How to Evaluate Functional Scheme Programs

Conventions

A law of the form

$$P \ = \ Q$$

where P and Q are program fragments (expressions or sequences of expressions) means that P and Q are logically equivalent; one can be substituted for the other without changing the meaning of the program. Hence, = means exactly what it means in high school algebra. In addition, every law

$$P \ = \ Q$$

has the property that Q is "closer" to an answer (assuming one exists) than P.

Unless otherwise stated, the integer N is restricted to the range $N \geq 0$.

Evaluating Expressions

Evaluating an expression means finding a value for an expression according to the following laws. Some well-formed expressions (according to the Laws of Syntax) do not have a value according to these rules.

Values are Values, are Values, ...

Values are the answers produced by computations. No evaluation is required (or possible!).

The Laws of `cond`

If the first condition (test expression) is a value, then one of the following rules applies:

$$\texttt{(cond [\#f } E_1\texttt{]} \ \ldots \ \texttt{[else } E_N\texttt{])} \ = \ \texttt{(cond} \ \ldots \ \texttt{[else } E_N\texttt{])}$$

$$\texttt{(cond [}V \ \ E_1\texttt{]} \ \ldots \ \texttt{[else } E_N\texttt{])} \ = \ E_1 \qquad \text{if } V \neq \texttt{\#f}$$

If the first condition is not a value, then it must be evaluated before the laws of `cond` can be used.

The Laws of Application

Given an application consisting of values

$$(V_1 \ V_2 \ \ldots \ V_N)$$

there are two complementary sets of laws that specify how to evaluate the application. If the “head” value V_1 is a primitive procedure, then there is a large table of laws for directly reducing the application to a value. You know most of them from *grammar school*; the remainder are decribed (implicitly) in the course lecture notes and Dybvig’s book. The following paragraph shows the subset of these laws for `car, cdr, cons, cons?`.

Primitive applications: If (and only if) U, V, and W are values, then:

$$\begin{array}{l} (\texttt{car}\ (\texttt{cons}\ U\ V)) \ = \ U \\ (\texttt{cdr}\ (\texttt{cons}\ U\ V)) \ = \ V \\ (\texttt{cons?}\ \ (\texttt{cons}\ U\ V)) \ = \ \texttt{\#t} \\ (\texttt{cons?}\ \ W) \ = \ \texttt{\#f} \qquad\qquad \text{if } W \neq (\texttt{cons}\ U\ V) \end{array}$$

where V is a list value.

If the “head” value V_1 is a `lambda`-expression

$$(\texttt{lambda}\ (name_1 \ \ldots \ name_N) \ E)$$

where $name_1$, …, $name_N$ are names and E is an expression, then the following rule specifies the next step in evaluating the application:.

`lambda` applications:

$$((\texttt{lambda}\ (name_1 \ \ldots \ name_N) \ E) \ V_1 \ \ldots \ V_N) \ = \ E_{[V_1 \text{ for } name_1]\ldots[V_N \text{ for } name_N]}$$

where the notation $E_{[Value \text{ for } name]}$ means E with all free occurrences of *name* *safely* replaced by $Value$.[1]

If the “head” value is not a procedure, then evaluation sticks; there are no rules for reducing applications of non-procedures.

If one or more of the expressions in an application

$$(E_1 \ E_2 \ \ldots \ E_N)$$

[1]Locally bound variables in E must be renamed if they clash with free variables in $V_1, \ldots, V_N$.

are not values, then they must be evaluated before the laws of application can be used. In Scheme, no order is specified for evaluating these expressions. In our hand-evaluations, we will always evaluate the leftmost unevaluated expression E_i first.

Evaluating Programs and Names

The preceding section gives laws for evaluating Scheme expressions in the absence of program definitions. But Scheme programs have the form

`(define` $name_1$ E_1`)`
`(define` $name_2$ E_2`)`
`. . .`
`(define` $name_N$ E_N`)`
E

where $name_1$, $name_2$, ..., $name_N$ are *names* and E_1, E_2, ..., E_N, E are *expressions* using Scheme primitives and the defined names $name_1$, $name_2$, ..., $name_N$. The expression E is called the *body* of the program and each expression E_k is called the *body* of the definition

`(define` $name_k$ E_k`)`

Given a program of the form

`(define` $name_1$ V_1`)`
`(define` $name_2$ V_2`)`
`. . .`
`(define` $name_N$ V_N`)`
E

we can evaluate the expression E as described above with the added provision that the names $name_1$, $name_2$, ..., $name_N$ have values V_1, V_2, ..., V_N, respectively. More precisely, the program evaluation law of Scheme says that the program

`(define` $name_1$ V_1`)`
`(define` $name_2$ V_2`)`
`. . .`
`(define` $name_N$ V_N`)`
E

reduces in one step to

```
(define name₁ V₁)
(define name₂ V₂)
. . .
(define name_N V_N)
E′
```

provided that E reduces in one step to E' given that $name_1$, $name_2$, ..., $name_N$ have values V_1, V_2, ..., V_N, respectively.

We still have to address the issue of evaluating the definition bodies $E_1, \ldots, E_N$ that are not values. A program

$$
\begin{array}{l}
\texttt{(define } name_1\ V_1\texttt{)}\\
.\quad .\quad .\\
\texttt{(define } name_{k-1}\ V_{k-1}\texttt{)}\\
\texttt{(define } name_k\ E_k\texttt{)}\\
.\quad .\quad .\\
\texttt{(define } name_N\ E_N\texttt{)}\\
E
\end{array}
$$

where $N > 0$, $k > 0$, and V_1, ..., V_{k-1} are values and E_k, ..., E_N are expressions, reduces in one step to

$$
\begin{array}{l}
\texttt{(define } name_1\ V_1\texttt{)}\\
.\quad .\quad .\\
\texttt{(define } name_{k-1}\ V_{k-1}\texttt{)}\\
\texttt{(define } name_k\ E'_k\texttt{)}\\
.\quad .\quad .\\
\texttt{(define } name_N\ E_N\texttt{)}\\
E
\end{array}
$$

provided that

$$
\begin{array}{l}
\texttt{(define } name_1\ V_1\texttt{)}\\
.\quad .\quad .\\
\texttt{(define } name_{k-1}\ V_{k-1}\texttt{)}\\
E_k
\end{array}
$$

reduces in one step to:

```
(define name_1 V_1)
. . .
(define name_{k-1} V_{k-1})
E'_k
```

In essence, these laws force us to evaluate the bodies of all definitions in sequential order before evaluating the body of the program.

Rules for `local`

To evaluate programs containing `local`, we need to introduce the concept of *promotion*. Given an expression of the form

`(local [(define` n_1 E_1`)` ... `(define` n_N E_N`)] E)`

where $E_1, \ldots, E_N$ are expressions, we must convert the local definitions of the names $n_1, \ldots, n_N$ to global definitions of new names $n'_1, \ldots, n'_N$, renaming all bound occurrences of $n_1, \ldots, n_N$, and evaluate the transformed expression E in the context of the new definitions. This conversion process is called the *promotion* or *flattening* of a `local` expression. The new names $n'_1, \ldots, n'_N$ must be chosen so that they are distinct from all other names in the program.

Let

```
(define name_1 V_1)
. . .
(define name_{k-1} V_N)
E
```

be a program where the program body E has the form

$\mathcal{C}[L]$

where L is an expression

`(local [(define` n_1 E_1`)` ... `(define` n_N E_N`)] E)`

enclosed in the surrounding program text $\mathcal{C}[\ ]$ to form the expression E. Assume that no subexpressions in E to the left of the subexpression L can be reduced. Hence, L is the leftmost expression in the entire program that can be reduced. In this case, the surrounding text $\mathcal{C}[\,]$ is called the *evaluation context* of L.

Using the notation introduced above, we can describe the *promotion step* reducing the program by the following rule:

```
(define name_1 V_1)
. . .
(define name_{k-1} V_N)
C[(local [(define n_1 E_1) ... (define n_N E_N)] E)]
```

$=$

```
(define name_1 V_1)
. . .
(define name_{k-1} V_N)
(define n'_1 E1[n'_1 for n_1] ... [n'_N for n_N])
. . .
(define n'_N EN[n'_1 for n_1] ... [n'_N for n_N])
C[E[n'_1 for n_1] ... [n'_N for n_N]]
```

In other words, we simply replaced L by the body of L with $n_1, \ldots, n_N$ renamed and we added appropriate definitions for the new names in the sequence of `define` statements preceding the program body. Note that free occurences of the names $n_1, \ldots, n_N$ must be renamed in the expressions $E_1, \ldots, E_N$, as well as E.