lec11: Lλ

Jana Dunfield

March 1, 2022

1 Lλ(big-step)

Try the exercises in lec10 before reading these notes!

Recall these evaluation rules from lec10:

$$\dfrac{}{(\texttt{Lam}\ x\ e) \Downarrow (\texttt{Lam}\ x\ e)}\ \text{eval-lam} \qquad \dfrac{e_1 \Downarrow (\texttt{Lam}\ x\ e_{\text{body}}) \qquad e_2 \Downarrow v_2 \qquad [v_2/x]e_{\text{body}} \Downarrow v}{(\texttt{Call}\ e_1\ e_2) \Downarrow v}\ \text{eval-call}$$

The rule eval-call is perfectly reasonable, but there is a simpler alternative. Rather than evaluate e_2 to v_2 and substitute v_2, we can substitute e_2 without evaluating it:

$$\dfrac{e_1 \Downarrow (\texttt{Lam}\ x\ e_{\text{body}}) \qquad [e_2/x]e_{\text{body}} \Downarrow v}{(\texttt{Call}\ e_1\ e_2) \Downarrow v}\ \text{eval-call-by-name}$$

This rule behaves the same, in the sense that if eval-call derives $(\texttt{Call}\ e_1\ e_2) \Downarrow v_v$ and eval-call-by-name derives $(\texttt{Call}\ e_1\ e_2) \Downarrow v_n$, the values v_v and v_n will be the same.

While the two rules have the same *extensional* behaviour—they give the same result—their *intensional* behaviour is quite different. For example, if we call the function $(\texttt{Lam}\ x\ (+\ x\ x))$ that doubles its argument, and the argument is $(-\ 6\ 1)$, the derivation using eval-call is

$$\dfrac{\dfrac{}{(\texttt{Lam}\ x\ (+\ x\ x)) \Downarrow (\texttt{Lam}\ x\ (+\ x\ x))}\ \text{eval-lam} \qquad \dfrac{\dfrac{}{6 \Downarrow 6} \qquad \dfrac{}{1 \Downarrow 1}}{(-\ 6\ 1) \Downarrow 5}\ \text{eval-sub} \qquad \dfrac{\dfrac{}{5 \Downarrow 5} \qquad \dfrac{}{5 \Downarrow 5}}{(+\ 5\ 5) \Downarrow 10}\ \text{eval-add}}{(\texttt{Call}\ (\texttt{Lam}\ x\ (+\ x\ x))\ (-\ 6\ 1)) \Downarrow 10}\ \text{eval-call}$$

We can estimate the *cost* of this evaluation by counting the number of horizontal lines in the derivation (here, 8). However, that's not a great estimate because some of the rules applied in the derivation have no real connection to machine operations in compiled code: there aren't really any instructions associated with eval-lam. So another, probably better, estimate of cost is to count the number of arithmetic operations. The derivation above does one subtraction (eval-sub) and one addition (eval-add), for a total of two operations.

If we instead use eval-call-by-name, we get this derivation:

$$\dfrac{\dfrac{}{(\texttt{Lam}\ x\ (+\ x\ x)) \Downarrow (\texttt{Lam}\ x\ (+\ x\ x))}\ \text{eval-lam} \qquad \dfrac{\dfrac{\dfrac{}{6 \Downarrow 6} \qquad \dfrac{}{1 \Downarrow 1}}{(-\ 6\ 1) \Downarrow 5}\ \text{eval-sub} \qquad \dfrac{\dfrac{}{6 \Downarrow 6} \qquad \dfrac{}{1 \Downarrow 1}}{(-\ 6\ 1) \Downarrow 5}\ \text{eval-sub}}{(+\ (-\ 6\ 1)\ (-\ 6\ 1)) \Downarrow 10}\ \text{eval-add}}{(\texttt{Call}\ (\texttt{Lam}\ x\ (+\ x\ x))\ (-\ 6\ 1)) \Downarrow 10}\ \text{eval-call-}\textbf{by-name}$$

This second derivation "costs more" by both our estimates: it has 10 horizontal lines (instead of 8) and does 3 arithmetic operations (two subtractions and one addition) instead of 2. If we replaced the argument `(- 6 1)` with a much larger expression, the difference between eval-call and eval-call-by-name would become more stark: eval-call-by-name would do all the work to evaluate the large argument twice.

The semantics of eval-call is known as *call-by-value*: eval-call substitutes the *value* v_2 of the argument e_2, rather than substituting the entire expression e_2.

However, eval-call-by-name wins when the called function doesn't use the argument:

$$\dfrac{\dfrac{}{\texttt{(Lam x 3)} \Downarrow \texttt{(Lam x 3)}}\,\text{eval-lam} \qquad \dfrac{\dfrac{}{6 \Downarrow 6} \qquad \dfrac{}{1 \Downarrow 1}}{\texttt{(- 6 1)} \Downarrow 5}\,\text{eval-sub} \qquad \dfrac{}{3 \Downarrow 3}}{\texttt{(Call (Lam x 3) (- 6 1))} \Downarrow 3}\,\text{eval-call}$$

$$\dfrac{\dfrac{}{\texttt{(Lam x 3)} \Downarrow \texttt{(Lam x 3)}}\,\text{eval-lam} \qquad \dfrac{}{3 \Downarrow 3}}{\texttt{(Call (Lam x 3) (- 6 1))} \Downarrow 3}\,\text{eval-call-}\textbf{by-name}$$

1.1 Lazy evaluation

A third evaluation strategy is *lazy evaluation* or *call-by-need*. (In contrast, call-by-value is often called *strict evaluation*.) In this style, the argument is evaluated *at most once*. If, as in our second example with `(Lam x 3)`, the function does not use its argument, lazy evaluation behaves like call-by-name. However, if the function uses its argument, lazy evaluation will evaluate it once and then remember the resulting value; subsequent uses of the argument will use the remembered value. If our measure is the number of arithmetic operations, laziness is the optimal strategy: for cases where call-by-name is better than call-by-value, laziness is as good as call-by-name; when call-by-value is better than call-by-name, laziness is as good as call-by-value.

While probably better than counting horizontal lines, counting arithmetic operations doesn't tell the whole story: asking whether an argument has been evaluated, and then remembering it, has higher constant factors in time (and space) than call-by-value.

1.2 Languages

Our discussion has focused on the design of the evaluation rule for function calls, in isolation. Actual lazy languages, such as Haskell, make laziness pervasive: no operation will be performed unless the result is needed (*demanded*). So the arithmetic operations shown in the derivations above would not be done *at all* unless "forced". This is not always obvious, because typing an expression into an implementation of Haskell *does* force evaluation: you, the user, are demanding to see the actual answer.

The question of which languages use call-by-value, call-by-name, or call-by-need is more complicated than it was in the 1980s. Many language designers have come to recognize that each evaluation strategy has advantages and disadvantages: the question is not whether a language is lazy, but whether it is lazy *by default*. (Algol-60, the first language to support call-by-name, also supported call-by-value.) Thus, if a Haskell programmer knows that the result of a function will always (or usually) be demanded, they can temporarily turn off laziness, reducing overhead. Similarly, an SML programmer can use the `lazy` keyword to temporarily turn on laziness. It is even possible for a compiler to generate multiple versions of a function that are lazy, strict, or a mix of the two (Dunfield 2015).

2 Lλ(small-step)

(Copied from lec10)

$\boxed{e \mapsto_{\mathsf{R}} e'}$ Expression e reduces to e'

$$\frac{}{\texttt{(+}\ n_1\ n_2\texttt{)} \mapsto_{\mathsf{R}} (n_1 + n_2)}\ \text{red-add} \qquad \frac{}{\texttt{(-}\ n_1\ n_2\texttt{)} \mapsto_{\mathsf{R}} (n_1 - n_2)}\ \text{red-sub}$$

$$\frac{}{\texttt{(=}\ n_1\ n_2\texttt{)} \mapsto_{\mathsf{R}} (n_1 = n_2)}\ \text{red-equals} \qquad \frac{}{\texttt{(<}\ n_1\ n_2\texttt{)} \mapsto_{\mathsf{R}} (n_1 < n_2)}\ \text{red-lessthan}$$

$\boxed{e \mapsto e'}$ expression e takes one step to e'

$$\frac{e \mapsto_{\mathsf{R}} e'}{\mathcal{C}[e] \mapsto \mathcal{C}[e']}\ \text{step-context}$$

Roughly, the idea is that $\mathcal{C}[]$ is an expression containing a *hole*, written $[]$. If we replace the hole with e, we get $\mathcal{C}[e]$, and if we replace it with e', we get $\mathcal{C}[e']$. Think of $\mathcal{C}$ as the *context* that surrounds the expression e. Since the premise $e \mapsto_{\mathsf{R}} e'$ says that e reduces to e' using one of the reduction rules (red-add, red-sub, ...), step-context says that if a subexpression e reduces to e', then $\mathcal{C}[e]$—which *contains* e—reduces to $\mathcal{C}[e']$.

$$\begin{array}{lrl}
\text{Contexts} & \mathcal{C} ::= & [] \\
 & & \mid \texttt{(+}\ \mathcal{C}\ e\texttt{)} \mid \texttt{(+}\ e\ \mathcal{C}\texttt{)} \\
 & & \mid \texttt{(-}\ \mathcal{C}\ e\texttt{)} \mid \texttt{(-}\ e\ \mathcal{C}\texttt{)} \\
 & & \mid \texttt{(=}\ \mathcal{C}\ e\texttt{)} \mid \texttt{(=}\ e\ \mathcal{C}\texttt{)} \\
 & & \mid \texttt{(<}\ \mathcal{C}\ e\texttt{)} \mid \texttt{(<}\ e\ \mathcal{C}\texttt{)} \\
 & & \mid \texttt{(Ite}\ \mathcal{C}\ e\ e\texttt{)} \\
 & & \mid \texttt{(Call}\ \mathcal{C}\ e\texttt{)} \\
 & & \mid \texttt{(Call}\ e\ \mathcal{C}\texttt{)}
\end{array}$$

Why not include $\texttt{(Ite}\ e\ \mathcal{C}\ e\texttt{)}$ and $\texttt{(Ite}\ e\ e\ \mathcal{C}\texttt{)}$? First, that would not correspond to our big-step rules for Ite, which don't evaluate the then- or else-parts until the condition has been evaluated. Second, there is no practical advantage to evaluating the then- or else-parts early: if the condition evaluates to True and we need to evaluate the then-part, we save nothing by evaluating it early; if the condition evaluates to False, we have wasted the work of evaluating the then-part.

I am exaggerating about there being *no* practical advantage. Since the late 1980s or so, microprocessors have had multiple execution units, and nowadays multiple cores—effectively, multiple processors. Speculative execution after a branch is, essentially, evaluating both the then-part and the else-part in parallel. Given recent security news, however, speculative execution may not be a great idea. And, even if speculative execution had no practical downsides, we want to be able to reason about programs *as if* speculative execution weren't happening. Thus, our semantics reflects the "naïve" view (which was reality before the 1980s) that we have a single processor to be used as efficiently as possible.

(Some PL semantics do reflect a non-naïve view, modelling multiple processes or threads. See, for example, work on weak memory models.)

2.1 Determinacy

The above grammar of contexts yields a semantics that is deterministic in one sense, but not in another. As an example, `(+ (- 4 3) (- 0 9))` can step to `(+ 1 (- 0 9))` *or* to `(+ (- 4 3) −9)`. However, each of those expressions can step only to `(+ 1 −9)` and then to −8. Stepping is, therefore, *confluent* (different sequences of steps will give the same value)—or *weakly deterministic*—but not *strongly* deterministic.

It is convenient, and more realistic, to ensure that our semantics is strongly deterministic, which can be achieved as follows.

1. Find the expression forms that have multiple productions in the grammar of contexts. (All such expression forms have at least two subexpressions, but not all expression forms with at least two subexpressions have two subexpressions: $(\texttt{Ite}\ e\ e_1\ e_2)$ has only one, because we don't want to reduce inside the branches before we know which branch will be taken.)

2. For each such expression form, in its "right-handed" production—the one in which the hole appears within the rightmost subexpression—change e to v.

This change gives us the grammar

$$
\begin{array}{lrcl}
\text{Contexts} & \mathcal{C} & ::= & [\,] \\
 & & | & (\texttt{+}\ \mathcal{C}\ e) \mid (\texttt{+}\ v\ \mathcal{C}) \\
 & & | & (\texttt{-}\ \mathcal{C}\ e) \mid (\texttt{-}\ v\ \mathcal{C}) \\
 & & | & (\texttt{=}\ \mathcal{C}\ e) \mid (\texttt{=}\ v\ \mathcal{C}) \\
 & & | & (\texttt{<}\ \mathcal{C}\ e) \mid (\texttt{<}\ v\ \mathcal{C}) \\
 & & | & (\texttt{Ite}\ \mathcal{C}\ e\ e) \\
 & & | & (\texttt{Call}\ \mathcal{C}\ e) \\
 & & | & (\texttt{Call}\ v\ \mathcal{C})
\end{array}
$$

Since no reductions can occur inside a value, this new grammar ensures that reduction is possible at no more than one position. For example, `(+ (- 4 3) (- 0 9))` can be viewed as

$$\mathcal{C}_1[(\texttt{-}\ 4\ 3)]$$

where $\mathcal{C}_1 = (\texttt{+}\ [\,]\ (\texttt{-}\ 0\ 9))$, but *not* as

$$\mathcal{C}_2[(\texttt{-}\ 0\ 9)]$$

where $\mathcal{C}_2 = (\texttt{+}\ (\texttt{-}\ 4\ 3)\ [\,])$, because our new grammar says that $\mathcal{C}_2$ isn't a context: `(- 4 3)` is not a value, and the new production is $(\texttt{+}\ v\ \mathcal{C})$.

2.2 Order of evaluation

(You may find the distinction between call-by-value, call-by-name, etc. called *order of evaluation*, but I prefer to use *evaluation strategy*.)

Now that our contexts enforce left-to-right evaluation, we can ask if that is necessarily the best choice. The OCaml language does not specify whether subexpressions (in arithmetic operators, and in function calls) should be evaluated left-to-right, or right-to-left. While OCaml has a single dominant implementation, that implementation includes *two* compilers: a compiler `ocamlc` that quickly generates bytecode (something like traditional Java compilers), and another compiler `ocamlopt` that generates fast assembly code. The OCaml developers determined that right-to-left evaluation of function calls sped up the bytecode interpreter, so depending on whether we generate bytecode or assembly, the OCaml program

```
( (print_string "hi\n"; (fun x -> x + 1)) (print_string "five\n"; 5));;
```

prints either

```
hi
five
```

or

```
five
hi
```

OCaml does provide a `let` expression, similar to Racket's `let`, for which order of evaluation is left-to-right (or downwards, depending on code formatting) in both compilers.

References

Jana Dunfield. Elaborating evaluation-order polymorphism. In *Int'l Conf. Functional Programming*, 2015. arXiv:`1504.07680 [cs.PL]`.