ADVANCED PROGRAMMING LANGUAGES

Fall 2023, CM 0632. MSc in Computer Science and Information Technology, Ca' Foscari University of Venice

The expressive power of the Lambda Calculus

We closed our last lecture mentioning the Church-Turing Thesis about the different characterizations of the notion of computable functions. A you (should) know, the thesis has remained without a proof, but the various cross-encodings that proved the equivalent expressive power of lambda-calculus and other models of computation has long been considered as a *de-facto* proof.

Instead of discussing such cross-encodings, below we illustrate the power of the Lambda Calculus by showing it at work on the encoding of simple datatypes (booleans and numerals) together with their operators.

Church Booleans. We start with Booleans and conditional expressions.

$$\begin{aligned} \mathsf{true} &\equiv \lambda x.\lambda y.x \\ \mathsf{false} &\equiv \lambda x.\lambda y.y \\ \mathsf{cond} &\equiv \lambda x.\lambda y.\lambda z(x\ y\ z) \end{aligned}$$

As we see, true and false are represented as functions that take two arguments and return, respectively, the first and the second argument. Though these definitions may appear surprising at first, you can make sense of them by looking at the encoding of the conditional and verifying that $\mathsf{cond}\ M\ N\ P$ corresponds to the application of M to N and P, which in turn behaves as the more familiar $\mathsf{if}\ M\ \mathsf{then}\ N\ \mathsf{else}\ P$, expression whenever M is either true and false.

Given that true and false choose the first and second argument, they may be combined to achieve the expected behavior of the logical operator, similarly to how they are employed to select the branches of the conditional.

$$\begin{aligned} \mathsf{and} &\equiv \lambda x.\lambda y.x\ y\ \mathsf{false} \\ \mathsf{or} &\equiv \lambda x.\lambda y.x\ \mathsf{true}\ y \\ \mathsf{not} &\equiv \lambda p.p\ \mathsf{false}\ \mathsf{true} \end{aligned}$$

We look at the definitions at work on few examples:

$$
\begin{aligned}
\text{and true false} &\Rightarrow\Rightarrow \text{true false false} \\
&\Rightarrow\Rightarrow \text{false} \\
\text{or true false} &\Rightarrow\Rightarrow \text{true true false} \\
&\Rightarrow\Rightarrow \text{true} \\
\text{not true} &\Rightarrow\Rightarrow \text{true false true} \\
&\Rightarrow\Rightarrow \text{false}
\end{aligned}
$$

Church Numerals. Next we show the encoding of integers by means of the well-known Church numerals. The underlying intuition is to use the structure of lambda terms to *count*.

Specifically, each number is represented as a lambda abstraction that expects two arguments — a function and an argument – and applies the function a corresponding number of times. Thus, given any function f and argument x, we have:

$$
\begin{aligned}
&\mathsf{0} \text{ applies } f \text{ to } x \text{ zero times}: && \mathsf{0}\, f\, x = x \\
&\mathsf{1} \text{ applies } f \text{ to } x \text{ one time}: && \mathsf{1}\, f\, x = f\, x \\
&\mathsf{2} \text{ applies } f \text{ to } x \text{ two times}: && \mathsf{2}\, f\, x = f(f\, x) \\
&\mathsf{3} \text{ applies } f \text{ to } x \text{ two times}: && \mathsf{3}\, f\, x = f(f(f\, x)) \\
&\ldots \text{ and so on } \ldots && \qquad \ldots
\end{aligned}
$$

The encoding can therefore be defined as follows:

$$
\begin{aligned}
\mathsf{0} &\equiv \lambda f.\lambda x.x \\
\mathsf{1} &\equiv \lambda f.\lambda x.(f\, x) \\
\mathsf{2} &\equiv \lambda f.\lambda x.f(f\, x) \\
\mathsf{3} &\equiv \lambda f.\lambda x.f(f(f\, x)) \\
&\vdots \qquad\qquad \vdots \\
\mathsf{n} &\equiv \lambda f.\lambda x.(f^n\, x)
\end{aligned}
$$

The arithmetic operators can then be represented as functions over the Church numerals.

- **Successor.** Given any numeral n, $\mathsf{succ\ n}$ is the function that, given f and x, applies f to x n+1 times. Now, $\mathsf{n}\ f\ x = f^n x$, so that $\mathsf{succ\ n} = \lambda f.\lambda x.f(\mathsf{n}\ f\ x)$. Hence:

  $\mathsf{succ} = \lambda n.\lambda f.\lambda x.f(n\ f\ x)$

- **Plus**. Given any two numeral m and n, $\mathsf{plus\ m\ n}$ is the function that, given f and x, applies f to x m+n times. Again, $\mathsf{n}\ f\ x = f^n x$: thus to obtain the result of the addition, we must

apply f m times on top of the n times accounted for by $\mathsf{n}\ f\ x$. But that can be accomplished by the application $\mathsf{m}\ f\ (f^n x) = \mathsf{m}\ f\ (\mathsf{n}\ f\ x)$. Hence

$$\mathsf{plus} = \lambda m.\lambda n.\lambda f.\lambda x.m\ f(n\ f\ x)$$

- **Mult**. In this case, given m and n, we must define a function that, given f (and x) applies f (to x) m∗n times. To see how that can be accomplished, first note that, by definition, $\mathsf{n}\ f = \lambda x.f^n x$. By η-equality, we have $\mathsf{n}\ f = f^n$ and then the desired result may be obtained by m applications of $(\mathsf{n}\ f)$ to x. Hence:

$$\mathsf{mult} = \lambda m.\lambda n.\lambda f.\lambda x.m\ (n\ f)\ x$$

Exercises

1. Practice with the definitions. Try and play with the definitions of the Church booleans and numerals with their operators to verify that their evaluation behaves as expected. For instance, check the following:

- $\mathsf{and}(\mathsf{not\ true})\ (\mathsf{or\ false\ true}) = \mathsf{false}$
- $\mathsf{plus}\ 3\ 4 = 7$
- $\mathsf{mult}\ 3\ 2 = 6$
- . . .

2. xor. Define the lambda calculus encoding of the logical operator XOR, which is true only when its two arguments have different truth values.

3. isZero. Define a lambda term to encode the isZero test function: given a numeral n the function returns true if n is 0, otherwise it returns false

4. Church pairs. In class we looked at a direct encoding of booleans and conditionals. Here we study a different approach that draws on pairs as the main building block. We use the following notation:

- def *fun* = M to introduce *fun* as a name for the λ term M
- M = N to denote that M evaluates to N (with some reduction order)

Pairs are encoded as the following structured lambda term:

def *pair* = λ fst. λ snd. λ f. (f fst snd)

A pair is a function that takes its two components arguments and a function which it applies to the components, so that

pair a b = λ f. (f a b)

Now we can introduce the two general selectors:

def *select-first* = λ fst. λ snd. fst
def *select-second* = λ fst. λ snd. snd

From them, we obtain two pair selectors by applying the pair constructor to the general selectors:

def *first* = λ p. (p *select-first*)
def *second* = λ p. p (*select-second*)

Exercise 4.1. Verify that:
first (*pair* a b) = a
second (*pair* a b) = b

Exercise 4.2. Write a version of *pair* called *fun-pair* that takes five arguments including f,g,x,y and makes a pair with first element f(x) and second g(y).

5. More Booleans

In class we have defined the boolean operators *and*, *or* and *not* from first principles. Here we look at a different encoding, which we build starting from the encoding of the conditional expression.

We start with *not*. Written as a C-like conditional, one has: *not* x = x ? *false* : *true*. Thus:

def *not* = λx. (*cond false true* x)

Using β reduction, the expression (*not* x) can be simplified to (x *false true*), which leads us to the simpler definition we have seen in class.

not = λx. (x *false true*)

We can repeat the same reasoning with *and* and *or*. For the former we observe that *and*(x,y) = x ? y : false, whence:

def *and* = λx. λy. (*cond* y *false* x)

For the latter from *or*(x,y) = x ? *true* : y

def *or* = λx. λy. (*cond true* y x)

Simplifying we have again the lambda terms discussed in class

and = λx. λy. (x y *false*)
or = λx. λy. (x *true* y)

Exercise 5.1 Repeat the above encoding of boolean operators with a new operator *nand* (¬(P ∧Q)). That is:

- Write a C-like x ? y : z conditional expression that implements the Boolean function *nand*. Only use the variables x, y the operator *not* and the constant *false* in your conditional expression.
- Convert your expression for *nand* into a λ-calculus *cond* expression: it should have two λs, one *cond*, and however many *false*, *not*, x, ys you need.
- Evaluate and simplify the inner body of this expression (removing *cond*) to get a more elegant function for *nand* of the form λx. λy. (…)
- Now apply the simplified expression for *nand* to the two arguments (x=*false*, y=*true*). Show it evaluates to the correct answer. Since the expression will involve strictly *true*, *false*, *not*, can convert to all *true*, *false*. Then remember they are *select-first*, *select-second*, and you can do the job in one line. No need to expand to λ-level.

Exercise 5.2. Show that the following two terms evaluate to the same thing for the Boolean argument pair (x,y) = (true, false).

- (*nand* x y)
- (*or* (*not* x) (*not* y))