Scribe: Helgi Ingolfsson and Yoni Ben-Simhoni Lecturer: Andrew Myers

1 Let

So far we've been missing the ability to declare local variables. The let expression (as in ML), written let $x = e_1$ in e_2 end, binds the variable x to the result of e_1 and then evaluates e_2. That is, the result of evaluating let $x = v_1$ in e_2 end should be the result of evaluating $e_2\{v_1/x\}$. This familiar-looking expression suggests that we can encode a let with the lambda-calculus term that reduces to the same expression: that is, the term $(\lambda x.\, e_2)e_1$.

Using call-by-value we evaluate this term in the following manner. Do the reduction of the expression e_1 to the value v_1, then substitute x for v_1 in e_2. This is exactly what we'd expect from the let expression.

2 Recursion

Consider the recursive form of the factorial function

$$\texttt{fun}\ fact(x) = if\ (x = 0)\ then\ 1\ else\ x * fact(x - 1)$$

Using λ-calculus we might try to write it as

$$FACT \triangleq \lambda x.\, IF\ (ZERO?\ x)\ 1\ (*\ (FACT\ (-\ x\ 1))\ x)$$

This is not a definition but an equation. Recall that *FACT* isn't really part of the λ-calculus, it's just an abbreviation. An abbreviation can't contain itself! Trying that here would result in an infinite string, always substituting *FACT*, but always having one left. In order to remove the recursive definition we will define a helper function *FACT'*.

$$\begin{aligned} FACT' &\triangleq \lambda f.\, \lambda x.\, IF\ (ZERO?\ x)\ 1\ (*\ (f\ f\ (-\ x\ 1))\ x) \\ FACT &\triangleq FACT'\ FACT' \end{aligned}$$

Running *FACT* with argument 3 gives:

$$\begin{aligned} FACT\ 3 &\rightarrow FACT'\ FACT'\ 3 \\ &\rightarrow (\lambda x.\, IF\ (ZERO?\ x)\ 1\ (*\ (FACT'\ FACT'\ (-\ x\ 1))\ x))\ 3 \\ &\rightarrow IF\ (ZERO?\ 3)\ 1\ (*\ (FACT'\ FACT'\ (-\ 3\ 1))\ 3) \\ &\rightarrow (*\ (FACT'\ FACT'\ 2)\ 3) \\ &\rightarrow (*\ ((\lambda x.\, IF\ (ZERO?\ x)\ 1\ (*\ (FACT'\ FACT'\ (-\ x\ 1))\ x))\ 2)\ 3) \\ &\rightarrow (*\ (IF\ (ZERO?\ 2)\ 1\ (*\ (FACT'\ FACT'\ (-\ 2\ 1))\ 2))\ 3) \\ &\rightarrow (*\ (*\ (FACT'\ FACT'\ 1))\ 2)\ 3) \\ &\rightarrow (*\ (*\ (*\ (FACT'\ FACT'\ 0))\ 1)\ 2)\ 3) \\ &\rightarrow (*\ (*\ (*\ 1\ 1)\ 2)\ 3) \\ &\rightarrow 6 \end{aligned}$$

Recursion-removal trick

We can generalize what we did to *FACT* for any recursive function F that we need to remove the recursion from.

1. Add extra (first) argument f to the function ($F' = \lambda f.\, F$).
2. Replace all recursive references of F in F' with $(f\ f)$.
3. Use $F = F'\ F'$ as the recursive function.

3 Fixed point

It is possible to remove recursion in an even easier way, by letting the λ-calculus work for us.

$$\begin{array}{rcl} F & = & \lambda f.\,\lambda x.\,IF\ (ZERO?\ x)\ 1\ (*\ (f\ (-\ x\ 1))\ x) \\ (F\ FACT) & = & FACT \\ & \rightarrow & \lambda x.\,IF\ (ZERO?\ x)\ 1\ (*\ (FACT\ (-\ x\ 1))\ x) \end{array}$$

FACT is a fixed point of F. A fixed point for function f is defined as a point x such that $f(x) = x$. We can represent recursive functions as fixed points of higher order functions. In this way they don't require the self application $(f\ f)$, which makes it hard to assign a type.

Can we find a fixed point for any recursive function? We know that the set of all λ-terms is countable, but the set of functions λ-terms $\rightarrow$ λ-terms is not, so finding a fixed point for any arbitrary function is not trivial. But it is possible, given an arbitrary function F we can find its fixed point YF by:

$$\begin{array}{rcl} F(YF) & = & YF \\ F(YF)\ x & = & (YF)\ x \\ YF & = & \lambda x.\,F(YF)\ x \\ Y & = & \lambda f.\,\lambda x.\,f\ (Y\ f)\ x \\ Y' & \triangleq & \lambda y.\,\lambda f.\,\lambda x.\,f\ (y\ y\ f)\ x \\ Y & \triangleq & Y'\ Y' \end{array}$$

A more traditional form would be: (Note: only works in a Call-By-Value languages)

$$Y \ = \ \lambda f.\,\big((\lambda x.\,(f\ x\ x))\ (\lambda x.\,(f\ x\ x))\big)$$

This is a function that is occasionally useful for real programming problems, for example when using recursive modules in ML.

4 Substitution

In the previous lecture we defined β-reduction in Call-By-Name as: $(\lambda x.\,e_1)e_2 \rightarrow e_1\{e_2/x\}$
This definition is incomplete because we do not wish to replace every occurrence of x by e_2, but only those that are addressed by the exterior λx.
Many mathematicians including Church, Hilbert and even Newton used incomplete definitions such as this for substitution.
Example 1, use this rule on the integral $\int(1 + \int x\ dx)dy$ for y ranging from 0 to some value x and you get

$$\int_{y=0}^{y=x}(1+\int x\ dx)dy \ = \ (y + y\int x\ dx)|_{y=0}^{y=x} = (x + \int x^2\ dx) - 0 = (x + \int x^2\ dx)$$

Example 2, $(y(\lambda x.\,x\ y))\{x/y\} = (x(\lambda x.\,x\ x))$ while we meant it to translate to $(x(\lambda a.\,a\ x))$. This is called *Variable Capture*.

Lets define the *free variables* in an expression:

$$\begin{array}{rcl} FV(e) & = & \text{set of free variables in } e \\ FV(x) & = & \{x\} \\ FV(e_1\ e_2) & = & FV(e_1)\ \cup\ FV(e_2) \\ FV(\lambda x.\,e) & = & FV(e) \setminus \{x\} \end{array}$$

so our final definition of substitution is:

$$
\begin{array}{rcll}
x\{e/x\} & = & e & \\
x'\{e/x\} & = & x' & (\text{ where } x \neq x' \text{ }) \\
(e_1\ e_2)\{e/x\} & = & e_1\{e/x\}\ \ e_2\{e/x\} & \\
(\lambda x.\, e_0)\{e/x\} & = & \lambda x.\, e_0 & (\text{ because any } x \text{ in } e_0 \text{ is not the } x \text{ from outside the } \lambda x \text{ }) \\
(\lambda y.\, e_0)\{e/x\} & = & \lambda y.\, (e_0\{e/x\}) & (\text{ where } x \neq y,\ y \notin FV(e) \text{ }) \\
(\lambda y.\, e_0)\{e/x\} & = & \lambda y'.\, e_0\{y'/y\}\{e/x\} & (\text{ where } x \neq y,\ x \neq y',\ y' \notin FV(e) \text{ and } y' \notin FV(e_0) \text{ })
\end{array}
$$