lec12: From addition to L\(\lambda\)

Joshua Dunfield

February 12, 2019

1 L\(\lambda\)(big-step)

In these notes, we extend our language from addition (and absolute value) with several useful features, and one unbelievably general feature.

The useful features are:

- subtraction, written \((- e_1 e_2)\);
- integer comparisons, \((= e_1 e_2)\) and \((< e_1 e_2)\);
- boolean constants True and False;
- if-then-else, \((\text{Ite } e \text{ then } e_\text{then } e_\text{else})\).

The if-then-else expression introduces nontrivial “control flow”: evaluating a large expression \(e\) no longer means that every subexpression will be evaluated.

The unbelievably general feature will be implemented by three expression forms:

- an anonymous function (procedure, subroutine) \((\text{Lam } x \ e)\);
- function call (procedure call, function application) \((\text{Call } e_1 e_2)\);
- variables (identifiers) \(x\). (The Racket implementation’s define-type uses \((\text{Id } x)\).

By building on these core constructs, which implement functions that take a single argument and return a single result, we can get multi-argument functions. We can also get let-binding, addition, comparisons, if-then-else, subtraction, and data structures (such as lists and trees). The necessary “encodings” to simulate these features using \(\text{Lam}\) and \(\text{Call}\) are rather awkward, but give some insight into why a language with this single feature is very powerful—equivalent to Turing machines.

The language with only \(\text{Lam}\), \(\text{Call}\) and variables \(x\) is the lambda calculus (\(\lambda\)-calculus). I could have started with that language and gradually added basic operations such as addition, but I thought it would be more clear to begin with the basic operations.

My notation for \(\text{Lam}\), \(\text{Call}\) and \(\text{Id}\) is not standard; each column of Table 1 collects synonyms and equivalent notation, and a sampling of notations in a variety of programming languages.
§ 1 \( \lambda \text{(big-step)} \)

<table>
<thead>
<tr>
<th>anonymous function abstraction</th>
<th>function call application</th>
<th>identifier variable</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \lambda ) function application</td>
<td>( \lambda )-application</td>
<td>( \lambda )-variable</td>
</tr>
<tr>
<td>( \lambda )-abstraction</td>
<td>( \lambda )-application</td>
<td>( \lambda )-bound variable</td>
</tr>
<tr>
<td>( \lambda )-bound variable</td>
<td>( \lambda )-application</td>
<td>( \lambda )-variable</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Alonzo Church</th>
<th>( \lambda x. e )</th>
<th>( e_1 ) ( e_2 )</th>
<th>( x )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Racket</td>
<td>( \text{lambda} (x) e )</td>
<td>( e_1 ) ( e_2 )</td>
<td>( x )</td>
</tr>
<tr>
<td>Haskell</td>
<td>( \text{fun} x \to e )</td>
<td>( e_1 ) ( e_2 )</td>
<td>( x )</td>
</tr>
<tr>
<td>SML</td>
<td>( \text{fn} x \to e )</td>
<td>( e_1 ) ( e_2 )</td>
<td>( x )</td>
</tr>
<tr>
<td>OCaml</td>
<td>( \text{fun} x \to e )</td>
<td>( e_1 ) ( e_2 )</td>
<td>( x )</td>
</tr>
<tr>
<td>Python</td>
<td>( \text{lambda} x: e )</td>
<td>( e_1(\ e_2) )</td>
<td>( x )</td>
</tr>
<tr>
<td>Java (added in 2014)</td>
<td>( x \to e )</td>
<td>( e_1(e_2) )</td>
<td>( x )</td>
</tr>
<tr>
<td>JavaScript</td>
<td>( x \Rightarrow e )</td>
<td>( e_1(e_2) )</td>
<td>( x )</td>
</tr>
<tr>
<td>C++ (added in 2011)</td>
<td>( [] \ (\text{type} \ x \to \text{type} \ { \ e } )</td>
<td>( e_1(e_2) )</td>
<td>( x )</td>
</tr>
</tbody>
</table>

Notes:

- In most non-Lisp-like languages, parentheses can always be added, so \( e_1 \ e_2 \) can also be written \( e_1(\ e_2) \).

- JavaScript has had multiple forms of \( \lambda \) that differ subtly (particular in their treatment of \text{this}); I have only listed the syntax added in ES6.

- The C++11 lambda has unusual scoping rules: “captured” variables must be listed between the brackets \[ \]. (Early versions of Python had a similar, but even more awkward, requirement.) This usage of “capture” is different from that in “capture-avoiding substitution” (when we get around to that).

<table>
<thead>
<tr>
<th>Table 1 Lambda notations of the world</th>
</tr>
</thead>
</table>

1.1 Non-lambda features

\[
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{( - e_1 e_2 ) \Downarrow n_1 - n_2} \quad \text{eval-sub} \\
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{( = e_1 e_2 ) \Downarrow (n_1 = n_2)} \quad \text{eval-equals} \\
\frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow n_2}{(< e_1 e_2 ) \Downarrow n_1 < n_2} \quad \text{eval-less-than} \\
\frac{\text{True} \Downarrow \text{True}}{\text{eval-true}} \\
\frac{\text{False} \Downarrow \text{False}}{\text{eval-false}} \\
\frac{e \Downarrow \text{True}}{(\text{Ite} e \text{ then } e_{\text{else}}) \Downarrow \nu} \quad \text{eval-ite-then} \\
\frac{e \Downarrow \text{False}}{(\text{Ite} e \text{ then } e_{\text{else}}) \Downarrow \nu} \quad \text{eval-ite-else} \\
\]

2019/2/12
1.2 Lambda

\[
\begin{align*}
(Lam \times e) \Downarrow (Lam \times e) & \quad \text{eval-lam} \\
& \\
\begin{array}{ll}
(\text{Lam } x \ e)_1 & \Downarrow (\text{Lam } x \ e) \\
(\text{Lam } x \ e)_2 & \Downarrow v_2 \\
[v_2/x]e_{\text{body}} & \Downarrow v \\
\text{(Call } e_1 e_2) & \Downarrow v
\end{array} & \quad \text{eval-call}
\end{align*}
\]

**Exercise 1.** With some of our rules, we didn’t have much choice about how to design them: I’m pretty sure there is no version of eval-sub that doesn’t do the same thing that our eval-sub rule does. There are different ways of writing eval-sub; for example, we could add a premise \( n = n_1 - n_2 \), and change the conclusion to \( \ldots \Downarrow n \). But that rule would derive exactly the same set of judgments as eval-sub.

With eval-call, we have more choices. Can you find another version of the rule that also seems to reasonably implement a function call, but is substantially different (not just a different way of writing my eval-call)?
Following the pattern of early lecture notes, we can systematically design small-step rules for \(-\), = and <, such as

\[
\begin{align*}
(- n_1 n_2) & \mapsto n_1 - n_2 & \text{step-sub} \\
(- e_1 e_2) & \mapsto (- e_1' e_2) & \text{step-sub-1} \\
(- e_1 e_2) & \mapsto (- e_1 e_2') & \text{step-sub-2}
\end{align*}
\]

Since this requires three rules per operation, we would need a total of 12 rules just for the four operations \(+\), \(-\), = and <. We would need two for Abs, for a total of 14. Such a large number of rules would be tedious to write, but the real pain comes when we try to prove anything about such derivations: a large number of rules, in general, leads to a large number of proof cases.

Many of these rules are very similar to each other: except for rules like step-sub and step-add that perform an actual arithmetic operation, these rules all “delegate” stepping to a subexpression. For example, step-sub-2 delegates the job of stepping to the subexpression \(e_2\). That delegation works in exactly the same way for \(+\), = and <.

Fortunately, we can abstract over this delegation by a technique called contexts. First, we distinguish between “real” computations (step-sub) and delegation (step-sub-1, step-sub-2, etc.). The real computations will be called reductions, and will get their own judgment, \(e \mapsto_R e'\). The “R” stands for “reduce”.

(While it is clearly reasonable to say that \((+ 1 2)\) reduces to 3, since 3 is a smaller expression than \((+ 1 2)\), later we’ll see reductions that can create larger expressions.)

We can define reductions for five operations (the four binary operators, along with Abs):

\[
\begin{align*}
\ldelim{[}{0em}e \mapsto_R e'\rddelim{]}{0em} & \text{Expression } e \text{ reduces to } e' \\
(+ n_1 n_2) & \mapsto_R (n_1 + n_2) & \text{red-add} \\
(- n_1 n_2) & \mapsto_R (n_1 - n_2) & \text{red-sub} \\
(= n_1 n_2) & \mapsto_R (n_1 = n_2) & \text{red-equals} \\
(< n_1 n_2) & \mapsto_R (n_1 < n_2) & \text{red-less-than} \\
(Abs n) & \mapsto_R |n| & \text{red-abs}
\end{align*}
\]

This leaves the problem of designing stepping rules equivalent to step-...-1, step-...-2. With the help of a grammar, we can do this using only one rule:

\[
\begin{align*}
e & \mapsto_R e' \\
C[e] & \mapsto C[e'] & \text{step-context}
\end{align*}
\]

Roughly, the idea is that \(C[\ ]\) is an expression containing a hole, written \([\ ]\). If we replace the hole with \(e\), we get \(C[e]\), and if we replace it with \(e'\), we get \(C[e']\). Think of \(C\) as the context that surrounds the expression \(e\). Since the premise \(e \mapsto_R e'\) 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 \(C[e]\)—which contains \(e\)—reduces to \(C[e']\).

Before giving the grammar of \(C\), let's look at some examples.
Example 1. **Old way:** Using step-sub-2 in the conclusion and step-sub-1 to derive the premise of step-sub-2, we can step \((- (- 9 \ 1) (- (- 100 \ 15) \ 6))\) to \((- (- 9 \ 1) (- 85 \ 6))\).

\[
\begin{align*}
(- 100 \ 15) \mapsto 85 & \quad \text{step-sub} \\
(- (- 100 \ 15) \ 6) \mapsto (- 85 \ 6) & \quad \text{step-sub-1} \\
(- (- 9 \ 1) (- (- 100 \ 15) \ 6)) \mapsto (- (- 9 \ 1) (- 85 \ 6)) & \quad \text{step-sub-2}
\end{align*}
\]

I have highlighted the subexpression \((- 100 \ 15)\) where the “real” computation happens. Notice that, as the derivation moves from the conclusion towards \((- 100 \ 15)\), the context surrounding it becomes smaller; when the context disappears, leaving only \((- 100 \ 15)\), we use step-sub.

**New way:** With an appropriate definition of \(C\), we should be able to derive the same \(\mapsto\) judgment using step-context and red-sub:

\[
\begin{align*}
(- 100 \ 15) \mapsto_R 85 & \quad \text{red-sub} \\
(- (- 9 \ 1) (- (- 100 \ 15) \ 6)) \mapsto (- (- 9 \ 1) (- 85 \ 6)) & \quad \text{step-context}
\end{align*}
\]

This derivation requires that one possible \(C\), according to our yet-to-be-written grammar, is \((- (- 9 \ 1) (- [] \ 6))\).

Example 2. **Old way:** We have previously discussed how our stepping rules are nondeterministic, in that they don’t always step the same subexpressions in the same order. For example, we could step the first \(-\) subexpression in \((- (- 9 \ 1) (- (- 100 \ 15) \ 6))\):

\[
\begin{align*}
(- 9 \ 1) \mapsto 8 & \quad \text{step-sub} \\
(- (- 9 \ 1) (- (- 100 \ 15) \ 6)) \mapsto (- 8 (- (- 100 \ 15) \ 6)) & \quad \text{step-sub-1}
\end{align*}
\]

**New way:** With an appropriate definition of \(C\), we should be able to derive the same \(\mapsto\) judgment using step-context and red-sub:

\[
\begin{align*}
(- 100 \ 15) \mapsto_R 85 & \quad \text{red-sub} \\
(- (- 9 \ 1) (- (- 100 \ 15) \ 6)) \mapsto (- 8 (- (- 100 \ 15) \ 6)) & \quad \text{step-context}
\end{align*}
\]

This derivation requires that one possible \(C\), according to our yet-to-be-written grammar, is \((- [] (- (- 100 \ 15) \ 6))\).

In these examples, much of the surrounding context is irrelevant: in the last example, if we changed 100 to 1 we could still reduce \((- 9 \ 1)\) in exactly the same way. That is, \((- [] (- (- 1 \ 15) \ 6))\) should also be a possible \(C\). In fact, if we change the “other” subexpression \((- (- 100 \ 15) \ 6)\) to *anything*, we should still have a possible \(C\):

\[
\begin{align*}
&(- [] (- (- 1 \ 15) \ 6)) \\
&(- [] (- \ 0 \ 6)) \\
&(- [] -11) \\
&(- [] (\text{Abs -11})) \\
&(- [] (\text{Abs True}))
\end{align*}
\]
The last expression is not even sensible, because \((\text{Abs True})\) has (I hope) no meaning, but even that should not impede us from reducing the expression to its left.

That is, for any expression \(e_2\), the context \((- \ [ \ ] \ e_2)\) should be in the grammar of \(C\). The same holds for \((- \ e_1 \ [ \ ]))\).

For our first example, we need to be able to nest contexts, so we won’t put literally \((- \ [ \ ] \ e_2)\) and \((- \ e_1 \ [ \ ]))\) in our grammar; instead, we will put \((- \ C \ e_2)\) and \((- \ e_1 \ C)\).

To maintain our ability to step without a surrounding context, e.g. \((-1 \ 3) \mapsto -2\), we include a production \([\ ]\).

\[
\text{Contexts} \quad C ::= \[\] \mid (+ \ C \ e) \mid (+ \ e \ C) \mid (- \ C \ e) \mid (- \ e \ C) \mid (= \ C \ e) \mid (= \ e \ C) \mid (< \ C \ e) \mid (< \ e \ C)
\]

**Exercise 2.** Extend \(C\) with productions for \(\text{Ite}\).