Lecture 4

Untyped λ-calculus, call-by-name

This is in Chapter 5 of Pierce's book and in the Agda book of Kokke and Wadler.

We recall the syntax

$$e \ ::= \ x \mid e\ e \mid \lambda x\ e$$

We define the set of free variables of e as follows.

$$FV(x) = \{x\} \qquad FV(e_0\ e_1) = FV(e_0) \cup FV(e_1) \qquad FV(\lambda\ x\ e) = FV(e) - \{x\}$$

An expression e is *closed* if we have $FV(e) = \emptyset$.
We define subsitution $e(t/x)$ for t *closed*. It is by case on e

- if $e = x$ then $e(t/x) = t$
- if $e = y \neq x$ then $e(t/x) = y$
- if $e = e_0\ e_1$ then $e(t/x) = e_0(t/x)\ e_1(t/x)$
- if $e = \lambda x\ e'$ then $e(t/x) = e$
- if $e = \lambda y\ e'$ with $y \neq x$ then $e(t/x) = \lambda y\ e'(t/x)$

We then define a *value* to be a closed expression of the form $\lambda x\ e$.

$$v \ ::= \ \lambda x\ e$$

We define the *call-by-name* evaluation relation $e \to e'$ for e and e' *closed* expressions

$$\frac{e \to e'}{e\ e_1 \to e'\ e_1} \qquad \frac{}{(\lambda x\ e)\ t \to e(t/x)}$$

Note that if $\delta = \lambda x\ x\ x$ then δ is a value and $\delta\ \delta \to \delta\ \delta$, so we have $\neg\exists e'\ NF(\delta\ \delta, e')$

This is closer to the evaluation in Haskell but there is a difference. In call-by-name, we may evaluate several time the same expression, as in

$$e = (\lambda y \lambda x\ y\ (y\ x))\ t\ I$$

where $t \to^* I$ and $I = \lambda x\ x$. The expression e will reduce to I but t will be evaluated twice.
The evaluation in Haskell is *call-by-need* which is more complex to describe.

The description does not work for *non closed* terms. For instance if $T = \lambda x \lambda y\ x$ we expect $T\ e_0\ e_1 \to^* e_0$. But if we take $T\ y\ e_1$ we have $T\ y \to (\lambda y\ x)(y/x) = \lambda y\ y$ and then $T\ y\ e_1 \to (\lambda y\ y)\ e_1 \to e_1$. What happens here is a *capture of variables*. This problem appears in the first implementation of LISP (by Steve Russell who is also known as the first implementor of video game, *Spacewar!*).

de Bruijn representation

We define the *terms* (in de Bruijn notation) as

$$t \ ::= \ n \mid \lambda t \mid t\ t$$

namely deBruijn index, or abstraction, or application.

The expressions $\lambda x\ x$ and $\lambda y\ y$ should be considered to be the same (the names of *bound* variables should not matter.) There is an elegant alternative representation of λ-terms where bound variables are represented by the "distance" to their introducing abstractions. This was used previously in compiling the language Algol.

E.g. $\lambda x \lambda y\ y\ (y\ x)$ is written $\lambda\lambda 0\ (0\ 1)$ while $\lambda y \lambda x\ y\ (y\ x)$ is written $\lambda\lambda 1\ (1\ 0)$. The algorithm is the following: the function dB takes a list of names and an expression and builds an expression with de Bruijn index.

- $dB\ (x : xs)\ x = 0$
- $dB\ (y : xs)\ x = 1 + dB\ xs\ x$ if $y \neq x$
- $dB\ xs\ (e_0\ e_1) = (dB\ xs\ e_0)\ (dB\ xs\ e_1)$
- $dB\ xs\ (\lambda x\ e) = \lambda\ (dB\ (x : xs)\ e)$

Krivine Abstract Machine

This provides an elegant way to "compile" evaluation in call-by-name. Note that we avoid to have to define *substitution* in this way. The use of *closure* goes back Peter Landin ("The Mechanical Evaluation of Expressions", 1964).

A *closure* u is a pair $t\rho$ of a term and an environment, where an *environment* ρ is a list of closures.

Krivine Abstract Machine has for states $t \mid \rho \mid S$ where $t\rho$ is a closure and S is a stack of values. The small step semantics is

$$\frac{}{0 \mid (t\rho, \nu) \mid S \rightarrow t \mid \rho \mid S} \qquad \frac{}{n+1 \mid (u, \nu) \mid S \rightarrow n \mid \nu \mid S}$$

$$\frac{}{\lambda t \mid \rho \mid u : S \rightarrow t \mid (u, \rho) \mid S}$$

$$\frac{}{t_0\ t_1 \mid \rho \mid S \rightarrow t_0 \mid \rho \mid (t_1\rho) : S}$$

So abstraction is "pop" while application is "push".

We can then evaluate $(\lambda\lambda\ 1\ (1\ 0))\ I\ I$ where $I = \lambda\ 0$ or $\delta\ \delta$ where $\delta = \lambda\ 0\ 0$.

Krivine Abstract Machine, other presentation

We define

$$e, t \ ::= \ n \mid e\ e \mid \lambda e \qquad n \ ::= \ 0 \mid n+1$$

$$c \ ::= \ (\lambda e, \rho) \mid c\ c \qquad \rho \ ::= \ () \mid \rho, c$$

We define substitution

$$0(\rho, c) = c \qquad (n+1)(\rho, c) = n\rho \qquad (e_0\ e_1)\rho = e_0\rho\ (e_1\rho) \qquad (\lambda e)\rho = (\lambda e, \rho)$$

and we can present the evaluation rule (call-by-name) as rules for deriving $c \to c'$

$$\frac{c \to c'}{c\ c_1 \to c'\ c_1} \qquad \frac{}{(\lambda t, \rho)\ c \to t(\rho, c)}$$

These rules can be "summarized" by the rule

$$\frac{}{(\lambda t, \rho)\ c\ c_1\ \ldots\ c_n \to t(\rho, c)\ c_1\ \ldots\ c_n}$$

For instance, if $\delta = \lambda\ 0\ 0$ then $\delta()\ \delta() \to \delta()\ \delta()$.