The Syntactic Semantics of Core Jam Programs

Corky Cartwright

Spring 2021

1 Notational Conventions

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

2 Core Jam Dialects

The syntax of Core Jam is rigorously defined in Assignment 3, but Assignment 3 includes the option of call-by-name/call-by-need semantics for let bindings which we exclude in Core Jam for the sake of simplicity.[1] For the sake of simplicity in hand evaluation (which is already clerically tedious), *we will only use call-by-value binding in* `let` *constructions in* Core Jam. On the other hand, will distinguish call-by-value and call-by-name interpretations for applications of `map`s and eager and lazy versions of `cons`. Restricting our interpretation of `let` to call-by-value binding reduces the total number of syntactic reduction rules for Core Jam,

We will only consider the syntactic evaluation of Jam programs, which are Jam expressions with no free variables. Nevertheless, our evaluation rules work for Jam expressions containing free variables if we simply add a rule that the evaluation of an unbound variable aborts with an error.

[1] Note that Assignment 4 introduces mutation distinguishing the meaning of call-by-name from call-by-need, but they are semantically equivalent in Assignment 3.

In Core Jam, the semantics for `let` is recursive. Hence, we do not identify `let` constructions with expansions into `map` applications. They have different semantics because `let` is recursive. As a result, we need separate reduction rules for `let` constructions.

All of our interesting hand evaluations involve programs where (*i*) the outermost construction is a `let` that introduces a collection of function definitions and (*ii*) the body is some expression that performs a computation using those defined functions. In other words, an interesting Core Jam program has the form

`let` f_1 `:=` E_1`;` ... f_N `:=` E_N`; in` E

where E_1, ..., E_N, E are Core Jam expressions. Note that the variables f_i can be bound to either functions or non-functions but will typically be bound to functions.

By distinguishing `let` constructions from the corresponding application of `map`s (required to support explicit recursion), we implicitly create what is effectively an environment of program function bindings accumulated as the the evaluated bindings of variables introduced in `let` constructions

3 Expression Evaluation

Evaluating a Core Jam 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 repeatedly evaluating the leftmost reducible sub-expression (any expression within the program) assuming there is no irreducible (also called "stuck") sub-expression preceding it. If there is a "stuck" sub-expression to the left of all reducible sub-expressions (a set which may be empty) in the expression being evaluated, the computation aborts because the stuck state prevents the expression from being reduced to a value. The evaluation process always operates on the leftmost reducible expression or stuck state.

3.1 The Reduction Laws

A law of the form

$$P \longrightarrow Q$$

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, $\longrightarrow$ means exactly what it means in high school algebra. In addition, every law

$$P \longrightarrow Q$$

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

3.2 Stuck 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 because there is no reduction law that matches such an expression. We say that evaluation of such expressions "sticks", which is a very simple, but perhaps crude approach to formalizing the notion of a run-time error.

3.3 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!). In Core Jam, the only values are: numbers, boolean constants, lists, and maps. Every map is a value.

Some examples:

Value	Kind of Value
`0`	number
`-7`	number
`true`	boolean
`false`	boolean
`empty`	list
`cons`	binary primitive function value
`cons?`	unary primitive function value
`=`	binary primitive function value written in infix notation
`cons(1,empty)`	list
`map x to x + y`	program-defined function

3.4 Infix abbreviations

The infix and prefix operators in Assignment 3 Jam are simply abbreviations for binary and unary function applications, respectively. Infix notation is a familiar convenience from mathematics that we often exploit in programming languages. (Lisp-like languages eschew these abbreviations because it complicates the process of syntactic transformation which is much simpler in Lisp-like languages because the concrete syntax is essentially abstract syntax.) Some examples of Jam infix notation are `7 + 5`, `7 > 5`, `-7`, and `empty = cons(1, empty)`.

3.5 Conditionals

3.5.1 The Laws of `if`

When the test of a Jam `if` expression is a value, the next step depends on whether the value is `true` or `false`. (If any other value appears in the test position for an `if` expression, that expression is a "stuck state".)

`if true then` E_1 `else` E_2 $\longrightarrow$ E_1
`if false then` E_1 `else` E_2 $\longrightarrow$ E_2

Note that if the test expression is a value V other than `true` or `false`, the `if` expression is a "stuck" state (an aborting error when in leftmost reducible position).

Here are some examples:

`if` $10 > 12$ `then` $7 + 8$ `else` $6 * 4$ $\longrightarrow$ `if false then` $7 + 8$ `else` $6 * 4$
`if` $12 > 10$ `then` $7 + 8$ `else` $6 * 4$ $\longrightarrow$ `if true then` $7 + 8$ `else` $6 * 4$

`if true then 7 + 8) else 6 * 4` $\longrightarrow$ `7 + 8`

`if 12 then 7 + 8 else 6 * 4` is a stuck state

3.5.2 The Laws of `|` and `&`

Let E be an arbitrary expression, B be an arbitrary boolean value, and W is an value that is not a boolean. Then

`true |` E $\longrightarrow$ `true`
`false |` B $\longrightarrow$ B
`false |` W is a stuck state
`false &` E $\longrightarrow$ `false`
`true &` B $\longrightarrow$ B
`true &` W is a stuck state

3.6 The Laws of Application

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 a `map`. If the head value is not a function (primitive function or a `map`), the application is "stuck". Some "stuck" expressions of this form are `1(2)`, `1()`, and `cons(1, empty)(empty)`. Recall that prefix and infix expressions are simply abbreviations for corresponding unary and binary primitive function applications.

3.6.1 Primitive applications

There is a large table of laws for directly reducing (to a value) the application of a primitive to a sequence of values. You know most of these rules from grammar school; the remainder are described (implicitly) in the course lecture notes. In addition, you should be familiar with all of the primitive functions in Core Jam because you implemented them in Assignment 3. If you are uncertain about a result in the table, you can use the reference interpreter to evaluate it, providing the answer.

For instance, given that `U`, $U_1, \ldots, U_n$ are values, `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 the primitive application is "stuck". Some sticking expressions are `(first empty)`, `(rest 1)`, and `(+ empty 2)`.

3.6.2 `map` Applications

There are two plausible semantics for the applications of map dubbed *call-by-value* and *call-by-name*. The former is much more common, so we will explicate it first.

Call-by-value law for map applications If the head value in an application is a `map` expression

`map` x_1, ..., x_N to E

where x_1, ..., x_N, ($N \geq 0$) are variable names and E is an expression, then the following rule specifies the next step in the call-by-value evaluation of the application:

(`map` x_1,...,x_N to E) (V_1,...,V_N) $\longrightarrow$ $E_{[V_1 \text{ for } x_1]...[V_N \text{ for } x_N]}$

where the notation $E_{[V \text{ for } x]}$ means E with all free occurrences[2] of x *safely* replaced by V.

Note: locally bound variables in E (introduced in either `let` or `map` constructions) 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. Fortunately, in the syntactic evaluation of Core Jam programs, we can prove that capture cannot happen. Hence, the qualification provided here is technically unnecessary. It is included here as a reminder that capture is a endemic problem in thed formalization of logics and programming languages.

Examples:

```
(map x to x + x) (7) ⟶ (+ 7 7)
(map f to map x to f(f(x))) (map y to 1 + y))
    ≠ map x to (map y to 1 + y) ((map y to 1 + y) x)))
((map (f) (lambda (x) (f (f x)))) (lambda (y) (+ x y)))
    ⟶ (lambda (z) ((lambda (y) (+ x y)) ((lambda (y) (+ x y)) z)))
```

Call-by-name law for map applications If the head value in an application is a `map` expression

`map` x_1, ..., x_N to E

where x_1, ..., x_N, ($N \geq 0$) are variable names and E is an expression, then the following rule specifies the next step in the call-by-name evaluation of the application:.

(map x_1,...,x_N to E) (E_1,...,E_N) $\longrightarrow$ $E_{[E_1 \text{ for } x_1]...[E_N \text{ for } x_N]}$

[2] An occurrence of a variable is a *binding occurrence* if it appears as the variable defined in a Jam `let` construct or a parameter in a `map`. 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 occurrence of the same variable name in P.

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

Note the similarity between these two rules (which are called call-by-value beta-reduction and call-by-name beta reduction); the only different is that the call-by-value rule is *not* applicable unless the argument expressions E_1, ..., E_N are *values*. The sample evaluation in the accompanying exercises, show how important this distinction is the definition of some simple functions like short-circuit "and" and "or" operations.

3.7 The Law for Evaluating the Bodies of `let` Constructions

The preceding section gives laws for evaluating Core Jam expressions in the absence of let constructions. But nearly all interesting Jam programs have the form b where n_1, n_2, ..., n_N are names and E_1, E_2, ..., E_N, E are expressions using Jam primitives and the defined names n_1, n_2, ..., n_N. The expression E is called the body of the `let` construction and each expression E_k is called the body of the binding n_k `:=` E_k`;`.

If the binding bodies E_k are all values

```
let n1  :=  V1;
    n2  :=  V2;
    ...
    nN  :=  VN;
in  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 already prevails if all `let` constructs bind variables to functions denoted by `map` constructions. Hence, the only `let` bindings where the bodies (right-hand sides) require evaluation are those that bind variables to expressions that are not functions.