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.

Scheme consists of a kernel language and a variety of additional syntactic forms. The Revised[5] Scheme Manual describes how to translate ("desugar") these additional forms—referred to as "derived syntax"—into Kernel Scheme. These translations can easily be described with rewrite rules and could be incorporated directly into a Substitution Model. But since the desugaring rules raise no new issues beyond those arising in the kernel rules, we will not consider them further.

2 Control Syntax for the Substitution Model

A simplified Backus-Naur Form (BNF) grammar for Scheme, including derived syntax, is given in an Appendix. In these notes, the unmodified word "Scheme" will henceforth refer to Kernel Scheme as given by this grammar.

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]:

⟨syntactic-value⟩ ::= ⟨immediate-value⟩ | ⟨list-value⟩ | ⟨nonlist-pair-value⟩
⟨immediate-value⟩ ::= ⟨self-evaluating⟩ | ⟨symbol⟩ | ⟨procedure⟩
⟨list-value⟩ ::= ⟨nil⟩ | `(list` ⟨syntactic-value⟩$^+$`)`
⟨nonlist-pair-value⟩ ::= `(cons` ⟨syntactic-value⟩ ⟨immediate-value⟩`)`
| `(cons` ⟨syntactic-value⟩ ⟨nonlist-pair-value⟩`)`

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

2.2 Control Contexts for Scheme

An important technical property of the Substitution Model is that the rewrite rule to apply at any evaluation step will 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 a single occurrence of a variable, [ ], called the "hole," indicating the subexpression that an evaluator would begin working on. We'll illustrate with an example before giving the formal definitions.

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

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

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

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

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

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

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

The fact that Scheme will apply the operator `pair?` is captured by the fact that P is an *immediate redex* consisting of a procedure constant, `pair?`, applied to a value, namely, the ⟨list-value⟩ `(list (list) 'a)`.

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

⟨control-context⟩ ::= ⟨hole⟩
| `(if` ⟨control-context⟩ ⟨expression⟩ ⟨expression⟩`)`
| `(begin` ⟨control-context⟩ ⟨expression⟩$^+$`)`
| `(set!` ⟨variable⟩ ⟨control-context⟩`)`
| `(`⟨let-keyword⟩
`(`⟨value-binding⟩* `(`⟨variable⟩ ⟨control-context⟩`)` ⟨binding⟩*`)`
⟨expression⟩`)`
| `(`⟨syntactic-value⟩* ⟨control-context⟩ ⟨expression⟩*`)`

⟨hole⟩ ::= [ ]

⟨value-binding⟩ ::= `(`⟨variable⟩ ⟨syntactic-value⟩`)`

Note that the hole in a control context is never within the *body* of a lambda or letrec expression, though it may appear within a nonvalue init of a letrec binding.

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

⟨immediate-redex⟩ ::= ⟨variable⟩
| `(if` ⟨syntactic-value⟩ ⟨expression⟩ ⟨expression⟩`)`
| `(`⟨let-keyword⟩ `(`⟨value-binding⟩*`)` ⟨expression⟩`)`
| `(`⟨nonpairing-procedure⟩ ⟨syntactic-value⟩*`)`
| `(begin` ⟨expression⟩`)`
| `(begin` ⟨syntactic-value⟩ ⟨expression⟩*`)`
| `(set!` ⟨variable⟩ ⟨syntactic-value⟩`)`

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

`(letrec (`⟨value-binding⟩*`)` N`)`

for some ⟨expression⟩, N. This outermost `letrec` of an environment form expression is called the environment `letrec` of the expression. We sometimes use the notation Env(N) to indicate an expression of the form above.

A *final value* is a ⟨syntactic-value⟩ or an environment form Env(⟨syntactic-value⟩).

Definition 2.4. Let M be a Scheme expression in environment form, R be a control context, and P be an immediate redex. Then M is said to *control-parse into R and P* iff M is of the form Env($R[P]$).

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

Proof. Suppose $M =$ Env(N). We proceed by structural induction on N. If N is:

- [⟨self-evaluating⟩, ⟨symbol⟩, or ⟨procedure⟩], then N is a ⟨syntactic-value⟩, and M is a final value. So we must show that M cannot be control-parsed. But it follows directly from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, that the only way a ⟨syntactic-value⟩ can be of the form $R[K]$ for any control-context, R, is if $R =$ ⟨hole⟩, which means K must be N. But R, N is not a control-parse of M because N is a ⟨syntactic-value⟩, and ⟨syntactic-value⟩'s are not ⟨immediate-redex⟩'s. So M is not control-parsable.
- [a variable] then the only possibility is $R =$ ⟨hole⟩ and $P = N$.
- [a combination] then if,
  - [the operator or some operand is not a ⟨syntactic-value⟩], then N is of the form

    ($V_0 \ldots$ K $M_0 \ldots$)

    where $V_0 \ldots$ is a (possibly empty) sequence of syntactic values, K is not a syntactic value, and $M_0 \ldots$ is a (possibly empty) sequence of expressions. So N is neither a

syntactic value nor an immediate redex. Now it follows directly from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, that any control context, R, such that $N = R[L]$ for some L, must be of the form $R_1[R']$ where $R_1 = (V_0 \ldots \ \langle\text{hole}\rangle \ M_0 \ldots)$ and R' is some control context such that $R'[L] = K$. But by induction, $K = R_2[P]$ for a unique ⟨control-context⟩, R_2, and ⟨immediate-redex⟩, P. Hence, N uniquely control-parses into $R_1[R_2]$ and P.

– [the operator and all operands are ⟨syntactic-value⟩'s], then N is of the form $(op\ V_0 \ldots)$ where $V_0 \ldots$ is a (possibly empty) sequence of syntactic values and *op* is a syntactic value. By induction, there is no control-parse for *op* or any V_i. Now from the definitions of ⟨syntactic-value⟩ and ⟨control-context⟩, we see that the only remaining possible control-parse of M must be with $R = \langle\text{hole}\rangle$ and $P = N$. If *op* is not a ⟨pairing-operator⟩, then N is an immediate redex, and so ⟨hole⟩, N is the uniquely determined control-parse, as required. On the other hand, if *op* is a ⟨pairing-operator⟩, then N is a value, not an immediate redex, so M is a final value that is not control-parsable, as required in this case.

• [etc.] The remaining cases are left to the reader.

□

Problem 2. Complete the proof of Lemma 2.5

3 Scheme Rewrite Rules

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

We state only the rules that apply to expressions in environment form. These rules implicitly specify the rules for expressions not in environment form by the following convention:

Definition 3.1. If M is not in environment form, then $M \to N$ is a Substitution Model rule iff

1. `(letrec () M)` → `(letrec () N)` is a Substitution Model rule, or
2. N is in environment form and `(letrec () M)` $\to N$ is a Substitution Model rule.

3.1 Simple Control Rules

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

Definition 3.2. A *simple control rule* is a rewrite rule of the form

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

where P an ⟨immediate-redex⟩ called the *immediate redex of the rule*, and T is called the *immediate contractum*. A simple control rule with immediate redex, P, and immediate contractum, T, will be referred to for short as "the simple control rule $P \to T$."

Note that by the convention of Definition 3.1, the simple control rule $P \to T$ also implicitly defines a rule

$$R[P] \to R[T]$$

when $R[P]$ is not in environment form.

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

- *`if`:*

$$\begin{aligned} \texttt{(if \#f }M\texttt{ }N\texttt{)} &\to N \\ \texttt{(if }V\texttt{ }M\texttt{ }N\texttt{)} &\to M, \qquad\qquad \text{for } V \neq \texttt{\#f} \end{aligned}$$

- *`lambda` no args:*

$$\begin{aligned} \texttt{((lambda () }M\texttt{))} &\to M \\ \texttt{((lambda }x\texttt{ }M\texttt{))} &\to \texttt{(letrec ((}x\texttt{ }\langle\text{nil}\rangle\texttt{)) }M\texttt{)} \end{aligned}$$

- *`lambda` bind an arg:*

$$\begin{aligned} \texttt{((lambda (}x_1\texttt{ ...) }M\texttt{) }V_1\cdots\texttt{)} &\to \texttt{(letrec ((}x_1\texttt{ }V_1\texttt{)) ((lambda (...) }M\texttt{) }\cdots\texttt{))} \\ \texttt{((lambda }x\texttt{ }M\texttt{) }V_1\ldots\texttt{)} &\to \texttt{(letrec ((}x\texttt{ (list }V_1\ldots\texttt{))) }M\texttt{)} \end{aligned}$$

- *`begin`:*

$$\begin{aligned} \texttt{(begin }M\texttt{)} &\to M \\ \texttt{(begin }V\texttt{ }\langle\text{expression}\rangle^+\texttt{)} &\to \texttt{(begin }\langle\text{expression}\rangle^+\texttt{)} \end{aligned}$$

- *`procedure?`:*

$$\begin{aligned} \texttt{(procedure? }V\texttt{)} &\to \texttt{\#t} \qquad\qquad \text{for } V \text{ a } \langle\text{procedure}\rangle, \\ \texttt{(procedure? }V\texttt{)} &\to \texttt{\#f} \qquad\qquad \text{for other } V. \end{aligned}$$

- *builtin operations:*

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

- *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, S_1, S_2 are ⟨identifier⟩'s.

- *lists:*

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

- *pairs:*

$$\begin{aligned}
\texttt{(pair? (cons } V_1 \ V_2\texttt{))} &\rightarrow \texttt{\#t} \\
\texttt{(car (cons } V_1 \ V_2\texttt{))} &\rightarrow V_1 \\
\texttt{(cdr (cons } V_1 \ V_2\texttt{))} &\rightarrow V_2
\end{aligned}$$

  where V_2 is an ⟨immediate-value⟩ or ⟨nonlist-pair-value⟩.

3.2 Environment Rules

The following rules model lookups and updates of the environment.

- *instantiation:*

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

- *assignment:*

$$\begin{aligned}
&\texttt{(letrec (}B_1 \texttt{ (}x\ V_1\texttt{) } B_2\texttt{) } R[\texttt{(set! } x \ V_2\texttt{)}]\texttt{)} \\
&\quad \rightarrow \ \texttt{(letrec (}B_1 \texttt{ (}x\ V_2\texttt{) } B_2\texttt{) } R[\texttt{(quote set!-done)}]\texttt{)}
\end{aligned}$$

- *nested* `letrec`*:*

$$\texttt{(letrec (}B_1\texttt{) } R[\texttt{(letrec (}B_2\texttt{) } M\texttt{)}]\texttt{)} \ \rightarrow \ \texttt{(letrec (}B_1\ B_2\texttt{) } R[M]\texttt{)}.$$

3.3 Rules for 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, very general feature of this kind, called `call-with-current-continuation`, or `call/cc` for short.

The process of applying `call/cc` involves creation of a new *continuation procedure* which describes how the evaluation would continue if a value was returned by the `call/cc` application. 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. In particular, the continuation procedure of an immediate redex `(call/cc` V`)` in an expression R[`(call/cc` V`)`] would be

`(lambda (`x`) (return-to-repl` $R[x]$`))`.

So continuation procedures are represented as ordinary procedure expressions that can abort an evaluation and return a value directly to the "top-level" read-eval-print loop.

This is the idea behind the following control rules for the procedure constants `call/cc` and `return-to-repl`[2]:

$$\texttt{(letrec (}B\texttt{)}\ R[\texttt{(return-to-repl}\ V\texttt{)}]\texttt{)} \to \texttt{(letrec (}B\texttt{)}\ V\texttt{)}, \qquad \text{(abort)}$$

$$\texttt{(letrec (}B\texttt{)}\ R[\texttt{(call/cc}\ V\texttt{)}]\texttt{)} \to \texttt{(letrec (}B\texttt{)}\ R[\texttt{(}V\ \texttt{(lambda (}x\texttt{) (return-to-repl}\ R[x]\texttt{)))}]\texttt{)}. \qquad \text{(call/cc)}$$

where x is fresh.

3.4 Unique Rewriting

An examination of the Substitution Model rewrite rules reveals that an environment form expression, M, matches the redex of a rule only if M control-parses. Moreover, whether it matches the redex of a rule is uniquely determined by the immediate redex of the control-parse of M. Combining this observation with the Unique Control Parsing Lemma 2.5, we conclude:

Lemma 3.3. *(**Unique Matching**) There is at most one Scheme Substitution Model rewrite rule whose pattern matches an environment form expression M, and if there is such a rewrite rule, its match is unique.*

Definition 3.4. For Scheme expressions M, N, we write $M \to N$ to indicate that M matches the redex of a rule and N correspondingly matches the contractum of the rule. That is, M rewrites to N by one application of a Scheme Substitution Model rewrite rule. We write $M \not\to$ to mean that M does not match the redex of any rule.

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

```
(call-with-current-continuation (lambda (repl) (set! return-to-repl 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.

So the Unique Matching Lemma immediately implies

Theorem 3.5. *(**Unique Rewriting**) For every Scheme expression, M, there is at most one N such that $M \to N$.*

Definition 3.6. We write $M \not\to$ when an expression, M, does not match the redex of any Substitution Model rewrite rule, that is, no rule applies to M.

A final value, F, does not control-parse, and therefore does not match the redex of any rule. Expressions which *do* control-parse, but nevertheless do not match the redex of any rule, correspond to dynamic errors. We can characterize these expressions explicitly:

Definition 3.7. An *error combination* is

- an expression, M, of the form `(`⟨procedure-constant⟩ ⟨syntactic-value⟩*`)` such that $M \not\to$, or of the form `(`V ⟨syntactic-value⟩*`)` where V is a ⟨syntactic-value⟩ but not a ⟨procedure⟩.
- ⟨lambda-expression⟩'s applied to the wrong number of arguments, in particular, expressions of the form

  `((lambda ()` ⟨expression⟩`)` ⟨syntactic-value⟩$^+$`)`,

  or

  `((lambda (`⟨variable⟩$^+$`)` ⟨expression⟩`))`.

  (Note that other cases of ⟨lambda-expression⟩'s applied to the wrong number of arguments will rewrite to one of the forms above.)
- `(cons` ⟨syntactic-value⟩*`)` if there are not exactly two values to which the `cons` is applied.

Clearly, error combinations cause immediate dynamic errors in Scheme, for example,

```
(/ 1 0)
(+ 'a 0)
(0 1)
(call/cc list +)
((lambda () (f 1)) 2)
(cons 1)
```

are error combinations.

Definition 3.8. An *error* `letrec` is an expression of the form

`(letrec ((`⟨value-binding⟩* `(`⟨variable⟩ $R[x]$`)` ⟨binding⟩*`)` ⟨expression⟩`)`

where the indicated occurrence of x is bound by one of the `letrec` bindings.

An arbitrary Scheme expression, M, is *an immediate error* if it control-parses with an immediate redex that is an error combination or an error `letrec`.

An expression, M is a *lookup error* if it control-parses with an immediate redex that is of the form x, or `(set!` x ⟨syntactic-value⟩`)`, where x is a free variable.

We distinguish errors caused by lookup of an undefined variable because, in contrast to immediate errors, an expression that causes a lookup error may do something useful in an extended environment where the undefined variable is assigned a value.

Theorem 3.9. *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.*

3.5 Control-context Independence

If a Scheme subexpression starts to be evaluated, then evaluation of that subexpression continues until a value for the subexpression is returned. This is called "sequential" evaluation, as opposed, say, to "parallel" evaluation where evaluation steps may alternate between disjoint subexpressions. A technical property that implies sequentiality is given in the following corollary.

Corollary 3.10. *(Control-context Independence) Restrict the Scheme Substitution Model Rules to exclude the (call/cc) and (abort) rules. If*

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

then

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

by the same rule.

Problem 3. Prove Corollary 3.10. *Hint:* Use Problem 1 and Unique Control Parsing.

The (call/cc) and (abort) rules are not control-context independent. To illustrate this, note that

$$\texttt{(return-to-repl 1)} \to \texttt{1}.$$

If independence held, we could conclude that for $R = \texttt{(+ 1 }\langle\text{hole}\rangle\texttt{)}$,

$$R[\texttt{(return-to-repl 1)}] \to R[\texttt{1}] \to \texttt{2}.$$

But in fact, by the (abort) rule,

$$R[\texttt{(return-to-repl 1)}] \to \texttt{1}.$$

This failure of control-context independence could repaired simply by treating `(return-to-repl` V`)` as a value and modifying the (abort) rule so that `return-to-repl` does not get deleted. But even with this repair, failures of independence arise from the (call/cc) rule. For example,

$$\texttt{(call/cc (lambda (k) (k 0)))} \xrightarrow{*} \texttt{(return-to-repl 0)}.$$

If control-context independence held, we could conclude that

$R[\texttt{(call/cc (lambda (k) (k 0)))}] \xrightarrow{*} R[\texttt{(return-to-repl 0)}] \to \texttt{(return-to-repl 0)}$,

but in fact

$R[\texttt{(call/cc (lambda (k) (k 0)))}] \xrightarrow{*} R[\texttt{(return-to-repl } R[\texttt{0}]\texttt{)}] \to^* \texttt{(return-to-repl 1)}$.

This kind of failure of independence is more complicated to repair, but it can be done. In the next set of notes we'll present revised rules (call/cc) and (abort) rules for which context independence does hold. So in particular, even with `call/cc`, Scheme evaluation would switch back and forth between disjoint subexpressions to evaluate them in parallel.

4 The Variable Convention

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

Problem 4. A Substitution Model rewrite rule *preserves the Variable Convention*, if, when M satisfies the Variable Convention and rewrites to N by an application of the rule, then N also satisfies the Convention. 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 wind up with two bindings in the outermost, "environment," `letrec` of N. For example,

Example 4.2.

```
(letrec ((x 1)) ((lambda (x) x) 2))
  → (letrec ((x 1)) (letrec ((x 2)) ((lambda () x))))
  → (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 that do not occur in any of the expressions at hand. This then forces us to consider simultaneous substitution of terms for several variables.

Definition 4.3. For functions, f, g, define the function $f \leftarrow g$ to be the function, h, such that

$$h(u) ::= \begin{cases} g(u) & \text{if } u \in \text{domain}\,(g)\,, \\ f(u) & \text{otherwise.} \end{cases}$$

Definition 4.4. A *substitution* is a mapping, σ, from a finite set of ⟨variable⟩'s to ⟨expression⟩'s. The notation

$$[x_1, \ldots, x_n := M_1, \ldots, M_n]$$

describes the substitution, σ, with domain $\{x_1, \ldots, x_n\}$ such that $\sigma(x_i) = M_i$ for $i = 1, \ldots n$.

Every substitution, σ, defines a mapping, $[\sigma]$, from ⟨expression⟩'s to ⟨expression⟩'s defined by structural induction on ⟨expression⟩'s:

- [⟨self-evaluating⟩, ⟨procedure-constant⟩, or ⟨pairing-operator⟩]

$$c[\sigma] ::= c.$$

- [⟨symbol⟩]

$$\texttt{(quote name)}[\sigma] ::= \texttt{(quote name)}.$$

- [⟨variable⟩]

$$x[\sigma] ::= \begin{cases} \sigma(x) & \text{if } x \in \text{domain}\,(\sigma), \\ x & \text{otherwise.} \end{cases}$$

- [⟨if⟩]

$$\texttt{(if}\ T\ C\ A\texttt{)}[\sigma] ::= \texttt{(if}\ T[\sigma]\ C[\sigma]\ A[\sigma]\texttt{)}.$$

- [⟨combination⟩]

$$\texttt{(}M_1\ \ldots\ M_n\texttt{)}[\sigma] ::= \texttt{(}M_1[\sigma]\ \ldots\ M_n[\sigma]\texttt{)}$$

- [⟨lambda-expression⟩]

$$\texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ N\texttt{)}[\sigma] ::= \texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ N[\sigma']\texttt{)}$$

  where σ' is the restriction of σ to the variables other than $x_1 \ldots x_n$.

- [⟨let-form⟩]

$$\texttt{(letrec ((}x_1\ M_1\texttt{)...(}x_n\ M_n\texttt{))}\ \ N\texttt{)}[\sigma] \quad ::= \\ \texttt{(letrec ((}z_1\ \ M_1[\sigma \leftarrow \rho]\texttt{)...(}z_n\ \ M_n[\sigma \leftarrow \rho]\texttt{))}\ \ N[\sigma \leftarrow \rho]\texttt{)}$$

  where

$$\rho \quad = \quad [x_1, \ldots, x_n := z_1, \ldots, z_n] \tag{1}$$

  for distinct fresh variables $z_1, \ldots, z_n$.

Definition 4.5. 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 5. Write a BNF grammar for ⟨context⟩.

Definition 4.6. The binary relation α*-equivalence* is the smallest equivalence relation, $=_\alpha$, on ⟨expression⟩'s such that

- for ρ from (1) above,

$$\texttt{(lambda (}x_1 \ldots x_n\texttt{)}\ N\texttt{)} =_\alpha \texttt{(lambda (}z_1 \ldots z_n\texttt{)}\ \ N[\rho]\texttt{)},$$
$$\texttt{(letrec ((}x_1\ M_1\texttt{)}\ldots\texttt{(}x_m\ M_m\texttt{))}\ \ N\texttt{)} =_\alpha \texttt{(letrec ((}z_1\ \ M_1[\rho]\texttt{)...(}z_n\ \ M_n[\rho]\texttt{))}\ \ N[\rho]\texttt{)}.$$

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

Problem 6. **(a)** Prove that if $M_1 =_\alpha M_2$, then M_1 is a ⟨syntactic-value⟩ iff M_2 is also a ⟨syntactic-value⟩; likewise M_1 is an ⟨immediate-redex⟩ iff M_2 is an ⟨immediate-redex⟩.

(b) 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 7. 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.

Definition 5.1. 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 $N \not\rightarrow$. 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 Theorem 3.5 implies that if $M \xrightarrow{*} N$ and $N \not\rightarrow$, then N is uniquely determined. So we have:

Lemma 5.2. *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 5.3. *If M diverges, then so does* `(letrec (`B`)` $R[M]$`)`.

Problem 8. Prove Lemma 5.3. *Hint:* Prove it for Scheme without `call/cc`, so you can use control-context independence. Then see if you can modify your proof to cover the `call/cc` rules as well.

Problem 9. **(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 Theorem 3.5 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`)`.

Finally, we can garbage-collect an empty environment:

$$\texttt{(letrec ()}\ M\texttt{)} \to M.$$

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 Theorem 3.5 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.*

Problem 10. Prove Theorem 6.1. *Hint:* 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 $N_1 =_{gc} N_2$.*

A Scheme Syntax in BNF

The following Backus-Naur Form (BNF) grammars describe the main constructs of Scheme[3]. In these grammars, superscript "*" indicates zero or more occurrences of a grammatical phrase, and superscript "$^+$" indicates one or more occurrences.

A.1 The Functional Kernel

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

⟨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⟩,
⟨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.

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

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

⟨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⟩*`)`

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

A.5 Continuations

⟨procedure-constant⟩ ::= ... | `call/cc` | `return-to-repl`