LAM — Semantic Variations

Neil Ghani and Conor McBride

October 9, 2018

Our grammar of λ-terms is given by

```
<term>
  ::= <var>
    | <term> <term>
    | \ <var> -> <term>
    | <number>
    | (<term> + <term>)
```

If we were real purists, we would omit numbers and addition: we include them to represent 'ordinary' computation, where the job of the λ-terms is to route data to whichever computation is appropriate.

1 Small Step Reduction

We write f, s and t for terms, and m and n for numerical constants. Our judgement form is

$$\boxed{s \leadsto t}$$

with s as an input and t as an output.

The key axioms which do the work are β-reduction and numerical addition.

$$\frac{}{(\backslash\ x\ \texttt{->}\ t[x])\ s \leadsto t[s]} \qquad \frac{}{(m\ \texttt{+}\ n) \leadsto m + n}$$

They are not, however, enough: before you make a reduction step, you have to find a redex! We add *contextual closure* rules:

$$\frac{f \leadsto f'}{f\ s \leadsto f'\ s} \qquad \frac{s \leadsto s'}{f\ s \leadsto f\ s'}$$

$$\frac{t[x] \leadsto t'[x]}{\backslash\ x\ \texttt{->}\ t[x] \leadsto \backslash\ x\ \texttt{->}\ t'[x]}$$

$$\frac{s \leadsto s'}{(s\ \texttt{+}\ t) \leadsto (s'\ \texttt{+}\ t)} \qquad \frac{t \leadsto t'}{(s\ \texttt{+}\ t) \leadsto (s\ \texttt{+}\ t')}$$

The contextual closure rules are very far from arbitrary: they are generated from the grammar, with one rule for every way to put a subterm inside a term. Their impact is to allow β-reduction and numerical addition *anywhere* inside a term. These rules are highly nondeterministic: when there are multiple redexes, they do not tell you which one to choose.

2 Big Step Reduction

Here is one way to give a big step semantics for LAM. Before we can write the rules, we need to say 'what success looks like': to what *values* are we trying to compute the terms?

In our language of functions and numbers, let us say that a function is a value if it is ready to be applied, and a number is a value if it is ready to be added.

```
<value>
  ::= \ <var> -> <term>
    | <number>
```

Note that for functions we make no demands on the body of the abstraction. Let us write u and v for values. Our judgement form is

$$\boxed{t \Downarrow v}$$

with t as input and v as output. We have only four rules. Two tell us how to stop

$$\frac{}{\backslash\ x\ \texttt{->}\ t[x] \Downarrow \backslash\ x\ \texttt{->}\ t[x]} \qquad \frac{}{n \Downarrow n}$$

and two tell us how to go.

$$\frac{f \Downarrow \backslash\ x\ \texttt{->}\ t[x] \quad t[s] \Downarrow v}{f\ s \Downarrow v} \qquad \frac{s \Downarrow m \quad t \Downarrow n}{\texttt{(}s\ \texttt{+}\ t\texttt{)} \Downarrow m + n}$$

There are several things to notice about this semantics:

- It does not give *every* term a value. (Think about which terms do not get a value, and why. There are several kinds of 'bad'.)
- It has a deterministic evaluation strategy.
- It never computes inside the body of an abstraction: rather, it leaves the body untouched until the function's argument turns up.
- It is still reliant on substitution.
- A function's arguments are not evaluated until after substitution: this is a win if the bound variable is unused, but unfortunate if it is used a lot.

3 Closures and Environments

There is no particularly good reason why values should be a subset of terms. They could be something apart. Here is a way to represent function values more efficiently, getting rid of substitution.

```
<val>
  ::= [<env>] <var> -> <term>
    | <number>

<env>
  ::=
    | <env>, <var> = <val>
```

Our function values are now given as structures called *closures* which pack up an *environment* along with an abstraction. An environment is a mapping of variables to values. A closure captures the situation where we are not quite ready to evaluate the body of an abstraction: we know the values of all the *free* variables, but not the value of the bound variable.

We write u and v for 'vals' and γ for environments. Our judgement form now takes an environment, to explain what the free variables mean.

$$\boxed{\gamma | t \Downarrow v}$$

We expect every free variable to have an entry in the environment. We make the further assumption that all the free variables have different names: this is achievable by α-conversion (but there is a better way).

We have one rule per construct.

$$\frac{}{\gamma|\backslash\ x \texttt{ -> } t \Downarrow [\gamma]x \texttt{ -> } t} \qquad \frac{}{\gamma|n \Downarrow n} \qquad \frac{}{\gamma, x = v, \gamma'|x \Downarrow v}$$

$$\frac{\gamma|f \Downarrow [\gamma']x \texttt{ -> } t \quad \gamma|s \Downarrow u \quad \gamma', x = u|t \Downarrow v}{\gamma|f\ s \Downarrow v} \quad \frac{\gamma|s \Downarrow m \quad \gamma|t \Downarrow n}{\gamma|\texttt{(}s \texttt{ + } t\texttt{)} \Downarrow m + n}$$

In a real implementation, we get rid of variable *names* entirely. We replace names variable usage sites by numbers called *de Bruijn indices*, locally distinguished from numerical constants by a # marker, which tell us how many λs to jump over before we find the one which bound the variable. That is

$$\backslash\ f \texttt{ -> } \backslash\ x \texttt{ -> } f\ (f\ x)$$

becomes

$$\backslash\ \ \texttt{ -> } \backslash\ \ \texttt{ -> } \#1\ (\#1\ \#0)$$

A de Bruijn index is just the thing to tell us *where* in γ, now an *array* of values (with 0 the rightmost index), is the value we want.

4 Thunks