CS611 Lecture 22 **Fixed Point Combinators** 20 October 2003

Scribe: Wojciech Moczydłowski, Jr and Adam Arbree Lecturer: Radu Rugina

1 Expressiveness of lambda calculus: Lists

We continue our task of showing how to encode various high-level programming constructs into lambda calculus. To define lists in a functional style, we need to define a list constructor (*cons*) and two destructors: *tail* and *head* and an empty list. This corresponds to the datatype (in ML syntax):

```
datatype ‘a List = Null | Cons of (‘a * (‘a List))
```

The list $[1, 2, 3]$ is represented as $cons\ c_1\ (cons\ c_2\ (cons\ c_3\ null))$. The naive implementation using pairs might look like:

$$\begin{aligned} cons &\equiv \lambda h.\lambda t.\ pair\ h\ t \\ head &\equiv \lambda l.\ fst\ l \\ tail &\equiv \lambda l.\ snd\ l \end{aligned}$$

However, we have no obvious way to define *null*. Note that *head* is operationally equivalent to *fst* — *head* k behaves as *fst* k. And the same can be said about *tail* and *snd*. We will say more about this relation (called η-reduction) later.

So how to encode lists with *null*? We simply represent list as a pair — the first element of pair tells us whether the list is empty, while the second is the list:

$$\begin{aligned} cons &\equiv \lambda h.\lambda t.\ pair\ false\ (pair\ h\ t) \\ head &\equiv \lambda l.\ fst\ (snd\ l) \\ tail &\equiv \lambda l.\ snd\ (snd\ l) \\ null &\equiv pair\ true\ true \\ isnull &\equiv \lambda l.\ fst\ l \end{aligned}$$

The second *true* in null serves only as a dummy. It should be easy now to see how to encode arbitrary datatypes, i.e. binary trees, any trees, etc.

2 Fixed point combinator

Let's see how we can do recursion in lambda calculus. We have a problem here, because functions are not named and we cannot use the schema we used in REC. Let's concentrate on the factorial function. We would like to say:

$$fact \equiv \lambda n.\ test\ (testz\ n)\ c_1\ (mul\ n\ (fact\ (dec\ n)))$$

However, the *fact* in the expression is not defined yet. What shall we do? First, let's define a function

$$F \equiv \lambda f.\ \lambda n.\ test\ (testz\ n)\ c_1\ (mul\ n\ (f\ (dec\ n)))$$

This is a well-defined lambda expression. Note that if we found a fixed point of this function, it would satisfy the equation above. So we could just define:

$$\mathit{fact} \equiv \mathit{fix}\ F$$

We can define a fix-point operator in lambda calculus, the Y combinator:

$$\mathit{fix} = Y \equiv \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x))$$

This looks rather mysterious. Note that there two cases of self-application in the body of Y — x applied to x and $(\lambda x.\ f\ (x\ x))$ applied to itself. What we expect from Y is that:

$$F(Y\ F) = Y\ F$$

A short proof demonstrates that this is true.

$$\begin{aligned} Y\ F &\rightarrow_1 (\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x)) \\ &\rightarrow_1 F\ \big((\lambda x.\ F\ (x\ x))\ (\lambda x.\ F\ (x\ x))\big) \\ &= F\ (Y\ F) \end{aligned}$$

It is not entirely clear what we mean by "=" here. We will come back to this topic later.

2.1 Other fixpoint combinators

Is the fixed point combinator unique? No! Consider the following exercise:

Find a fixed point combinator Q such that $Q = A\ A$, where A is of the form: $\lambda a.\lambda b.e$.

How to solve it? We must have: $Q\ F = A\ A\ F = e\ [a \rightarrow A]\ [b \rightarrow F] = F\ (A\ A\ F)$. We can guess that A should be defined as: $A = \lambda a.\lambda b.\ b\ (a\ a\ b)$ and check that indeed it works.[1]

3 Y Combinator Example

Consider a simple example. Let $g = \lambda x.\, c_1$, the constant function which returns c_1. Obviously, the fixed point of this function is c_1. The Y combinator should be able to find it by evaluating Yg:

$$\begin{aligned} Y &= \lambda f.\,(\lambda x.\,f(xx))(\lambda x.\,f(xx)) \\ g &= \lambda x.\,c_1 \\ A &= (\lambda x.\,(\lambda x.\,c_1)(xx)) \\ \\ Yg &\rightarrow_1 (\lambda x.\,g(xx))(\lambda x.\,g(xx)) \\ &\equiv (\lambda x.\,(\lambda x.\,c_1)(xx))(\lambda x.\,(\lambda x.\,c_1)(xx)) \\ &\rightarrow_1 (\lambda x.\,c_1)((\lambda x.\,(\lambda x.\,c_1)(xx))(\lambda x.\,(\lambda x.\,c_1)(xx))) \\ &= (\lambda x.\,c_1)(AA) \end{aligned}$$

Now the evaluation will proceed differently using Call-By-Value or Call-By-Name.

[1] Since we are using equality without defining it, it's not obvious at all whether this combinator is different from Y. The answer is yes, but the proof is nontrivial.

3.1 Call-By-Name

With Call-By-Name, a function can be evaluated without evaluating its arguments. Thus:

$$\begin{array}{rcl} Yg & = & (\lambda x.\, c_1)(AA) \\ & \rightarrow_1 & c_1 \end{array}$$

3.2 Call-By-Value

With Call-By-Value, the arguments, AA, must be evaluated first. From above,

$$\begin{array}{rcl} AA & = & (\lambda x.\, (\lambda x.\, c_1)(xx))(\lambda x.\, (\lambda x.\, c_1)(xx)) = Yg \\ & \rightarrow_1^* & (\lambda x.\, c_1)(AA) \end{array}$$

Thus Call-By-Value evaluation of Yg never ends:

$$Yg \quad = \quad (\lambda x.\, c_1)(\lambda x.\, c_1)(\lambda x.\, c_1)...$$

To fix this problem, the definition of the Y combinator must be changed. The recursive parts the combinator can be wrapped in abstractions to prevent their evaluation during Call-By-Value. Define:

$$Y_{CBV} \quad = \quad \lambda f.\, (\lambda x.\, f(\lambda y.\, xxy))(\lambda x.\, f(\lambda y.\, xxy))$$

This function remains a fixed point function since:

$$\begin{array}{rcl} Y_{CBV}F & \rightarrow_1 & (\lambda x.\, F(\lambda y.\, xxy))(\lambda x.\, F(\lambda y.\, xxy)) \\ & \rightarrow_1 & F(\lambda y.\, (\lambda x.\, F(\lambda y.\, xxy))(\lambda x.\, F(\lambda y.\, xxy))y) \\ & = & F(\lambda y.\, Y_{CBV}Fy) \\ & \sim & F(Y_{CBV}F) \end{array}$$

The final step is the η-reduction mentioned above, and the nature of the equivalence will be discussed in the next lecture.

4 Factorial Example Revisited

Finally, the factorial example can be evaluated. Now that we re-defined the Y combinator for CBV, we wonder if the following holds:

$$\mathit{fact}\ \ c_n = Y_{CBV}\ \ F\ \ c_n = c_{n!}$$

Consider evaluating the inner part of the factorial function for a particular c_n:

$$\mathit{test}\ (\mathit{testz}\ c_n)\ c_1\ (\mathit{mul}\ n\ (f\ (\mathit{dec}\ c_n)))$$

Evaluating the first *test* function requires the evaluation of its three terms, the third of which is the function itself, making the evaluation process infinite. So like the fixed point operator above, we are still not ready to evaluate the factorial by Call-By-Value. To make the final evaluation, we must further wrap parts of the factorial function in abstraction:

$$F \equiv \lambda f.\ \lambda n.\ \mathit{test}\ (\mathit{testz}\ n)\ (\lambda y.\, c_1)\ (\lambda y.\, \mathit{mul}\ n\ (f\ (\mathit{dec}\ n)))\ c_0$$