if an expression of kind *nonvalue* is to go in the hole, then rewriting cannot proceed without more information about the expression.

The Standard Context Lemma 8.4 below summarizes the way contexts can be rewritten. To state it, we need to generalize control contexts to have multiple holes.

Definition 8.3. If C is a multihole context with $n+1$ holes one of which is designated as the "main hole", and $\mathbf{M}_n ::= M_1, \ldots, M_n$ is a sequence of expressions, we write

$$C[M_1]_1 \ldots [M_n]_n[\,], \text{ abbreviated } C[\mathbf{M}_n][\,],$$

to denote the single hole context that results from replacing the n non-main occurrences of ⟨hole⟩ by the expressions $\mathbf{M}_n$, without any renaming of bound variables.

A *multihole control context* for kind pattern $\mathbf{k}$ is a multihole context, R, such that $R[\mathbf{M}_n][\,]$ is a control context for all expressions $\mathbf{M}_n$ with kind pattern $\mathbf{k}$.

For example,

`(+ [ ]₁ 2 (* [ ]₂ [ ] (- n [ ]₃)))`

is a control context for any kind pattern in which the first and second holes would be assigned expressions having one of the value kinds *proc, constant, struct*. Also

`(letrec ((n [ ]₄)) (+ [ ]₁ 2 (* [ ]₂ [ ] (- n [ ]₃))))`

is a control context for any kind pattern in which $[\,]_1$, $[\,]_2$, and $[\,]_4$ have one of the value kinds *proc, constant, struct*; the kind of $[\,]_3$ doesn't matter.

This example illustrates the fact that if there are *some* expressions $\mathbf{M}_n$ with kind pattern $\mathbf{k}$ such that $R[\mathbf{M}_n][\,]$ is a control context, then R must be a control context for $\mathbf{k}$. That is, $R[\mathbf{M}'_n][\,]$ will be a control context for all $\mathbf{M}'_n$ with kind pattern $\mathbf{k}$. This follows because the only distinction among expressions used by the BNF rules specifying control contexts is whether an expression is a syntactic value. In fact R will also be a control context for all patterns $\mathbf{k}'$ obtained by changing any of the kinds in $\mathbf{k}$ into any of the *value* kinds.

Lemma 8.4. *(**Standard Context**) Let E be a multihole context, and let $\mathbf{k} ::= k_1, \ldots, k_n$ be a kind pattern. Then either*

1. *$E[\mathbf{M}_n] \downarrow F[\mathbf{M}_n]$ for some some context F and all expressions $\mathbf{M}_n$ with kind pattern $\mathbf{k}$, or*
2. *$E[\mathbf{M}_n] \xrightarrow{*} F[\mathbf{M}_n]$ for some context, F, and variable, x, such that $F[\mathbf{M}_n]$ is a lookup error of x for all expressions $\mathbf{M}_n$ with kind pattern $\mathbf{k}$, or*
3. *$E[\mathbf{M}_n]\uparrow$ for all expressions $\mathbf{M}_n$ with kind pattern $\mathbf{k}$, or*
4. *there is a control context, R, and an integer i, $1 \leq i \leq n$, such that for all expressions $\mathbf{M}_n$ with kind pattern $\mathbf{k}$, either M_i is of kind*
   - *(a)* ***nonvalue****, and $E[\mathbf{M}_n] \xrightarrow{*} R[\mathbf{M}_n][M_i]$.*
   - *(b)* ***procedure****, and there is a (possibly empty) sequence of syntactic values $V_1, \ldots, V_k$, such that*

$$E[\mathbf{M}_n] \xrightarrow{*} R[\mathbf{M}_n][\ \texttt{(}M_i\ V_1 \ldots\ V_k\texttt{)}\ ].$$

   - *(c)* ***constant****, and there is a sequence of syntactic values $V_1, \ldots, V_k$, $V_{k+1} \ldots$ and a* ⟨procedure-constant⟩, op, *such that*

$$E[\mathbf{M}_n] \xrightarrow{*} R[\mathbf{M}_n][\ \texttt{(}\mathit{op}\ V_1 \ldots\ V_k\ M_i\ V_{k+1} \ldots\texttt{)}\ ].$$

   - *(d)* ***pair****, and*

$$E[\mathbf{M}_n] \xrightarrow{*} R[\mathbf{M}_n][\ \texttt{(}\mathit{op}\ M_i\texttt{)}\ ],$$

   for op $\in \{\texttt{car}, \texttt{cdr}, \texttt{null?}, \texttt{pair?}\}$.

The proof of the Standard Context Lemma 8.4 involves analyzing, along the lines of the examples above, how a control context for a given kind pattern can control parse. We omit the proof.

9 Proving Observational Equivalence

The Standard Context Lemma provides a basis for proving many observational equivalences. For example, Scheme subexpressions that diverge can cause an evaluation to diverge, but otherwise are useless. In fact, they are all equally useless. More precisely, the following fundamental observational equivalence holds:

Theorem 9.1. *If $M\uparrow$ and $N\uparrow$, then $M \equiv N$.*

To prove Theorem 9.1, we need to show that for any context, E, if $E[M]$ converges, then so does $E[N]$. Intuitively, this follows from the fact that, since $E[M]\downarrow$ and $M\uparrow$, the subexpression M can never have been evaluated during the evaluation of $E[M]$, so the convergence of $E[M]$ does not depend on what is in the hole. This intuition is captured in the Standard Context Lemma 8.4, and Theorem 9.1 is an easy corollary.

Proof. We can prove something stronger, namely, if E is a *multihole* context, and $E[M]\downarrow$, then $E[N]\downarrow$.

So suppose $E[M]\downarrow$. One possibility is that Standard Context Lemma 8.4.1 applies, namely $E[\mathbf{M}_n] \downarrow F[\mathbf{M}_n]$ for all $\mathbf{M}_n$. In particular, $E[N]\downarrow$, as required.

Since M diverges, it is of nonvalue kind. So the only other possibility is that the Lemma 8.4.4a applies, namely $E[M] \xrightarrow{*} R[M]$ for some control context, R. But $R[M]\uparrow$, by Lemma 5.7, and hence $E[M]\uparrow$, a contradiction.

□

The Standard Context Lemma also captures the property that Scheme evaluation is *sequential*, namely, if a context depends on what's in its holes, then there is a particular hole whose contents are always evaluated first. So behavior that requires evaluating holes in parallel is beyond Scheme's expressive power:

Corollary 9.2. *There is no Scheme context, G, such that for all closed expressions M, N,*

$$G[M, N]\downarrow \quad \textit{iff} \quad M\downarrow \textit{ or } N\downarrow .$$

Proof. Suppose to the contrary that there was such a G.

Now if Standard Context Lemma 8.4.1 applies to G, then $G[\mathbf{M}_n]\downarrow$ for all $\mathbf{M}_n$. In particular, $G[M, M]\downarrow$ for any divergent expression, M, contradicting the fact that $G[M, M]$ should diverge in this case.

So the only other possibility is that the Lemma 8.4.4 applies. In particular, there is a control context, R, for nonvalue kinds and an integer, i, such that $G[M_1, M_2] \xrightarrow{*} R[M_1, M_2][M_i]$ for all M_1, M_2 of nonvalue kind. Without loss of generality, suppose $i = 1$. Then choose some M that diverges, and let N be some convergent expression of nonvalue kind, *e.g.*, $N =$ `(+ 1)`. By Lemma 5.7, $R[M, N][M]$ diverges, so $G[M, N]$ does too, contradicting the fact that $G[M, N]$ should converge because N converges.

□

We can now also give a precise formulation of the slogan "A Scheme procedure is a black-box," which reflects the idea that the only way to learn about a procedure is by applying it to arguments. Another way to say this is that if two procedures can be distinguished from each other, it is only because there is a set of arguments on which they yield distinguishable results.

Corollary 9.3. *[Operational Extensionality] If M_1 and M_2 are closed expressions of procedure kind and*

$$\texttt{(}M_1 \;\; V_1 \ldots V_n\texttt{)} \equiv \texttt{(}M_2 \;\; V_1 \ldots V_n\texttt{)}$$

for all $n \geq 0$ and closed ⟨syntactic-value⟩*'s $V_1, \ldots, V_n$, then*

$$M_1 \equiv M_2.$$

Problem. Prove Corollary 9.3.

Problem 11. Prove Lemma 7.2.

Problem 12. Prove that if R is a control context and no free variable of M occurs in $R[x]$, then

$$\texttt{((lambda (}x\texttt{)}\ R[x]\texttt{)}\ M\texttt{)} \equiv R[M].$$

Note that M need not be a ⟨syntactic-value⟩.

A fairly powerful method for proving that two expressions are observationally equivalent is to repeatedly apply Substitution Model rewrite rules to their subexpressions until the two expressions have been rewritten to be the same. The following Lemma shows that this is a sound way to prove observational equivalences.

Lemma 9.4. *If $M \to N$, then $M \equiv N$.*

Problem 13. **(a)** Let M, N be Scheme expressions such that $M \to N$. Call a sequence of expressions an M, N *sequence* if it is a sequence of M's and N's. Show that if $C[\mathbf{M}_n]\downarrow$ for some M, N sequence, $\mathbf{M}_n$, and multihole context, C, then $C[\mathbf{M'}_n]\downarrow$ for all M, N sequences, $\mathbf{M'}_n$. *Hint:* By induction on the number of steps $C[\mathbf{M}_n]$ takes to converge. Use the Context Independence Corollary 3.4.

(b) Use part (a) to complete a proof of Lemma 9.4.

Problem 14. **(a)** Describe an expression M such that `(+` M M`)` and `(* 2` M`)` are *not* observationally equivalent. What is the distinguishing context?

(b) Show that if M is *closed*, then

$$\texttt{(+}\ M\ \ M\texttt{)} \equiv \texttt{(* 2}\ M\texttt{)}.$$

The Standard Context Lemma 8.4 also allows us to deduce interesting observational equivalences that do not simply follow by rewriting subexpressions or equating divergent ones.

Lemma 9.5. *If E is a context such that for each sequence, $\mathbf{M}_n$, of expressions of procedure kind, $E[\mathbf{M}_n]$ converges to some number, then in fact $E[\mathbf{M}_n]$ converges to the same number for all $\mathbf{M}_n$.*

Proof. Suppose $\mathbf{M}_n$ are chosen to be procedures of the form `(lambda (x)` D`)` where D is a divergent expression. Then since $E[\mathbf{M}_n]\downarrow$, only the first case of the Standard Context Lemma can apply. That is, $E[\mathbf{M}_n] \downarrow F[\mathbf{M}_n]$ for some context F and all expressions $\mathbf{M}_n$ of procedure kind. Since $E[\mathbf{M}_n]$ has a numerical value, $F[\mathbf{M}_n]$ must in fact be some number n. Hence $F[\mathbf{M'}_n]$ will also be n. $\square$

Problem 15. Let T_1, T_2 be procedure expressions such that `(`T_1`)`$\downarrow$ and `(`T_2`)`$\uparrow$, and let M be any closed expression. Prove that

$$\texttt{(- (}M\ T_1\ T_2\texttt{) (}M\ T_2\ T_1\texttt{))} \equiv \texttt{(* 0 (}M\ T_1\ T_2\texttt{) (}M\ T_2\ T_1\texttt{))}.$$

Hint: Use the Standard Context Lemma and Lemma 9.5. Argue by cases according to whether `(`$M\ T_1\ T_2$`)` converges to a number, converges to a non-number, diverges, or causes a lookup error.

The samples above illustrate how a rich set of observational equivalences can be verified using the Standard Context Lemma. Further examples include several of the list rewriting rules that hold as observational equivalences even when the arguments are not values, *e.g.*,

Example 9.6.

$$\begin{aligned}\texttt{(cons } M_1 \texttt{ (list } M_2\ M_3\texttt{))} &\equiv \texttt{(list } M_1\ M_2\ M_3\texttt{)},\\ \texttt{(apply } M \texttt{ (list } N_1 \ \ldots\ \texttt{))} &\equiv \texttt{(}M\ N_1\ \ldots\ \texttt{)}\end{aligned}$$

Also, variables can only be instantiated by ⟨syntactic-value⟩'s, so many equivalences involving syntactic values will hold when variables appear instead of values. For example, we observed in Problem 14

$$\texttt{(* } M\ M\texttt{))} \not\equiv \texttt{(* 2 } M\texttt{)},$$

because M may have side-effects on a free variable, but we have:

Example 9.7.

$$\texttt{(* x x)} \ \equiv\ \texttt{(* 2 x)}$$

Finally, we state an equivalence that reflects a deeper property of Scheme: external procedures can only affect local variables if they are explicitly passed the ability to do so. For example, if only the ability to add 2 to some local variable `x` is passed to an external procedure, `pr`, and the value of `x` is initially even, then it will still be even if and when the external procedure returns a value:

Example 9.8.

```
(letrec((x 0))(begin(pr (lambda() (set! x (+ x 2)))) #t))
  ≡ (letrec((x 0))(begin(pr (lambda() (set! x (+ x 2)))) (even? x)))
```

Problem 16. Prove the equivalence in Example 9.8. **Warning**: this may be hard.

10 Calling Continuations

Most programmimg languages include various "escape" and error handling mechanisms that allow processes to be interrupted with direct return of a value. Scheme provides a single, every general feature of this kind, called `call-with-current-continuation`, or `call/cc` for short.

10.1 Rules for `call/cc`

There is a fairly simple way to extend the Substitution Model to handle `call/cc`. We add two new procedure constants:

$$\langle\text{nonpairing-procedure}\rangle ::= \ldots \mid \texttt{call/cc} \mid \texttt{abort}$$

The `call/cc` and `abort` procedures each take one argument. The process of applying `call/cc` involves creation of a new *continuation procedure* which describes how the evaluation will continue once the value of the `call/cc` application is found.

In the Substitution Model, the control context surrounding an immediate redex specifies the further evaluation to be performed once the value of the redex has been found. This is reflected in two control rules for `call/cc`. The first rule is

$$R[\texttt{(call/cc }V\texttt{)}] \to R[\texttt{(}V\texttt{ (lambda (}x\texttt{) (abort }R[x]\texttt{)))}],$$

where x is fresh. Here the expression `(lambda (`x`) (abort `$R[x]$`))` describes the current continuation.

The second rule is the same as the first but in the context of the environment `letrec`:

$$\begin{array}{l}\texttt{(letrec (}B\texttt{) }R[\texttt{(call/cc }V\texttt{)}]\texttt{)} \\ \quad\to\ \texttt{(letrec (}B\texttt{) }R[\texttt{(}V\texttt{ (lambda (}x\texttt{) (abort }R[x]\texttt{)))}]\texttt{)}.\end{array}$$

So continuation procedures are represented as ordinary procedure expressions that `abort` with a value[5].

10.2 Rules for `abort`

Aborting means replacing the current continuation with the "top-level" read-eval-print loop (REPL) continuation. In the substitution model, this behavior is captured by two rewrite rules for `abort`. The first is

$$R[\texttt{(abort }V\texttt{)}] \to \texttt{(abort }V\texttt{)},$$

and the second is, again, the same as the first in the context of the outer "environment" `letrec`:

$$\texttt{(letrec (}B\texttt{) }R[\texttt{(abort }V\texttt{)}]\texttt{)} \to \texttt{(letrec (}B\texttt{) (abort }V\texttt{))}.$$

[5]For expressions with `abort` to have the same behavior in Scheme as in the Substitution Model, an `abort` procedure has to be installed into Scheme. This can be accomplished by defining `abort` to have some dummy value in the initial environment, `(define abort 'dummy)`, and then evaluating

```
(call-with-current-continuation (lambda (repl) (set! abort repl)))
```

in the top level read-eval-print-loop. Also, this Substitution Model uses the more succint name `call/cc`, so the definition

```
(define call/cc call-with-current-continuation)
```

should be evaluated.

Note that real Scheme evaluation doesn't distinguish aborting with a value from successfully returning that value. For example, evaluating

```
(call/cc (lambda (c) (c 1)))
```

in the Scheme REPL returns the value `1`.

Applying the preceding continuation rewrite rules to this example, we see that

```
(call/cc (lambda (c) (c 1)))
  → ((lambda (c) (c 1)) (lambda (x) (abort x)))
  → (letrec ((c (lambda (x) (abort x))) (c 1)))
  → ((lambda (x) (abort x)) 1)
  → (letrec ((x 1)) (abort x))
  → (abort 1).                                        (2)
```

However, there is no rule for reducing `(abort 1)`. It seems we should add an additional control rule:

$$\texttt{(abort } V\texttt{)} \to V. \qquad (\texttt{abort}\text{-elim})$$

This would allow us to reach the desired conclusion from (2) that

$$\texttt{(call/cc (lambda (c) (c 1)))} \downarrow \texttt{1}.$$

Indeed, if we include (`abort`-elim) as a Substitution Model rewrite rule, rewriting will accurately reflect Scheme evaluation using `call/cc`.

But as natural as (`abort`-elim) may seem, it is inconsistent with the theory of observational congruence developed up to this point. Most fundamentally, the control-context independence property of Corollary 3.4 will fail if we adopt (`abort`-elim). For example, consider the control context

$$R_1 ::= \texttt{(+ 1 [ ])}.$$

Since `(abort 1)` → `1` by (`abort`-elim), if control-context independence held, we could conclude

$$R_1[\texttt{(abort 1)}] \to R_1[\texttt{1}] \to \texttt{2}.$$

But in fact,

$$\begin{aligned} R_1[\texttt{(abort 1)}] &\to \texttt{(abort 1)} && \text{(by the first } \texttt{abort} \text{ rule)} \\ &\to \texttt{1} && \text{(by (}\texttt{abort}\text{-elim)).} \end{aligned}$$

So to avoid this failure of control-context independence, we will *not* include (`abort`-elim) in the Substitution Model. Instead, we treat aborted values as values. Namely, we modify the value grammar so that if V is an ⟨syntactic-value⟩ according to the current grammar (including `abort` as a ⟨procedure-constant⟩), then `(abort` V`)` will now also be an ⟨syntactic-value⟩.

This creates a minor problem: there is no context for distinguishing different aborted values, and consequently all aborted values are observationally congruent. Since this conclusion is not what we intend, we extend the definition of observational distinguishability so that expressions that abort with distinct printable values are considered distinguishable.

Problem 17. **(a)** Describe a context, C, such that C[`(abort 3)`] converges, but C[`3`] diverges.

(b) Suppose C is a context in kernel Scheme without `call/cc` or `abort`. Prove that

$$C[\texttt{(abort } V_1\texttt{)}]\downarrow \quad \text{iff} \quad C[\texttt{(abort } V_2\texttt{)}]\downarrow$$

for all syntactic values, V_1, V_2.

10.3 Control-stack Garbage Collection

Scheme interpreters and compilers typically represent the current continuation by a stack of further operations to be performed, called the *control-stack*. When the current stack is replaced during application of a continuation procedure, it can be garbage collected. This is reflected to a degree by the rewrite rule for `abort`, but that rule would cause the stack to be garbage collected *after* application of a continuation procedure had returned a value. But in fact, the old stack can be garbage collected even before the argument of the application is evaluated.

For example, we can add *control-stack garbage collection* rules:

$$R[\texttt{(abort } M\texttt{)}] \to \texttt{(abort } M\texttt{)},$$
$$\texttt{(letrec (}B\texttt{) R[(abort } M\texttt{)])} \to \texttt{(letrec (}B\texttt{) (abort } M\texttt{))},$$

where M is any expression, not necessarily a syntactic value. As with environment garbage collection, the uniqueness of final values continues to hold with this additional kind of control stack garbage collection.

10.4 Problems with the Rules for `call/cc`

Rejecting rule (`abort`-elim) ensures control-context independence for expressions involving the `abort` constant alone, but control-context independence still fails in the presence of the `call/cc` rules above. As a consequence, rewriting no longer preserves observational congruence, namely, Lemma 9.4 no longer holds, and this causes most of the theory of observational equivalence established so far to fail for Scheme with `call/cc`.

To illustrate this, note that by (2) in the Section 10.2

$$\texttt{(call/cc (lambda (c) (c 1)))} \downarrow \texttt{(abort 1)}. \tag{3}$$

If Control-context Independence held, then (3) implies that

$$R[\texttt{(call/cc (lambda (c) (c 1)))}] \xrightarrow{*} \texttt{R[(abort 1)]},$$

and hence that

$$R[\texttt{(call/cc (lambda (c) (c 1)))}] \downarrow \texttt{(abort 1)}, \tag{4}$$

for any control context, R. But letting

$$R_1 ::= \texttt{(+ 1 [ ])},$$

we have

$$
\begin{array}{ll}
\multicolumn{2}{l}{R_1[\texttt{(call/cc (lambda (c) (c 1)))}]} \\
= & \texttt{(+ 1 (call/cc (lambda (c) (c 1))))} \\
\to & \texttt{((lambda (c) (c 1)) (lambda (x) (abort (+ 1 x))))} \\
 & \qquad\qquad\text{(by the first \texttt{call/cc} rule)} \\
\xrightarrow{*} & \texttt{((lambda (x) (abort (+ 1 x))) 1)} \\
\xrightarrow{*} & \texttt{(abort (+ 1 1))} \\
\to & \texttt{(abort 2)}. \qquad (5)
\end{array}
$$

But (5) contradicts (4), and we conclude that Control-context Independence fails.

10.5 Better rules for `call/cc` [Optional]

M. Felleisen and R. Hieb in "The Revised report on the syntactic theories of sequential control and state," *Theoretical Computer Science*, Elsevier, 103 (1992) 235-271, consider a λ-calculus with a control operator similar to `call/cc` and develop a substitution model for their calculus whose rewriting rules do preserve observational congruence. We adapt Felleisen's and Hieb's techniques to Scheme.

The troubles described in the previous section arise from having a procedure constant `abort` that always jumps to "top-level." So the constant `abort` that would have been created by the rules of Section 10.2 will be replaced by an "abort" *variable* that is bound in the `call/cc` application. We do this by including, in addition to an environment `letrec`, a top-level `call/cc` explicitly representing the continuation for the expression evaluation. Now a simple control rule specified as $P \to T$ also serves as an abbreviation for a rule of the form

$$
\begin{array}{l}
\texttt{(call/cc (lambda (}\mathit{abort}\texttt{) } R[P]\texttt{))} \\
\quad \to \ \texttt{(call/cc (lambda (}\mathit{abort}\texttt{) } R[T]\texttt{))},
\end{array}
$$

where R parses as a control context, P parses as an ⟨immediate-redex⟩, *abort* is a fresh variable, and free occurrences of *abort* in $R[P]$ are treated as ⟨procedure-constant⟩'s. Each simple control rule also has a version applicable to top-level `call/cc`'s with the environment `letrec`:

$$
\begin{array}{l}
\texttt{(letrec (}B\texttt{) (call/cc (lambda (}\mathit{abort}\texttt{) } R[P]\texttt{)))} \\
\quad \to \ \texttt{(letrec (}B\texttt{) (call/cc (lambda (}\mathit{abort}\texttt{) } R[T]\texttt{)))}.
\end{array}
$$

By extending the meaning of simple rules in this way, we allow all of the previous simple rules to be applied immediately within the top-level `call/cc`.

We also need versions of the previous non-simple rules, *e.g.*, the first `lambda` *bind an arg* rule now has a top-level `call/cc` version:

$$
\begin{array}{l}
\texttt{(letrec (}B\texttt{) (call/cc (lambda (}\mathit{abort}\texttt{) } R[\texttt{((lambda (}x_1 \ \ldots\texttt{) } M\texttt{) } V_1 \cdots\texttt{)}]\texttt{)))} \\
\quad \to \ \texttt{(letrec (}B\texttt{ (}x_1\, V_1\texttt{)) (call/cc (lambda (}\mathit{abort}\texttt{) } R[\texttt{((lambda (}\ldots\texttt{) } M\texttt{) } \cdots\texttt{)}]\texttt{)))},
\end{array}
$$

and similarly for the other environment rules.

An "abort" variable gets its own rule:

$$\begin{aligned}&\texttt{(call/cc (lambda (}\mathit{abort}\texttt{) } R[\texttt{(}\mathit{abort}\ M\texttt{)}]\texttt{))} \\ &\quad\to\ \texttt{(call/cc (lambda (}\mathit{abort}\texttt{) } M\texttt{))}.\end{aligned} \qquad \text{(abort)}$$

Expressions with top-level `call/cc`'s get created by a new rule for `call/cc` application:

$$\begin{aligned}&R[\texttt{(call/cc } V\texttt{)}] \\ &\quad\to\ \texttt{(call/cc (lambda (}\mathit{abort}\texttt{) } R[\texttt{(}V\texttt{ (lambda (}x\texttt{) (}\mathit{abort}\ R[x]\texttt{)))}]\texttt{))}\end{aligned} \qquad (\texttt{call/cc}.1)$$

where $R \neq \langle\text{hole}\rangle$, and *abort* and x are fresh variables.

We also need a rule to put an outermost `call/cc` into top-level form. Namely, if V is a syntactic value that is not a lambda-expression, we have

$$\texttt{(call/cc } V\texttt{)} \to \texttt{(call/cc (lambda (}\mathit{abort}\texttt{) (}V\ \mathit{abort}\texttt{)))} \qquad (\text{top } \eta)$$

where *abort* is a fresh variable.

Finally, there is another `call/cc` application rule for expressions that already have a top-level `call/cc`:

$$\begin{aligned}&\texttt{(call/cc (lambda (}\mathit{abort}\texttt{) } R[\texttt{(call/cc } M\texttt{)}]\texttt{))} \\ &\quad\to\ \texttt{(call/cc (lambda (}\mathit{abort}\texttt{) } R[\texttt{(}M\texttt{ (lambda (}x\texttt{) (}\mathit{abort}\ R[x]\texttt{)))}]\texttt{))},\end{aligned} \qquad (\texttt{call/cc}.2)$$

where x is a fresh variable. Notice how (`call/cc`.2) incorporates the inner `call/cc` into the top-level one.

Each of these rules also has an environment form allowing the rule to be applied immediately within the environment `letrec`; to avoid clutter, but we have not written out these environment form versions.

We conjecture that these rules both accurately reflect Scheme evaluation with `call/cc`, satisfy Control-context Independence, and preserve observational equivalence. However, this remains to be carefully checked.

A Scheme Syntax in BNF

The following Backus-Naur Form (BNF) grammars describe the main constructs of Scheme[6]. In these grammars, superscript "*" indicates zero or more occurrences of a grammatical phrase, and superscript "$^+$" indicates one or more occurrences.

[6]For the official, full Scheme grammar, see the Revised^5 Scheme Manual available on the web at:

`http://www.schemers.org/Documents/Standards/R5RS`

A.1 The Functional Kernel

⟨expression⟩ ::= ⟨self-evaluating⟩ | ⟨symbol⟩ | ⟨variable⟩
  | ⟨procedure⟩ | ⟨let-form⟩ | ⟨if⟩ | ⟨combination⟩

⟨self-evaluating⟩ ::= ⟨numeral⟩ | ⟨boolean⟩ | ⟨string⟩ | . . .
⟨numeral⟩ ::= `0` | `-1` | `314159` | . . .
⟨boolean⟩ ::= `#t` | `#f`
⟨string⟩ ::= `"hello there"` | . . .

⟨symbol⟩ ::= `(quote` ⟨identifier⟩`)`
⟨identifier⟩ ::= identifiers that are not ⟨self-evaluating⟩
⟨variable⟩ ::= identifiers that are neither ⟨self-evaluating⟩, ⟨keyword⟩
  ⟨procedure-constant⟩, nor ⟨pairing-operator⟩

⟨keyword⟩ ::= `quote` | `lambda` | ⟨let-keyword⟩ | `if`

⟨procedure⟩ ::= ⟨nonpairing-procedure⟩
⟨nonpairing-procedure⟩ ::= ⟨lambda-expression⟩ | ⟨procedure-constant⟩
⟨lambda-expression⟩ ::= `(lambda (`⟨formals⟩`)` ⟨expression⟩`)`
⟨formals⟩ ::= ⟨variable⟩* (Note: all ⟨variable⟩'s must be distinct.)
⟨procedure-constant⟩ ::= `+` | `-` | `*` | `/` | `=` | `<` | `atan` | `string=?` | . . .
  | `number?` | `symbol?` | `procedure?` | `string?` | `boolean?` | `eq?` | . . .

Note that no "side-effect" procedures such as `display`, `set-car!`, `string-set!` nor "pairing" operators `list`, `cons` are included among the procedure constants. Also, as a further reflection of our explanation why mutable lists have been omitted from the Substitution Model, we restrict application of `eq?` to values that are ⟨symbol⟩'s.

⟨let-form⟩	::=	`(`⟨let-keyword⟩ `(`⟨binding⟩*`)` ⟨expression⟩`)`
		(Note: all variables bound by the ⟨let-form⟩ must be distinct.)
⟨let-keyword⟩	::=	`letrec`
⟨binding⟩	::=	`(`⟨variable⟩ ⟨init⟩`)`
⟨init⟩	::=	⟨expression⟩
⟨if⟩	::=	`(if` ⟨test⟩ ⟨consequent⟩ ⟨alternative⟩`)`
⟨test⟩	::=	⟨expression⟩
⟨consequent⟩	::=	⟨expression⟩
⟨alternative⟩	::=	⟨expression⟩
⟨combination⟩	::=	`(`⟨operator⟩ ⟨operand⟩*`)`
⟨operator⟩	::=	⟨expression⟩
⟨operand⟩	::=	⟨expression⟩

A.2 Functional (Immutable) Lists

⟨expression⟩	::=	. . . | ⟨nil⟩
⟨nil⟩	::=	`(list)`
⟨procedure⟩	::=	. . . | ⟨pairing-operator⟩
⟨pairing-operator⟩	::=	`cons` | `list`
⟨procedure-constant⟩	::=	. . . | `car` | `cdr` | `map` | `apply` | `null?` | `pair?`
		(Note: `cons` and `list` are *not* considered to be procedure constants.)
⟨lambda-expression⟩	::=	. . . | `(lambda` ⟨variable⟩ ⟨expression⟩`)`

A.3 The Full Kernel

⟨expression⟩	::=	. . . | ⟨begin⟩ | ⟨assignment⟩
⟨keyword⟩	::=	. . . | `begin` | `set!`
⟨begin⟩	::=	`(begin` ⟨expression⟩$^+$`)`
⟨assignment⟩	::=	`(set!` ⟨variable⟩ ⟨expression⟩`)`

A.4 Derived Syntax

⟨keyword⟩	::=	… | `define`
⟨body⟩	::=	⟨internal-defines⟩⟨expression⟩$^+$
⟨define⟩	::=	… | `(define` ⟨variable⟩ ⟨expression⟩`)`
		| `(define (`⟨variable⟩ ⟨formals⟩`)` ⟨body⟩`)`
⟨internal-defines⟩	::=	⟨define⟩* (Note: all defined variables must be distinct.)
⟨let-form⟩	::=	… | `(`⟨let-keyword⟩ `(`⟨binding⟩*`)` ⟨body⟩`)`
⟨let-keyword⟩	::=	… | `let` | `let*`
⟨lambda-expression⟩	::=	… | `(lambda (`⟨formals⟩`)` ⟨body⟩`)` | `(lambda` ⟨variable⟩ ⟨body⟩`)`

⟨expression⟩	::=	… | ⟨cond⟩ | ⟨and⟩ | ⟨or⟩ | ⟨quoted⟩
⟨keyword⟩	::=	… | `cond` | `else` | `and` | `or`

⟨cond⟩	::=	`(cond` ⟨clause⟩⟨clause⟩*`)`
⟨clause⟩	::=	`(`⟨test⟩ ⟨expression⟩*`)` | `(else` ⟨expression⟩$^+$`)`
⟨and⟩	::=	`(and` ⟨expression⟩*`)`
⟨or⟩	::=	`(or` ⟨expression⟩*`)`

⟨s-expression⟩	::=	⟨identifier⟩ | ⟨self-evaluating⟩ | `(`⟨s-expression⟩*`)`

A.5 Continuations

⟨procedure-constant⟩ ::= … | `call/cc` | `abort`