

## CM20167 Topic 10: Programming in the $\lambda$ -calculus

Guy McCusker

1W2.1

### Remember these claims?

Way back when we were discussing Lisp programming, I said that:

- ▶ if we have a *test for zero* and *successor and predecessor* (add and subtract one) operations, we can do a lot of computation with numbers
- ▶ if we have a *test for null* and *pairing and unpairing* operators, we can do a lot of computation with lists

and nobody complained.

(Look in the Topic 3 notes if you don't believe me.)

We will now see how the  $\lambda$ -calculus lets us do all of this, making it a fully expressive programming language.

### Lisp-like lists

The first thing we'll do is to encode Lisp-style lists in the  $\lambda$ -calculus.

In Lisp, a non-empty list is really a pair consisting of the `car` of the list, and another list which is the `cdr` of the list.

The main things we can do are to put an element and a list together, to form a list, using `cons`, and then split it apart again using `car` and `cdr`.

There's also a special constant, `nil`, which we think of as a list with no elements; and a test, `null`, which lets us tell whether something is `nil` or not.

## A secret about cons

In Lisp, cons isn't just for building lists!

It actually lets us make a pair containing any two values, and car and cdr give us back those two values. A pair is written in so-called "dotted pair" notation, i.e. as something like (a . b).

Lisp's printing function treats pairs where the second element is another pair specially:

- ▶ it doesn't print the dot and it doesn't print the next set of parentheses.
- ▶ if the cdr is the empty list, it prints nothing. That means lists get printed as lists rather than as pairs.

## Lists and pairs in Lisp

```
user> (cons 3 4)
(3 . 4)
user> (cons 3 (cons 4 nil))
(3 4)
user> (car '(3 . 4))
3
user> (cdr '(3 . 4))
4
```

Dotted-pair notation for an ordinary pair

List notation for a special pair i.e. a list

## Pairing

Let's see how to encode pairing, i.e. cons, in the  $\lambda$ -calculus.

What we are looking for is a *pairing operator* which takes two terms  $M$  and  $N$  and gives us a new term  $\langle M, N \rangle$ , together with *projection or selection* functions  $L$  and  $R$  which give us back the left and right components of a pair.

That is, we want

$$\begin{aligned} L(\langle M, N \rangle) &=_{\beta} M \\ R(\langle M, N \rangle) &=_{\beta} N \end{aligned}$$

Then  $\langle M, N \rangle$  is like (cons M N),  $L$  is like car and  $R$  is like cdr.

## Pairing

There are lots of ways to encode pairing in the  $\lambda$ -calculus, but the best known is to define the pairing combinator  $P$  by

$$P \equiv \lambda x. \lambda y. \lambda s. sxy.$$

The idea here is that  $s$  is going to be a *selector* which can choose between  $x$  and  $y$ .

Then given terms  $M$  and  $N$  we can define

$$\langle M, N \rangle \equiv P M N.$$

## Selectors

To extract the terms  $M$  and  $N$  from the pair  $\langle M, N \rangle$ , we need to pass it a selector.

By definition,  $\langle M, N \rangle \equiv \lambda s. s M N$ , so

$$\langle M, N \rangle (\lambda x. \lambda y. x) \rightarrow_{\beta} (\lambda x. \lambda y. x) M N \rightarrow_{\beta} (\lambda y. M) N \rightarrow_{\beta} M$$

and similarly

$$\langle M, N \rangle (\lambda x. \lambda y. y) \rightarrow_{\beta}^* N.$$

## Selectors

Then we can define

$$L \equiv \lambda p. p(\lambda x. \lambda y. x)$$

$$R \equiv \lambda p. p(\lambda x. \lambda y. y)$$

so that

$$L(\langle M, N \rangle) \equiv (\lambda p. p(\lambda x. \lambda y. x))(\langle M, N \rangle) \rightarrow_{\beta} \langle M, N \rangle (\lambda x. \lambda y. x) \rightarrow_{\beta}^* M$$

and similarly

$$R(\langle M, N \rangle) \rightarrow_{\beta}^* N.$$

So we have the pairing and selection operators we wanted!

## Booleans

According to my claim on the first slide, the other thing we need for list programming is to be able to *test for null*.

To test for anything, we need some notion of truth value: that is, we'd like to have values  $T$  and  $F$  to represent true and false, and a *conditional* operator  $C$  such that

$$\begin{aligned}C T M N &=_{\beta} M \\C F M N &=_{\beta} N\end{aligned}$$

The idea is that  $C B M N$  is like

if  $B$  then  $M$  else  $N$ .

## Conditional is like selection from a pair

Notice that the conditional operator we wanted above is very similar to the selection operators we wanted for pairs.

You give  $C$  a pair of terms, and a truth value (although not in that order), and it picks one of the terms based on the truth value you give it.

So we could try the following encoding:

$$\begin{aligned}T &\equiv L \\F &\equiv R \\C &\equiv \lambda b.\lambda m.\lambda n.b(\langle m, n \rangle)\end{aligned}$$

This works: for instance

$$C T M N \equiv L(\langle M, N \rangle) \rightarrow_{\beta}^* M.$$

## Booleans, again

There's quite a bit of unnecessary computation in the reduction of  $C T M N$  with this definition.

We can abbreviate it slightly: if we change our definition of  $T$  and  $F$  so that

$$\begin{aligned}T &\equiv \lambda x.\lambda y.x \\F &\equiv \lambda x.\lambda y.y\end{aligned}$$

and also change  $C$ :

$$C \equiv \lambda b.\lambda m.\lambda n.b m n$$

then we get the same behaviour:

$$C T M N \equiv (\lambda b m n. b m n)(\lambda x y. x) M N \rightarrow_{\beta}^* (\lambda x y. x) M N \rightarrow_{\beta}^* M.$$

## Booleans, again, again

Someone pointed out the other day that we can simplify still further: the conditional operator  $C$  can be defined as

$$C \equiv \lambda x. x$$

i.e. the identity function. This works because, for example, it means that

$$C F M N \equiv ((\lambda x. x) F) M N \rightarrow_{\beta} FMN \rightarrow_{\beta}^* N.$$

## Test for null

To complete our encoding of pairing, lists and so on, we need two more things:

- ▶ a value that behaves like “nil”, i.e. something that is not a pair
- ▶ a test which gives  $F$  when applied to a pair, and  $T$  when applied to our “nil” value.

One way to do this is:

$$\begin{aligned} \text{nil} &\equiv \lambda x. T \\ \text{null} &\equiv \lambda p. p(\lambda x. \lambda y. F) \end{aligned}$$

## Let's see if it works

What we want is

$$\begin{aligned} \text{null}(\text{nil}) &=_{\beta} T \\ \text{null}(\langle M, N \rangle) &=_{\beta} F. \end{aligned}$$

Does it work?

$$\text{null}(\text{nil}) \equiv (\lambda p. p(\lambda xy. F))(\lambda x. T) \rightarrow_{\beta} (\lambda x. T)(\lambda xy. F) \rightarrow_{\beta} T.$$

$$\begin{aligned} \text{null}(\langle M, N \rangle) &\equiv (\lambda p. p(\lambda xy. F))(\langle M, N \rangle) \\ &\rightarrow_{\beta} \langle M, N \rangle(\lambda xy. F) \\ &\rightarrow_{\beta}^* (\lambda s. s M N)(\lambda xy. F) \\ &\rightarrow_{\beta} (\lambda xy. F) M N \\ &\rightarrow_{\beta}^* F. \end{aligned}$$

## Encoding of lists

Since we've got pairs and a nil value, we can now encode a list: given terms  $M_1, \dots, M_n$ , the term

$$\langle M_1, \langle M_2, \langle \dots, \langle M_k, \text{nil} \rangle \rangle \rangle \rangle$$

encodes the list  $(M_1 M_2 \dots M_k)$

## Recursion

Now that we can encode pairs, and nil, and test for nil, we can almost do everything we need to do to write Lisp-style programs.

The other thing we need is recursion.

In the  $\lambda$ -calculus, recursion can be encoded by means of *fixed-point combinators*.

A fixed-point combinator is a term  $Y$  such that

$$YM =_{\beta} M(YM)$$

for any term  $M$ .

## Encoding recursion with fixed-points

We've already seen how fixed-point combinators in Lisp can simulate recursion, but it's worth looking at again.

Suppose we want to write the map function, i.e. we want to be able to apply some  $\lambda$ -term  $F$  to every element of a "list".

If we had recursion in the  $\lambda$ -calculus, we could define something like

$$\text{map} \equiv \lambda f. \lambda l. C (\text{null } l) \text{ nil } \langle f (L l), \text{map } f (R l) \rangle$$

But we don't have recursion, so instead we define

$$\text{map} \equiv Y(\lambda m. \lambda f. \lambda l. C (\text{null } l) \text{ nil } \langle f (L l), m f (R l) \rangle)$$

## Encoding recursion with fixed-points

The defining property of fixed-point combinators means that

$$\text{map} =_{\beta} (\lambda m. \lambda f. \lambda l. C (\text{null } l) \text{ nil } \langle f (L l), m f (R l) \rangle)(\text{map})$$

which reduces in one step to

$$\lambda f. \lambda l. C (\text{null } l) \text{ nil } \langle f (L l), \text{map } f (R l) \rangle$$

i.e. implements the required recursion.

## Fixed-point combinators

But how can we define such a combinator  $Y$ ?

In fact there are infinitely many of them. The best known is the one discovered by Curry and most commonly called  $Y$ :

$$Y \equiv \lambda f. (\lambda x. f(xx))(\lambda x. f(xx))$$

We can easily check that it has the required property:

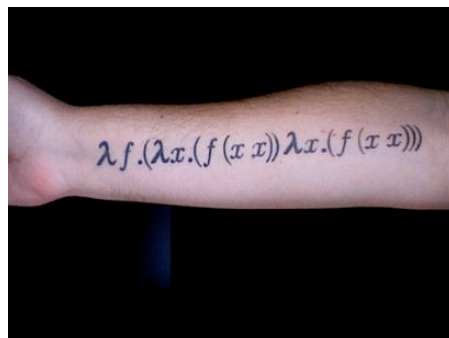
$$\begin{aligned} YM &\equiv (\lambda f. (\lambda x. f(xx))(\lambda x. f(xx)))M \\ &\rightarrow_{\beta} (\lambda x. M(xx))(\lambda x. M(xx)) \\ &\rightarrow_{\beta} M((\lambda x. M(xx))(\lambda x. M(xx))) \end{aligned}$$

so that  $YM =_{\beta} M(YM)$ .

Note that this is a beta-equality but we don't have  $YM \rightarrow_{\beta}^* M(YM)$ .

## $\lambda$ -calculus drives people mad

From <http://www.paulgraham.com>:



Why oh why?



## Note on Klop's combinator

Remember that

$$\lambda abcde \dots$$

means

$$\lambda a. \lambda b. \lambda c. \lambda d. \lambda e. \dots$$

and that

*thisisafixedpointcombinator*

means

$$((((((t\ h)\ i)\ s)\ i)\ s)\ \dots$$

### Exercise

Check that Klop's combinator works. Hint: the phrase "this is a fixed point combinator" contains 27 letters.

## Encoding numbers

Back to encoding programming constructs.

Obviously a very important thing to have in a programming language is numbers.

In mathematics, the natural numbers are often defined using *Peano arithmetic*, which stipulates that there is a constant 0 and an operator *succ*, for *successor* ("add one"), and then a number like 5 is represented as

$$\text{succ}(\text{succ}(\text{succ}(\text{succ}(\text{succ}(0))))))$$

This is not much different from a list with five *succs* followed by a zero symbol; and of course it doesn't matter what these symbols are as long as we can tell them apart. Let's use that idea to encode numbers.

## Barendregt numerals

Barendregt's definition of an encoding of numbers is given by the equations:

$$\begin{aligned} \bar{0} &\equiv \lambda x. x \\ \overline{(n+1)} &\equiv \langle F, \bar{n} \rangle \end{aligned}$$

So for instance 3 is the term

$$\langle F, \langle F, \langle F, \lambda x. x \rangle \rangle \rangle.$$

We're using *F*, the "false" value, as the marker for a number which is non-zero, and the identity function as zero.

## Successor and predecessor

We need to encode the function which adds one to a number, the function which subtracts one from a number, and the function which tests a number for equality to zero.

Successor drops straight out of the definition: since

$$\overline{n+1} \equiv \langle F, \bar{n} \rangle$$

we can define

$$S \equiv \lambda n. \langle F, n \rangle$$

and it is clear that

$$S(\bar{n}) =_{\beta} \overline{n+1}.$$

Predecessor is also easy: it is encoded by the selector  $R$ , because the right hand side of the pair that represents a number represents that number's predecessor.

## Zerop

To define the zero-test, notice that  $\bar{0} \equiv \lambda x. x$ , so

$$\bar{0}(T) \rightarrow_{\beta} T.$$

Also notice that any non-zero number is a pair  $\langle F, \dots \rangle$ , and by definition this is the same as

$$\lambda p. p F \text{ (something)}$$

But  $T$  is  $\lambda xy. x$ , so applying this pair to  $T$  gives us

$$(\lambda p. p F \text{ (something)})(\lambda xy. x) \rightarrow_{\beta} (\lambda xy. x) F \text{ (something)} \rightarrow_{\beta}^* F.$$

## Zerop

This means we can define

$$Z \equiv \lambda n. n T$$

and then we have

$$\begin{aligned} Z \bar{0} &=_{\beta} T \\ Z \overline{n+1} &=_{\beta} F. \end{aligned}$$

## Addition and so on

We saw how to define addition using recursion in Lisp:

```
(defun o+ (n m)
  (if (zerop m)
      n
      (add1 (o+ n (sub1 m)))))
)
```

We can therefore do the same in the  $\lambda$ -calculus:

$$Y(\lambda a.\lambda n.\lambda m.C (Z m) n (S (a n (R m))))).$$

## Church numerals

The original encoding of numbers which Church gave was quite different from Barendregt's: Church encoded a number  $n$  as

$$\lambda f.\lambda x.\underbrace{f(f(\dots(f(x))))}_n$$

That is,

$$\begin{aligned}\bar{0} &\equiv \lambda f.\lambda x.x \\ \bar{1} &\equiv \lambda f.\lambda x.fx \\ \bar{2} &\equiv \lambda f.\lambda x.f(fx) \\ \bar{3} &\equiv \lambda f.\lambda x.f(f(fx)) \\ &\vdots\end{aligned}$$

## Church numerals

This encoding has the advantage that addition is very easy to define:

$$\lambda m.\lambda n.\lambda f.\lambda x.m f (n f x)$$

There's a direct encoding of predecessor too, but it's a bit strange:

$$\lambda n.\lambda f.\lambda x.n (\lambda g.\lambda h.h (g f)) (\lambda u.x) (\lambda u. u)$$

I won't try to explain how this works. Try playing with it!

## Church numerals

Suffice it to say that we can do everything we need with Church numerals, just as we can with Barendregt numerals.

We can even translate from one system to the other quite easily: for instance, given the Church numeral  $\bar{n}$ , the term

$$\bar{n} S (\lambda x.x)$$

applies  $S$  (the Barendregt successor function)  $n$ -times to  $\lambda x.x$  (the Barendregt zero), so it reduces to the Barendregt encoding of  $n$ .

### Exercise

Define the other direction of the translation, i.e. from Barendregt numerals to Church numerals.

## Turing complete!

Since we can express numerals, arithmetic and recursion, we have a Turing-complete programming language.

“Turing-complete” means “capable of expressing any computable function on natural numbers.”

This also means that the Halting Problem rears its head: for instance, there is no algorithm for checking whether two terms are related by  $=_{\beta}$ .

## No test for normal forms

A simple version of the Halting problem is to test whether a term has a normal form. This cannot be done.

Let's suppose there is a term  $N$  such that  $NM =_{\beta} T$  if  $M$  has a normal form, and  $NM =_{\beta} F$  if not. We will show that this leads to a contradiction.

Recall that  $\omega \equiv \lambda x.xx$  and  $\Omega \equiv \omega\omega$ .

We will consider the term  $Z \equiv \lambda z.N(\omega z)\Omega(\lambda x.x)$ .

We have:

$$\omega Z \rightarrow_{\beta} ZZ \rightarrow_{\beta} N(\omega Z)\Omega(\lambda x.x)$$

so that  $\omega Z$  has a normal form if and only if  $N(\omega Z)\Omega(\lambda x.x)$  does (by Church-Rosser).

## No test for normal forms

If  $\omega Z$  has a normal form, then  $N(\omega Z) =_{\beta} T$ , so

$$N(\omega Z)\Omega(\lambda x.x) =_{\beta} T\Omega(\lambda x.x) =_{\beta} \Omega$$

which has no normal form. This is a contradiction.

If on the other hand  $\omega Z$  has no normal form, then  $N(\omega Z) =_{\beta} F$  so

$$N(\omega Z)\Omega(\lambda x.x) =_{\beta} F\Omega(\lambda x.x) =_{\beta} \lambda x.x$$

which is in normal form; this is also a contradiction,

Either way we get a contradiction, so no such  $N$  can exist.

Is this a shame or a very good thing?

## No test for $\beta$ -equality

We can also show that there is no  $\lambda$ -term which can test terms for  $\beta$ -equality.

First we need a lemma. Suppose that  $\mathcal{S}$  is a set of  $\lambda$ -terms which is closed under  $=_{\beta}$  (i.e. if  $M \in \mathcal{S}$  and  $M =_{\beta} N$  then  $N \in \mathcal{S}$ ), and  $H$  is a  $\lambda$ -term which tests for membership of  $\mathcal{S}$ , that is

$$\begin{array}{ll} HM =_{\beta} T & \text{if } M \in \mathcal{S} \\ HM =_{\beta} F & \text{if } M \notin \mathcal{S} \end{array}$$

Then either  $\mathcal{S}$  contains all  $\lambda$ -terms or  $\mathcal{S}$  is empty.

This is a bit like *Rice's theorem* (aka the Rice-Myhill-Shapiro theorem) from computability theory.

## Proof of the lemma

Suppose  $\mathcal{S}$  is neither empty nor contains all the  $\lambda$ -terms, so that there are terms  $M_1$  and  $M_2$  with  $M_1 \in \mathcal{S}$  and  $M_2 \notin \mathcal{S}$ . We will show that this is impossible (proof by contradiction).

Consider the  $\lambda$ -term  $G$  defined by

$$\lambda x.C (Hx) M_2 M_1.$$

Notice that if  $M \in \mathcal{S}$  then  $GM =_{\beta} M_2 \notin \mathcal{S}$  and if  $M \notin \mathcal{S}$  then  $GM =_{\beta} M_1 \in \mathcal{S}$ .

But the term  $YG$  is  $\beta$ -equal to  $G(YG)$ , so we have that  $YG \in \mathcal{S}$  if and only if  $YG =_{\beta} G(YG) \notin \mathcal{S}$ , which is impossible.

## No test for $\beta$ -equality

Now lets imagine there is a term  $H$  which tests for  $\beta$ -equality i.e.

$$\begin{array}{ll} H M N =_{\beta} T & \text{if } M =_{\beta} N \\ H M N =_{\beta} F & \text{if } M \neq_{\beta} N \end{array}$$

Choose any  $\lambda$ -term  $M$  and let  $S$  be the set of terms  $N$  such that  $H M N =_{\beta} T$ . By our lemma,  $S$  must be empty or must contain all  $\lambda$ -terms.

But we know  $S$  is not empty, since it contains  $M$ .

And we know  $S$  does not contain every  $\lambda$ -term, since  $=_{\beta}$  is consistent.

So  $H$  cannot exist.