Evaluating Functional Scheme Programs

Comp 210

Spring 2001

Contents

1 Conventions

Evaluating an expression means finding a value for that expression. We use a step-by-step process to repeatedly simplify an expression until it is so simple that it is a value. Evaluating a program means evaluating each of its expressions (all but the last of which are definitions) in turn.

A law of the form

$$P = Q$$

where P and Q are program fragments (expressions or sequences of expressions) means that P and Q have the same behavior; 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.

E, E_1, E_2, $\ldots$ are expressions. V, V_1, V_2, $\ldots$ are values. n, n_1, n_2, $\ldots$ are names (variables, placeholders). N is a non-negative integer.

2 Evaluating Expressions

Some syntactically well-formed expressions — such as (+ 'a 2), (*first empty*), (1 2), etc. — do not have a value according to these rules. We say that evaluation of such expressions "sticks".

2.1 Values are values, are values, ...

Values are the answers produced by computations. Every value is also an expression, but no evaluation is required (or possible!).

Some examples:

Value	Kind of Value
0	number (exact)
1/3	number (exact)
0.3333333333333333	number (inexact)
6.023e23	number (inexact)
true	boolean
false	boolean
'piston	symbol
"Scheme"	string
empty	list
(*cons* 'a *empty*)	list
(*list* 6 120)	list
+	built-in function (primitive operation)
(**lambda** (x) (+ x y))	user-defined function (**lambda** expression)

Note: The evaluation rules assume that the abbreviated syntax for Scheme function definitions has been be expanded so that the right hand sides of fucnction defintions are **lambda** expressions.

2.2 Conditionals

2.2.1 The Laws of if

If the test of an **if** expression is not a value, evaluate it to one by repeatedly applying the following rule

$$(\textbf{if}\ E_1\ E_2\ E_3) \ = \ (\textbf{if}\ E_1'\ E_2\ E_3) \qquad \text{if}\ \ E_1 \ = \ E_1'$$

If the test of an **if** expression is a value, the next step depends on whether the value is *true*. (Stylistically, you should use a boolean expression for the test, but Scheme permits any value and treats anything but *false* as true.)

$$(\textbf{if}\ \mathit{false}\ E_2\ E_3) \ = \ E_3$$
$$(\textbf{if}\ V\ E_2\ E_3) \ = \ E_2 \qquad \text{if}\ \ V \neq \mathit{false}$$

2.2.2 The Laws of cond

If the test of the first clause is not a value or **else**, evaluate it to a value.

$$(\text{cond } [E_1\ E_2] \ldots) \;=\; (\text{cond } [E_1'\ E_2] \ldots) \qquad \text{if } E_1 \;=\; E_1'$$

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

$$\begin{aligned}
&(\textbf{cond } [\mathit{false}\ E] \ldots) \;=\; (\textbf{cond } \ldots)\\
&(\textbf{cond } [V\ E] \ldots) \;=\; E \qquad \text{if } V \neq \mathit{false}\\
&(\textbf{cond } [\textbf{else } E] \ldots) \;=\; E
\end{aligned}$$

If there are no clauses — as in "(**cond**)" — the value is undefined. Generally, evaluation of a **cond** expression should result in selection of one of the clauses (and evaluation of its consequent expression.)

Here are some examples:

$$\begin{aligned}
(\textbf{cond } [(>\ 10\ 12)\ (+\ 7\ 8)]\ [\textbf{else } (*\ 6\ 4)]) \;&=\; (\textbf{cond } [\mathit{false}\ (+\ 7\ 8)]\ [\textbf{else } (*\ 6\ 4)])\\
&=\; (\textbf{cond } [\textbf{else } (*\ 6\ 4)])\\
&=\; (*\ 6\ 4)
\end{aligned}$$

$$\begin{aligned}
&(\textbf{cond } [\mathit{true}\ (+\ 7\ 8)]\ [\textbf{else } (*\ 6\ 4)]) \;=\; (+\ 7\ 8)\\
&(\textbf{cond } [\text{'foo}\ (+\ 7\ 8)]\ [\textbf{else } (*\ 6\ 4)]) \;=\; (+\ 7\ 8)
\end{aligned}$$

2.3 The Laws of Application

Evaluate each of the subexpressions of an application in turn from left to right.

$$(V_1\ \ldots\ V_{i-1}\ E\ \ldots\ ) \;=\; (V_1\ \ldots\ V_{i-1}\ E'\ \ldots\ ) \qquad \text{if } E \;=\; E'$$

Given an application consisting of values

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

we apply different laws depending on whether the head value V_1 is a primitive procedure or a user-defined procedure (a **lambda** expression). If the head value is not a procedure, then evaluation sticks; there are no rules for reducing applications of non-procedures. Some sticking expressions are (1 2), (1), and $((\mathit{cons}$ 'a $\mathit{empty})\ \mathit{empty})$.

2.3.1 Primitive applications

There is a large table of laws for directly reducing to a value the application of a primitive to a set of values. You know most of these rules from grammar school; the remainder are decribed (implicitly) in the course lecture notes and Kent Dybvig's book.

For instance, if (and only if) U is a value, V is a list value, and W is a non-list value, then:

$$\begin{aligned}
&(\mathit{first}\ (\mathit{cons}\ U\ V)) \;=\; U\\
&(\mathit{rest}\ (\mathit{cons}\ U\ V)) \;=\; V\\
&(\mathit{cons?}\ (\mathit{cons}\ U\ V)) \;=\; \mathit{true}\\
&(\mathit{cons?}\ W) \;=\; \mathit{false}
\end{aligned}$$

Examples:

$$
\begin{array}{l}
(\mathit{first}\ (\mathit{cons}\ 1\ \mathit{empty})) \ = \ 1 \\
(\mathit{rest}\ (\mathit{cons}\ 1\ \mathit{empty})) \ = \ \mathit{empty} \\
(\mathit{cons?}\ 1) \ = \ \mathit{false} \\
(\mathit{cons?}\ (\mathit{cons}\ 1\ \mathit{empty})) \ = \ \mathit{true} \\
(+\ 1\ 2) \ = \ 3
\end{array}
$$

If a primitive operation is applied to illegal inputs, then evaluation sticks and does not produce an answer. Some sticking expressions are $(\mathit{first}\ \mathit{empty})$, $(\mathit{rest}\ 1)$, and $(+\ \mathit{empty}\ 2)$.

2.3.2 lambda applications

If the head value in an application is a **lambda** expression

$$(\textbf{lambda}\ (\mathit{name}_1\ \ldots\ \mathit{name}_N)\ E)$$

where $\mathit{name}_1, \ldots, \mathit{name}_N$ are names and E is an expression, then the following rule specifies the next step in evaluating the application:.

$$((\textbf{lambda}\ (\mathit{name}_1\ \ldots\ \mathit{name}_N)\ E)\ V_1\ \ldots\ V_N) \ = \ E_{[V_1 \text{ for } \mathit{name}_1]\ldots[V_N \text{ for } \mathit{name}_N]}$$

where the notation $E_{[\mathit{Value} \text{ for } \mathit{name}]}$ means E with all free occurrences of *name safely* replaced by *Value*. (Locally bound variables in E must be renamed if they clash with free variables in $V_1, \ldots, V_N$.)

Examples:

$$
\begin{array}{l}
((\textbf{lambda}\ (x)\ (+\ x\ x))\ 7) \ = \ (+\ 7\ 7) \\
((\textbf{lambda}\ (f)\ (\textbf{lambda}\ (x)\ (f\ (f\ x))))\ (\textbf{lambda}\ (y)\ (+\ x\ y))) \\
\qquad \neq (\textbf{lambda}\ (x)\ ((\textbf{lambda}\ (y)\ (+\ x\ y))\ ((\textbf{lambda}\ (y)\ (+\ x\ y))\ x))) \\
((\textbf{lambda}\ (f)\ (\textbf{lambda}\ (x)\ (f\ (f\ x))))\ (\textbf{lambda}\ (y)\ (+\ x\ y))) \\
\qquad = (\textbf{lambda}\ (z)\ ((\textbf{lambda}\ (y)\ (+\ x\ y))\ ((\textbf{lambda}\ (y)\ (+\ x\ y))\ z)))
\end{array}
$$

3 Evaluating definitions

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

$$
\begin{array}{l}
(\textbf{define}\ n_1\ E_1) \\
(\textbf{define}\ n_2\ E_2) \\
\ldots \\
(\textbf{define}\ n_N\ E_N) \\
E
\end{array}
$$

where n_1, n_2, ..., n_N are names and E_1, E_2, ..., E_N, E are expressions using Scheme primitives and the defined names n_1, n_2, ..., n_N. The expression E is called the body of the program and each expression E_k is called the body of the definition $(\textbf{define}\ n_k\ E_k)$.

If the definition bodies E_k are all values

$$
\begin{array}{l}
(\textbf{define}\ n_1\ V_1)\\
(\textbf{define}\ n_2\ V_2)\\
\ldots\\
(\textbf{define}\ n_N\ V_N)\\
E
\end{array}
$$

then we evaluate the expression E as described above with the added provision that the names n_1, n_2, ..., n_N have values V_1, V_2, ..., V_N, respectively. More precisely, the program evaluation law says

$$
\begin{array}{l}
(\textbf{define}\ n_1\ V_1)\\
(\textbf{define}\ n_2\ V_2)\\
\ldots\\
(\textbf{define}\ n_N\ V_N)\\
E
\end{array}
=
\begin{array}{l}
(\textbf{define}\ n_1\ V_1)\\
(\textbf{define}\ n_2\ V_2)\\
\ldots\\
(\textbf{define}\ n_N\ V_N)\\
E'
\end{array}
\quad \text{if } E = E', \quad \begin{array}{l}\text{assuming } n_1, n_2, \ldots, n_N \text{ have}\\ \text{values } V_1, V_2, \ldots, V_N\text{, respectively}\end{array}
$$

If the definition bodies $E_1, \ldots, E_N$ that are not all values, use this rule:

$$
\begin{array}{l}
(\textbf{define}\ n_1\ V_1)\\
\ldots\\
(\textbf{define}\ n_{k-1}\ V_{k-1})\\
(\textbf{define}\ n_k\ E_k)\\
\ldots\\
(\textbf{define}\ n_N\ E_N)\\
E
\end{array}
=
\begin{array}{l}
(\textbf{define}\ n_1\ V_1)\\
\ldots\\
(\textbf{define}\ n_{k-1}\ V_{k-1})\\
(\textbf{define}\ n_k\ E'_k)\\
\ldots\\
(\textbf{define}\ n_N\ E_N)\\
E
\end{array}
\quad \text{if} \quad
\begin{array}{l}
(\textbf{define}\ n_1\ V_1)\\
\ldots\\
(\textbf{define}\ n_{k-1}\ V_{k-1})\\
E_k
\end{array}
=
\begin{array}{l}
(\textbf{define}\ n_1\ V_1)\\
\ldots\\
(\textbf{define}\ n_{k-1}\ V_{k-1})\\
E'_k
\end{array}
$$

These laws force us to evaluate the bodies of all definitions in sequential order before evaluating the body of the program.

3.1 Rules for local

To evaluate programs containing *local*, we need to introduce the concept of *promotion* (also called *flattening*). Given an expression of the form

$$(\textit{local}\ [(\textbf{define}\ n_1\ E_1)\ \ldots\ (\textbf{define}\ n_N\ E_N)]\ E)$$

we first 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 $n_1, \ldots, n_N$. Then we 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

$$
\begin{array}{l}
(\textbf{define}\ n_1\ V_1)\\
\ldots\\
(\textbf{define}\ n_{k-1}\ V_N)\\
E
\end{array}
$$

be a program where the program body E has the form

$$\mathcal{C}[L]$$

where L is an expression

$$(\mathit{local}\ \texttt{[}(\textbf{define}\ n_1\ E_1)\ \ldots\ (\textbf{define}\ n_N\ E_N)\texttt{]}\ E)$$

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:

$$\begin{array}{l}
(\textbf{define}\ n_1\ V_1) \\
\ldots \\
(\textbf{define}\ n_{k-1}\ V_N) \\
\mathcal{C}[(\mathit{local}\ \texttt{[}(\textbf{define}\ n_1\ E_1)\ \ldots\ (\textbf{define}\ n_N\ E_N)\texttt{]}\ E)] \\
\\
= \\
\\
(\textbf{define}\ n_1\ V_1) \\
\ldots \\
(\textbf{define}\ n_{k-1}\ V_N) \\
(\textbf{define}\ n'_1\ E_{1[n'_1\text{ for }n_1]\ \ldots\ [n'_N\text{ for }n_N]}) \\
\ldots \\
(\textbf{define}\ n'_N\ E_{N[n'_1\text{ for }n_1]\ \ldots\ [n'_N\text{ for }n_N]}) \\
\mathcal{C}[E_{[n'_1\text{ for }n_1]\ \ldots\ [n'_N\text{ for }n_N]}]
\end{array}$$

In other words, we 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.