lec10: From addition to Lλ

Jana Dunfield

February 16, 2022

(A previous version of the lecture notes incorrectly mentioned an absolute value expression, which is not in the rest of the lecture notes. Absolute value can be implemented using the other new language features.)

1 Lλ(big-step)

In these notes, we extend our language that has only addition by adding 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, `(Ite` e e_{then} e_{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) `(Lam` x e`)`;
- function call (procedure call, function application) `(Call` e_1 e_2`)`;
- variables (identifiers) 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 `Lam` and `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 `Lam`, `Call` and variables x is the *lambda calculus* (λ-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 `Lam`, `Call` and `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.

	anonymous function	function call	identifier
	abstraction	application	variable
	λ	function application	λ-variable
	λ-abstraction	λ-application	λ-bound variable
	`(Lam` x e`)`	`(Call` e_1 e_2`)`	x
Alonzo Church	$\lambda x.\, e$	$e_1\ e_2$	x
Racket	`(lambda (`x`)` e`)`	$(e_1\ e_2)$	x
Haskell	`\` x `->` e	$e_1\ e_2$	x
SML	`fn` x `=>` e	$e_1\ e_2$	x
OCaml	`fun` x `->` e	$e_1\ e_2$	x
Python	`lambda` x`:` e	$e_1(e_2)$	x
Java (added in 2014)	x `->` e	$e_1(e_2)$	x
JavaScript	x `=>` e	$e_1(e_2)$	x
C++ (added in 2011)	`[](`*type* x`) ->` *type* `{` e `}`	$e_1(e_2)$	x

Notes:

- In many languages (including Haskell, SML and OCaml, but not including Racket), extra parentheses may be added, so $e_1\ e_2$ can also be written $e_1(e_2)$.
- JavaScript has had multiple forms of λ that differ subtly (particular in their treatment of `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 1 **Lambda notations of the world**

1.1 Non-lambda features

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{(\texttt{-}\ e_1\ e_2) \Downarrow n_1 - n_2}\ \text{eval-sub}$$

$$\frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{(\texttt{=}\ e_1\ e_2) \Downarrow (n_1 = n_2)}\ \text{eval-equals} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow n_2}{(\texttt{<}\ e_1\ e_2) \Downarrow n_1 < n_2}\ \text{eval-lessthan}$$

$$\frac{}{\textsf{True} \Downarrow \textsf{True}}\ \text{eval-true} \qquad \frac{}{\textsf{False} \Downarrow \textsf{False}}\ \text{eval-false}$$

$$\frac{e \Downarrow \textsf{True} \qquad e_{\text{then}} \Downarrow v}{(\texttt{Ite}\ e\ e_{\text{then}}\ e_{\text{else}}) \Downarrow v}\ \text{eval-ite-then} \qquad \frac{e \Downarrow \textsf{False} \qquad e_{\text{else}} \Downarrow v}{(\texttt{Ite}\ e\ e_{\text{then}}\ e_{\text{else}}) \Downarrow v}\ \text{eval-ite-else}$$

1.2 Lambda

$$\frac{}{(\texttt{Lam}\ x\ e) \Downarrow (\texttt{Lam}\ x\ e)}\ \text{eval-lam}$$

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

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 $\cdots \Downarrow n$. 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)?

2 Lλ(small-step)

Following the pattern of early lecture notes, we can systematically design small-step rules for -, = and <, such as

$$\frac{}{(\texttt{-}\ n_1\ n_2) \mapsto n_1 - n_2}\ \text{step-sub}$$

$$\frac{e_1 \mapsto e_1'}{(\texttt{-}\ e_1\ e_2) \mapsto (\texttt{-}\ e_1'\ e_2)}\ \text{step-sub-1} \qquad \frac{e_2 \mapsto e_2'}{(\texttt{-}\ e_1\ e_2) \mapsto (\texttt{-}\ e_1\ e_2')}\ \text{step-sub-2}$$

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 absolute value 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):

$\boxed{e \mapsto_R e'}$ Expression e reduces to e'

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

$$\frac{}{(\texttt{=}\ n_1\ n_2) \mapsto_R (n_1 = n_2)}\ \text{red-equals} \quad \frac{}{(\texttt{<}\ n_1\ n_2) \mapsto_R (n_1 < n_2)}\ \text{red-lessthan} \quad \frac{}{(\texttt{Abs}\ n) \mapsto_R |n|}\ \text{red-abs}$$

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*:

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

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 $e \mapsto_R e'$ 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 $\mathcal{C}[e']$.

Before giving the grammar of $\mathcal{C}$, 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))`.

$$\dfrac{\dfrac{\dfrac{}{\texttt{(- 100 15)} \mapsto 85}\;\text{step-sub}}{\texttt{(- (- 100 15) 6)} \mapsto \texttt{(- 85 6)}}\;\text{step-sub-1}}{\texttt{(- (- 9 1) (- (- 100 15) 6))} \mapsto \texttt{(- (- 9 1) (- 85 6))}}\;\text{step-sub-2}$$

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 $\mathcal{C}$, we should be able to derive the same $\mapsto$ judgment using step-context and red-sub:

$$\dfrac{\dfrac{}{\texttt{(- 100 15)} \mapsto_R 85}\;\text{red-sub}}{\texttt{(- (- 9 1) (- (- 100 15) 6))} \mapsto \texttt{(- (- 9 1) (- 85 6))}}\;\text{step-context}$$

This derivation requires that one possible $\mathcal{C}$, 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))`:

$$\dfrac{\dfrac{}{\texttt{(- 9 1)} \mapsto 8}\;\text{step-sub}}{\texttt{(- (- 9 1) (- (- 100 15) 6))} \mapsto \texttt{(- 8 (- (- 100 15) 6))}}\;\text{step-sub-1}$$

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

$$\dfrac{\dfrac{}{\texttt{(- 9 1)} \mapsto_R 8}\;\text{red-sub}}{\texttt{(- (- 9 1) (- (- 100 15) 6))} \mapsto \texttt{(- 8 (- (- 100 15) 6))}}\;\text{step-context}$$

This derivation requires that one possible $\mathcal{C}$, 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 $\mathcal{C}$. In fact, if we change the "other" subexpression `(- (- 100 15) 6)` to *anything*, we should still have a possible $\mathcal{C}$:

```
(- [] (- (- 1 15) 6))
(- [] (- 0 6))
(- [] −11)
(- [] (Abs −11))
(- [] (Abs True))
```

The last expression is not even sensible, because (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 $\mathcal{C}$. 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 (- $\mathcal{C}$ e_2) and (- e_1 $\mathcal{C}$).

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

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

Exercise 2. Extend $\mathcal{C}$ with productions for Ite.