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

$$\boxed{s \rightsquigarrow t}$$

with s as an input and t as an output.

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

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

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

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

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

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

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.

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

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.

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

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

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.

Note that in this variation, an application's argument is evaluated agressively, so that its value is ready to go in the environment. That creates an extra risk of failure: an argument which contains some sort of error will cause a problem, even if that argument is never used by the function.

4 Thunks

Next comes a variation which is less aggressive about evaluating arguments. The idea is that instead of evaluating things just before we put them into the environment, we evaluate things just after we take them out.

Our environments will thus contain terms, but that is not enough! In order to evaluate a term properly, we need to remember *its* environment.

```
<thunkEnv>
  ::=
    | <thunkEnv>, <var> = (<thunkEnv>|<term>)

<thunkVal>
  ::= [<thunkEnv>] <var> -> <term>
    | <number>
```

A pair of an environment and an unvaluated term in that environment is called a *thunk*. A thunk stores all the information needed to start the evaluation process whenever the value is demanded. We call that *forcing* the thunk.

Let us write θ for thunk environments. Our values change only in that closures store thunk environments instead of value environments: let us still use u and v for them.

The judgement form is thus

$$\theta|t\ \Downarrow\ v$$

We still have one rule per construct, but the work happens in different places.

$$\frac{}{\theta|\backslash\ x\ \texttt{->}\ t\ \Downarrow\ [\theta]x\ \texttt{->}\ t} \qquad \frac{}{\theta|n\ \Downarrow\ n} \qquad \frac{\theta|s\ \Downarrow\ u}{\theta_0, x = (\theta|s), \theta_1|x\ \Downarrow\ v}$$

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

See how work has moved from the application rule to the variable rule? Instead of evaluating the argument, we just thunk it ('thunk' is a verb as well as a noun), carefully saving the environment appropriate to it. It is only when the value of a variable is demanded that we look up the thunk and force it, using the saved environment.

5 Laziness

The trouble with our previous variation is that it repeats the same evaluation process every time the same thunk is forced. That is, if a variable is used more than once, we do the same work more than once. The potential for woeful inefficiency results. There's a trick to get out of that: the first time a thunk is forced, *overwrite* it with the resulting value. That is actually how Haskell works, and it is ironic that such a 'pure' language with no assignment works deep down by sneakily mutating a store.

A *thunk store*, (σ, n) is an array σ of n entries which are *either* thunks or values, so that $\sigma.i$ is either a thunk or a value if $i < n$. We write $\sigma(i := v)$ to represent the array updated by overwriting the thunk at i with its value. The number n thus tells us how to allocate a new thunk in the store. We can now represent thunk environments as mappings $\ldots, x = i, \ldots$ from variables to indices in the thunk store.

Our judgement form is now

$$\boxed{(\sigma, n), \theta | t \Downarrow v, (\sigma', n')}$$

with the evolving thunk store as an extra input and an extra output.

We now have two rules for variables.

$$\frac{\sigma.i = (\theta, s) \quad (\sigma, n), \theta s \Downarrow v, (\sigma', n')}{(\sigma, n), \theta_0, x = i, \theta_1 | x \Downarrow v, (\sigma'(i = v), n')} \qquad \frac{\sigma.i = v}{(\sigma, n), \theta_0, x = i, \theta_1 | x \Downarrow v, (\sigma, n)}$$

The application rule has to allocate a new thunk.

$$\frac{(\sigma_0, n_0), \theta | f \Downarrow [\theta']x \texttt{->} t(\sigma_1, n_1) \quad (\sigma_1(n_1 := (\theta|s)), n_1 + 1), \theta', x = n' | t \Downarrow v, (\sigma_2, n_2)}{(\sigma_0, n_0)\theta | f\ s \Downarrow v, (\sigma_2, n_2)}$$

The other rules must take care to thread the mutation of the thunk store, but are otherwise the same.

$$\frac{}{(\sigma, n), \theta m \Downarrow m, (\sigma, n)} \qquad \frac{}{(\sigma, n), \theta | \backslash\ x \texttt{->} t \Downarrow [\theta]x \texttt{->} t, (\sigma, n)}$$

$$\frac{(\sigma_0, n_0), \theta | s \Downarrow m, (\sigma_1, n_1) \quad (\sigma_1, n_1), \theta | t \Downarrow n, (\sigma_2, n_2)}{(\sigma_0, n_0), \theta | \texttt{(}s\ \texttt{+}\ t\texttt{)} \Downarrow m + n, (\sigma_2, n_2)}$$