Massachusetts Institute of Technology
6.844, Spring '05: Computability Theory of and with Scheme
Prof. Albert R. Meyer

Course Notes 8
April 21
revised April 29, 2005, 134 minutes

Scheme Equations (partial draft)

1 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 observation of the result of an evaluation that is even more fundamental than its numerical value, is whether any value is returned at all. We choose this more fundamental observation as the basis for our definition of distinguishability.

Definition 1.1. 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[1], 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 problem demonstrates that observing returned self-evaluating or symbol values, rather than just observing convergence, yields the same observational equivalence relation on Scheme expressions.

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

Problem 1. Let S denote an expression that is $\langle$self-evaluating$\rangle$ or a $\langle$symbol$\rangle$.

(a) Show that if $M \equiv N$, and $M \downarrow S$, then $N \downarrow S$.

(b) Show that if M and N are distinguishable, then for any S, there is a context, C_S, such that the one of $C_S[M]$ and $C_S[N]$ has final value S and the other one diverges.

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

$C[M]$ has a $\langle$self-evaluating$\rangle$ or $\langle$symbol$\rangle$ value iff $C[N]$ has the same $\langle$self-evaluating$\rangle$ or $\langle$symbol$\rangle$ value,

for all contexts, C.

We remark that 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 the hole in a context is supposed to parse as a variable. But when an $\langle$identifier$\rangle$ is quoted, it parses as a $\langle$symbol$\rangle$ not a variable. This technicality is important, as indicated in the next problem.

Problem 2. Explain why no two expressions would be observationally equivalent if contexts could have a quoted hole.

Many statements about Scheme properties are implicitly about observational equivalences. For example, when we say that bound variable names are "private" in Scheme, we mean precisely that α-equivalent expressions are observationally equivalent.

Problem 3. Prove that α-equivalence implies observational equivalence.

Similarly, we know that in an environment `letrec`, the order of the bindings it doesn't matter. "Doesn't matter" more precisely means observational equivalence:

Problem 4.

$$
\begin{aligned}
&\texttt{(letrec ((}x_1\,V_1\texttt{)...(}x_i\,V_i\texttt{)...(}x_j\,V_j\texttt{)...)}\,M\texttt{)}\\
&\quad\equiv\ \texttt{(letrec ((}x_1\,V_1\texttt{)...(}x_j\,V_j\texttt{)...(}x_i\,V_i\texttt{)...)}\,M\texttt{)}
\end{aligned}
$$

Hint: Similar to the proof that α-equivalence implies observational equivalence.

In the next sections we'll develop methods for proving some more sophisticated observational equivalences.

2 Revised (call/cc) Rules

We observed in Notes 7 that control-context independence does not hold for the (call/cc) and (abort) rules given there. This property is an important one for reasoning about Scheme evaluation, so we describe how to modify the rules so that they become control-context independent.

2.1 Continuation-form Rules

The idea behind the new rules is that constant `return-to-repl` that appears in the rules of Notes 7 will be replaced by a *variable* that is bound in the argument of an outermost, "continuation," `call/cc` application. [2]

The new set of Substitution Model rules will first of all include all the rules from Notes 7, with the exception of the three rules mentioning the procedure contant, `return-to-repl`, namely, the (abort), (call/cc), and control-stack garbage collection rules.

To achieve the effect of `return-to-repl`, we add special versions of the rules for expressions in "continuation form." An expression is in *continuation form*, Cnt(M), if it is of the form

$$\texttt{(call/cc (lambda (}\mathit{return}\texttt{) } M\texttt{))}.$$

If return is not a free variable of M, we say the continuation is *superfluous*.

Now for every rule $M \to N$ adopted from Notes 7, we aim to add a continuation-form version Cnt(M) $\to$ Cnt(N). This should ensure that any expression, M, will evaluate as usual inside a continuation form as long as none of the three omitted rules is needed.

For example, for each simple control rule of the form

$$\texttt{(letrec (}B\texttt{) } R[P]\texttt{)} \to \texttt{(letrec (}B\texttt{) } R[T]\texttt{)}, \tag{1}$$

the idea would be to add a new rule

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

This is almost right, but for one technicality: in the continuation form version of the rule, we want occurrences of the continuation *variable*, *return*, to be parsed in the same way as the procedure *constant*, `return-to-repl`. So for each rule of the form (1), we actually add a rule

$$\begin{aligned}&\texttt{(call/cc (lambda (}\mathit{return}\texttt{) (letrec (}\widehat{B}\texttt{) } \widehat{R}[\widehat{P}]\texttt{)))}\\ &\quad\to\ \texttt{(call/cc (lambda (}\mathit{return}\texttt{) (letrec (}\widehat{B}\texttt{) } \widehat{R}[T]\texttt{)))}.\end{aligned}$$

where $\widehat{R}$ is a context that would be a control-context if all occurrences of the continuation variable were parsed as syntactic values, namely, $\widehat{R}[\mathit{return} := V]$ is a real control context for some $\langle$syntactic-value$\rangle$, V. (Note that the parsing rules for control contexts don't distinguish one syntactic value from another, so $\widehat{R}[\mathit{return} := V]$ is a control-context for some V iff $\widehat{R}[\mathit{return} := \texttt{+}]$ is a

[2]This approach was first developed by 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, where they consider a λ-calculus with a control operator similar to `call/cc` and develop control-context independent rewriting rules for their calculus.

control-context.) Likewise, $\widehat{P}$ is an expression such that $\widehat{P}[\mathit{return} := \texttt{+}]$ is an $\langle$immediate-redex$\rangle$[3], and $\widehat{B}$ would be a sequence of $\langle$value-binding$\rangle$'s if *return* was parsed as a $\langle$syntactic-value$\rangle$. Similarly, we need versions of the other Substitution model rules that will apply in the presence of outer `call/cc`'s. For example, the instantiation rule now has a continuation version[4]

$$\begin{aligned}&\texttt{(call/cc (lambda (}\mathit{return}\texttt{) (letrec (}\widehat{B_1}\ \texttt{(}x\ \widehat{V}\texttt{)}\ \widehat{B_2}\texttt{)}\ \widehat{R}[x]\texttt{)))}\\ &\quad\to\ \texttt{(call/cc (lambda (}\mathit{return}\texttt{) (letrec (}\widehat{B_1}\ \texttt{(}x\ \widehat{V}\texttt{)}\ \widehat{B_2}\texttt{)}\ \widehat{R}[\widehat{V}]\texttt{)))}.\end{aligned}$$

We also extend the set of final values to include continuation forms. Namely, expressions of the form

$$\texttt{(call/cc (lambda (}\mathit{return}\texttt{)}\ \widehat{F}\texttt{))}$$

are now considered final values, where $\widehat{F}$ would be a final value in the sense of Notes 7 if *return* is parsed as a $\langle$procedure$\rangle$, that is, $\widehat{F}[\mathit{return} := \texttt{+}]$ is an expression that is a $\langle$syntactic-value$\rangle$ or is of the form Env($\langle$syntactic-value$\rangle$).

2.2 Revised `call/cc` Rules

The rule replacing the (abort) rule will simply be the old rule in continuation form:

$$\begin{aligned}&\texttt{(call/cc (lambda (}\mathit{return}\texttt{) (letrec (}\widehat{B}\texttt{)}\ \widehat{R}[\texttt{(}\mathit{return}\ \widehat{V}\texttt{)}]\texttt{)))}\\ &\quad\to\texttt{(call/cc (lambda (}\mathit{return}\texttt{) (letrec (}\widehat{B}\texttt{)}\ \widehat{V}\texttt{)))},\end{aligned}\qquad\text{(return)}$$

There is also a version of this rule without the environment `letrec`:

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

Likewise, the (call/cc) rule of Notes 7 will be replaced by its continuation form

$$\begin{aligned}&\texttt{(call/cc (lambda (}\mathit{return}\texttt{) (letrec (}\widehat{B}\texttt{)}\ \widehat{R}[\texttt{(call/cc }\widehat{V}\texttt{)}]\texttt{)))}\\ &\quad\to\texttt{(call/cc (lambda (}\mathit{return}\texttt{)}\\ &\qquad\qquad\texttt{(letrec (}\widehat{B}\texttt{)}\\ &\qquad\qquad\quad\widehat{R}[\texttt{(}\widehat{V}\ \texttt{(lambda (}k\texttt{) (}\mathit{return}\ \widehat{R}[k]\texttt{)))}]\texttt{)))},\end{aligned}\qquad\text{(call/cc.1)}$$

where k is a fresh variable. Also, to handle `call/cc` applications within expressions that are not in continuation form, we include

$$\begin{aligned}&\texttt{(letrec (}B\texttt{)}\ R[\texttt{(call/cc }V\texttt{)}]\texttt{)}\\ &\quad\to\texttt{(call/cc (lambda (}\mathit{return}\texttt{)}\\ &\qquad\qquad\texttt{(letrec (}B\texttt{)}\\ &\qquad\qquad\quad R[\texttt{(}V\ \texttt{(lambda (}k\texttt{) (}\mathit{return}\ R[k]\texttt{)))}]\texttt{))},\end{aligned}\qquad\text{(call/cc.2)}$$

[3]Note that the rewrite rules do distinguish among $\langle$syntactic-value$\rangle$'s, so we can't use `#f` instead of `+` here, though we could use any other $\langle$procedure$\rangle$ in place of `+`. This ensures, for example, that

$$\texttt{(call/cc (lambda (}\mathit{return}\texttt{) (procedure? }\mathit{return}\texttt{)))}\to\texttt{(call/cc (lambda (}\mathit{return}\texttt{) \#t))}$$

[4]The rule for `set!` needs a technical modification in the special case when the variable being set is *return*. The details are not important and are omitted.

where k and $return$ are fresh variables. There are also versions, which we have not written out, of each of these rules without the environment `letrec`.

Notice that the (return) rule removes applications of $return$, and the (call/cc.1-2) rules replace `call/cc` applications by applications of $return$.

Finally, we add a garbage collection rule for continuations:

$$\text{Cnt}(M) \to M,$$

when the outer continuation is superfluous.

In the rest of these notes, "Substitution Model rules" will mean the set of rules determined by the rules above. Like the rules of Notes 7, this new set of Substitution Model rules also provides a complete and accurate model of Kernel Scheme evaluation.

2.3 Control-context Merge Independence

In contrast to the rules of Notes 7, the current Substitution Model rules are control-context independent in the following slightly broader sense:

Definition 2.1. The evaluations of expressions M and N are said to *merge*, in symbols, $M \Downarrow N$, iff there is an expression, J, such that $M \xrightarrow{*} J$ and $N \xrightarrow{*} J$.

Lemma 2.2. *If $M \to N$ by any of the* `call/cc` *rules above, then*

$$R[M] \Downarrow R[N]$$

for any control context, R.

In fact, we can use this concept of merged evaluations to state a simplified version of control-context independence for the full set of Substitution Model Rules:

Corollary 2.3. *(Control-context Merge Independence) If $M \to N$ by any of the rules in the Substitution Model, then*

$$R[M] \Downarrow R[N]$$

for any control context, R.

Problem 5. Verify Lemma 2.2 and Corollary 2.3.

Problem 6. Prove that if $M \Downarrow N$, then

$$R[M] \Downarrow R[N]$$

for any control context, R.

3 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.

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:

```
(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 3.1. A *multihole context*, C, is a Scheme expression except that the $\langle$hole$\rangle$, 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 $\langle$hole$\rangle$ 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 $\langle$hole$\rangle$ 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 control context independence, 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 3.2. 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 ⟨nonlist-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 $\mathbf{k}_n ::= k_1, \ldots, k_n$ such that $k_i \in \{$*nonval, proc, constant, struct*$\}$ indicates the kind of M_i, for $1 \leq i \leq n$.

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 3.5 below summarizes the way contexts can be rewritten. To state it, we need to generalize control contexts to have multiple holes.

Definition 3.3. 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

$$\texttt{(letrec ((n } [\,]_4\texttt{)) (+ } [\,]_1 \texttt{ 2 (* } [\,]_2\ [\,] \texttt{ (- n } [\,]_3\texttt{))))}$$

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.

Definition 3.4. A multihole *environment context* for a kind pattern, $\mathbf{k}$, is a context, E, with a designated main hole, such that $E[\mathbf{M}_n][\,]$ is a single hole environment context for all $\mathbf{M}_n$ with kind pattern $\mathbf{k}$. An *environment control context*, F, for $\mathbf{k}$ is an environment context for $\mathbf{k}$ whose body is a control context for $\mathbf{k}$.

Lemma 3.5. *(**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, $1 \le i \le n$, 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*

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

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

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

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

The proof of the Standard Context Lemma 3.5 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.

4 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 4.1. *If $M\uparrow$ and $N\uparrow$, then $M \equiv N$.*

To prove Theorem 4.1, 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 3.5, and Theorem 4.1 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 3.5.1 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 3.5.4a applies, namely $E[M] \xrightarrow{*} R[M]$ for some control context, R. But as we observed in Notes 7, $R[M]\uparrow$ by control context independence, 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 4.2. *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 3.5.1 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 3.5.4 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 control control independence, $R[M, N][M]$ 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 4.3. *[Operational Extensionality] If M_1 and M_2 are closed expressions of procedure kind and*

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

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

$$M_1 \equiv M_2.$$

Problem 7. Prove Corollary 4.3.

Problem 8. Prove that if R is a control context and no free variable of M occurs in R, then

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

when x is a fresh variable. 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 4.4. *If $M \rightarrow N$, then $M \equiv N$.*

Problem 9. **(a)** Let M, N be Scheme expressions such that $M \rightarrow N$. 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, using control context independence.

(b) Use part (a) to complete a proof of Lemma 4.4.

(c) Conclude that if $M \Downarrow N$, then $M \equiv N$.

Several of the list rewriting rules can be used to simplify expressions even when not all the arguments are values. These simplifications arise from some simple observational equivalences:

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

Some other simplifications follow from the fact that 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,

$$\begin{aligned}\texttt{(car (list } M \texttt{ y))} &\equiv \texttt{M},\\ \texttt{(* x x)} &\equiv \texttt{(* 2 x)}\end{aligned}$$

Problem 10. **(a)** Describe an expression, M such that `(+` M M`)` and `(* 2` M`)` are observationally distinguishable. 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 3.5 also allows us to deduce interesting observational equivalences that do not simply follow by rewriting subexpressions or equating divergent ones.

Lemma 4.5. *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. □

Problem 11. 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 4.5. 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.

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, `use`, and the value of `x` is initially even, then it will still be even if and when the external procedure returns a value:

Example 4.6.

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

Problem 12. Prove the equivalence in Example 4.6. **Warning**: this may be hard.