Evaluating Functional Racket Programs

Corky Cartwright

Fall 2020

1 Conventions

Italicized metavariables will be used to stand for pieces of syntax within Racket programs as follows:

- E, E_1, E_2, ... are expressions.
- V, V_1, V_2, ... are values.
- Lower-case letters like f, g, h, n, s, t, u, v, x, y, z potentially augmented by numerical subscripts stand for variables. They are sometimes called "names". Some examples of such metavariables are f_1, g_2, h, n_5.
- N is a non-negative integer.

A sequence of items like $E_1, \ldots E_N$ indexed by consecutive integers beginning with 1, is empty if N is 0.

We will typically use the (possibly subscripted) metavariables f, g, and h to refer to variables that are bound to functions and the metavariables u, v, x, y, z to stand for variables that are bound to data values (objects) that are not functions. Sometimes we need metavariables that can be bound to either functions or non-functions. We will try to mention when this situation can arise.

A program to be evaluated consists a (possibly empty) sequence of definitions followed by an expression constructed from program-defined and primitive functions and variables:

`(define` f_1 `...)` ... `(define` f_N `...)` E

Note that the variables f_i can be bound to either functoins or non-functions. If the sequence of definitions is empty, the program to be evaluated is degenerate it simply consists of an expresson where there are no program-defined functions.

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 left-to-right (top-to-bottom) order. We will discuss these two notions or evaluation in more detail below.

In this formal account of the semantics of Racket, we consider a function definition of the form

`(define (`f x_1`, ...` x_N`)` E`)`

to be an abbreviation for the program text

`(define` f `((lambda (`x_1 ... x_N`)` E`)`

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 a value (assuming one exists) than P.

2 Evaluating Expressions

Some syntactically well-formed expressions — such as `(+ 'a 2)`, `(first empty)`, `(1 2)`, `(/ 1 0)`, etc. — do not have a value according to these rules. We say that evaluation of such expressions "sticks", which is a very simple, but admittedly crude approach to formalizing the notion of a run-time error.

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
`"Racket"`	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 Racket 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

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

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.)

(`if false` E_2 E_3) = E_3
(`if` V E_2 E_3) = E_2 if $V \neq$ `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.

(cond [E_1 E_2] ...) = (cond [E_1' E_2] ...) if $E_1 = E_1'$

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

(`cond` [`false` E] ...) = (`cond` ...)
(`cond` [V E] ...) = E if $V \neq$ `false`
(`cond` [`else` E] ...) = E

If there are no clauses — as in "`(cond)`" — the computation is stuck! A run-time error has occurred. 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:

```
(cond [(> 10 12) (+ 7 8)] [else (* 6 4)]) = (cond [false (+ 7 8)] [else
   (* 6 4)])
                                           = (cond [else (* 6 4)])
                                           = (* 6 4)
(cond [true (+ 7 8)] [else (* 6 4)]) = (+ 7 8)
(cond ['foo (+ 7 8)] [else (* 6 4)]) = (+ 7 8)
```

2.3 The Laws of Application

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

(V_1 ... V_{i-1} E ...) = (V_1 ... V_{i-1} E' ...) if $E = E'$

Note that the name f of a *program-defined* function is *not* a value. Hence, the first step in evaluating the application of an program-defined function is to replace the program defined function by its value which is the `lambda`-expression to which it is bound.

Given an application consisting of values

(V_1 V_2 ... V_N)

we apply different laws depending on whether the head value V_1 is a primitive function or a user-defined function (a `lambda` expression). If the head value is not a function, then evaluation sticks; there are no rules for reducing applications of non-procedures. Some sticking expressions are `(1 2)`, `(1)`, and `((cons 'a empty) 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 Racket Help Desk embedded in DrRacket.

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

```
(first (cons U V)) = U
(rest (cons U V)) = V
(cons?  (cons U V)) = true
(cons?  W) = false
```

Examples:

```
(first (cons 1 empty)) = 1
(rest (cons 1 empty)) = empty
(cons? 1) = false
(cons? (cons 1 empty)) = true
(+ 1 2) = 3
```

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

A chain of `cons` applications to values terminating in `empty` is clumsy syntax for a list value. For this reason, we introduce the alternate syntax

`(list` E_1 E_2 ... E_N`)`

as a more readable abbreviation for

`(cons` E_1 `(cons` E_2 ... `(cons` E_N `empty)` ... `))`

When evaluating applications of the variable arity function `list`, there are two cases If `list` is applied to no arguments, it reduces to the empty list `()`. Any application of `list` to a finite number $N > 0$ of Racket expressions

`(list` E_1 E_2 ... E_N`)`

then it simply abbreviates

`(cons` E_1 `(cons` E_2 ... `(cons` E_N `empty)` ... `))`

Of course we generally prefer to use `list` notation express the intermediate results of reducing an application of `list` to argument expressions.

2.3.2 lambda applications

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

(`lambda` (x_1 ... x_N) E)

where x_1, ..., x_N are variable names and E is an expression, then the following rule specifies the next step in evaluating the application:.

((`lambda` (x_1 ... x_N) E) V_1 ... V_N) = $E_{[V_1 \text{ for } x_1]...[V_N \text{ for } x_N]}$

where the notation $E_{[V \text{ for } x]}$ means E with all free occurrences[1] of x *safely* replaced by *Value*. (Locally bound variables in E must be renamed if they clash with free variables in $V_1, \ldots, V_N$. This anomaly is called *capturing free variables* and it is the bane of existence for logicians and programming language theorists.

Examples:

```
((lambda (x) (+ x x)) 7) = (+ 7 7)
((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ x y)))
    ≠ (lambda (x) ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) x)))
((lambda (f) (lambda (x) (f (f x)))) (lambda (y) (+ x y)))
    = (lambda (z) ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) z)))
```

[1] An occurrence of a variable is a *binding occurrence* if it appears as the variable defined in a Racket `define` construct or a parameter in a `lambda`-expression. A use occurrence of a variable is *free within a particular program fragment* P (expression or whole program) iff it not enclosed by a binding occurence of the same variable name in P.

3 Evaluating definitions

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

```
(define n_1 E_1)
(define n_2 E_2)
...
(define n_N E_N)
E
```

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 (`define` n_k E_k).

If the definition bodies E_k are all values

```
(define n_1 V_1)
(define n_2 V_2)
...
(define n_N V_N)
E
```

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. This situation prevails if all top-level `define` constructs bind variables to functions denoted by `lambda`-expressions. Hence, the only Racket programs that require evaluating the right-hand sides of `define` constructs are those with definitions (in the form of `define` constructs) that bind variables to expressions that are not functions!

If all of the top-level `define` constructs bind variables to values, the program evaluation law says

```
(define n_1 V_1)          (define n_1 V_1)
(define n_2 V_2)          (define n_2 V_2)
...                   =   ...                   if E = E',   assuming n_1, n_2, ..., n_N have
(define n_N V_N)          (define n_N V_N)                   values V_1, V_2, ..., V_N, respectively
E                         E'
```

If the definition bodies $E_1, \ldots, E_N$ that are not all values, we reduce the body of the first `define` that is not a value using this rule:

```
(define n_1 V_1)              (define n_1 V_1)
...                           ...
(define n_{k-1} V_{k-1})      (define n_{k-1} V_{k-1})
(define n_k E_k)          =   (define n_k E'_k)          where  (define n_1 V_1)          =  (define n_1 V_1)
...                           ...                              ...                             ...
(define n_N E_N)              (define n_N E_N)                 (define n_{k-1} V_{k-1})        (define n_{k-1} V_{k-1})
E                             E                                E_k                             E'_k
```

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 define-struct

For the sake of simplicity (and sanity), we will assume that all of the `define-struct` definitions for a program appear in a prelude that is preprocessed to augment the set of program values and primitive functions. Every `define-struct` definition

```
(define-struct n (n_1 ... n_k))
```

simply augments the primitive operations of Racket by the k-ary constructor `make-`n, the unary boolean function n`?`, and the accessors n`-`n_1, ..., n`-`n_k. Hence, the Racket program following such a prelude is evaluated like any other Racket program except of the expanded set of primitive operations.

3.2 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

(`local` [(`define` n_1 E_1) ... (`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

(`define` n_1 V_1)
...
(`define` n_{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` n_1 V_1`)`
...
`(define` n_{k-1} V_N`)`
$\mathcal{C}$`[(local [(define` n_1 E_1`)` ... `(define` n_N E_N`)] E)]`

=

`(define` n_1 V_1`)`
...
`(define` n_{k-1} V_N`)`
`(define` n'_1 $E_{1[n'_1 \text{ for } n_1] \ldots [n'_N \text{ for } n_N]}$`)`
...
`(define` n'_N $E_{N[n'_1 \text{ for } n_1] \ldots [n'_N \text{ for } n_N]}$`)`
$\mathcal{C}[E_{[n'_1 \text{ for } n_1] \ldots [n'_N \text{ for } n_N]}]$

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.