 Expressiveness of Lambda Calculus 18 October, 2002

Scribe: Veselin Stoyanov and Vivek Vishnumurthy Lecturer: Radu Rugina

In this lecture, we will show that Lambda Calculus is expressive enough to allow us to represent a range of common primitives, such as booleans and integers, and common constructs, such as boolean and arithmetic operations, conditionals, functions with multiple arguments, and pairs. We can encode all of these using only λ-calculus constructs, i.e., only abstractions and applications.

1 Functions with multiple arguments

We would like to express functions of the form

$$\begin{array}{rcl} e := \ldots & | & e_0(e_1, \ldots, e_n) \\ & | & \lambda(x_1, \ldots, x_n).\, e \end{array}$$

We can express the above constructs in terms of functions with single arguments by currying.

$$\begin{array}{rcll} \lambda(x_1, \ldots, x_n).\, e & \to & \lambda x_1.\, \lambda x_2.\, \ldots \lambda x_n.\, e & \\ & = & \lambda x_1.\, (\lambda x_2.\, \ldots (\lambda x_n.\, e)) & \text{(by default)} \end{array}$$

$$\begin{array}{rcll} e_0(e_1, \ldots, e_n) & \to & e_0 e_1 e_2 \ldots e_n & \\ & = & (((e_0 e_1) e_2) \ldots e_n) & \text{(because application is left associative)} \end{array}$$

Having complex constructs in the language, which can be expressed in terms of simpler constructs is known as *Syntactic Sugar*. *Desugaring* is the process of expressing complex constructs in terms of the simpler constructs.

Another example of syntactic sugar is the *let* construct:

$$let\ x = e_1\ in\ e_2 \to (\lambda x.\, e_2) e_1$$

Note: In the above example we do not need * to lift the domain because if the evaluation of e_1 does not terminate then the entire evaluation of the function does not terminate.

2 Booleans

We will use the following definitions for the boolean constants *true* and *false*.

$$\begin{array}{rcl} true & = & \lambda x.\, \lambda y.\, x \\ false & = & \lambda x.\, \lambda y.\, y \end{array}$$

Note: *true* and *false* are not variables, but only short-hand notations for the λ-expressions on the right-hand side. The booleans defined above are known as *Church Booleans*.

Now we define *test*, a conditional expression that uses the booleans just defined.

$$test = \lambda b.\, \lambda e_0.\, \lambda e_1.\, b\ e_0\ e_1$$

Now when *test* is applied to *true* and two expressions, we have:

$$
\begin{array}{rcl}
\mathit{test}\ \mathit{true}\ e_0'\ e_1' & = & (\lambda b.\, \lambda e_0.\, \lambda e_1.\, b\ e_0\ e_1)\mathit{true}\ e_0'\ e_1' \\
& \rightarrow_1 & (\lambda e_0.\, \lambda e_1.\, \mathit{true}\ e_0\ e_1)e_0'\ e_1' \\
& \rightarrow_1^{CBN} & (\lambda e_1.\, \mathit{true}\ e_0'\ e_1)e_1' \\
& \rightarrow_1^{CBN} & \mathit{true}\ e_0'\ e_1' \\
& = & (\lambda x.\, \lambda y.\, x)e_0'\ e_1' \\
& \rightarrow_1^{CBN} & (\lambda y.\, e_0')e_1' \\
& \rightarrow_1^{CBN} & e_0'
\end{array}
$$

Note that we used CBN to reduce the above expression. Using CBV would yield the same result (if the evaluation of e_1' terminates), the only difference being that CBV evaluates both e_0' and e_1' although the value of e_1' is not being used.

Similarly, we can check that:

$$\mathit{test}\ \mathit{false}\ e_0'\ e_1' \rightarrow_1^* e_1'$$

Now we see that the definition of booleans has allowed us to define the function *test* that is semantically similar to IMP's *if* and REC's *ifnz* constructs.

We will now see how we can define operations on booleans.

2.1 Conjunction

$$\mathit{and} = \lambda b_0.\, \lambda b_1.\, b_0\ b_1\ \mathit{false}$$

Now we check that this definition of *and* is a valid conjunction for Church booleans.

$$
\begin{array}{rcl}
\mathit{and}\ \mathit{true}\ \mathit{true} & = & (\lambda b_0.\, \lambda b_1.\, b_0\ b_1\ \mathit{false})\mathit{true}\ \mathit{true} \\
& \rightarrow_1 & (\lambda b_1.\, \mathit{true}\ b_1\ \mathit{false})\mathit{true} \\
& \rightarrow_1 & \mathit{true}\ \mathit{true}\ \mathit{false} \\
& = & (\lambda x.\, \lambda y.\, x)\mathit{true}\ \mathit{false} \\
& \rightarrow_1 & (\lambda y.\, \mathit{true})\mathit{false} \\
& \rightarrow_1 & \mathit{true}
\end{array}
$$

Similarly,

$$
\begin{array}{rcl}
\mathit{and}\ \mathit{false}\ \mathit{true} & \rightarrow_1^* & \mathit{false}\ \mathit{true}\ \mathit{false} \\
& \rightarrow_1^* & \mathit{false}
\end{array}
$$

We can check that *and* is a valid conjunction in the two remaining cases.

Disjunction for Church booleans can be defined similarly.

3 Pairs

Here we define *pair*, which is similar to a structure with fields in a high-level programming language.

$$\mathit{pair} = \lambda f.\, \lambda s.\, \lambda b.\, b\ f\ s$$

The expression *pair* is a function which takes two parameters, f (the first element) and s (the second element) and returns a function that takes b as argument and applies it to f and s. Intuitively, b is a boolean that allows us to select one of the elements in the pair.

Then, the expression *pair* u v constructs a pair consisting of elements u and v.

Now we define:

$$\begin{array}{rcl} \mathit{fst} & = & \lambda p.\, p\ \mathit{true} \\ \mathit{snd} & = & \lambda p.\, p\ \mathit{false} \end{array}$$

We can check that *fst* and *snd* have the properties we need:

$$\begin{array}{rcl} \mathit{fst}(\mathit{pair}\ u\ v) & = & (\lambda p.\, p\ \mathit{true})(\mathit{pair}\ u\ v) \\ & \to_1^{CBN} & (\mathit{pair}\ u\ v)\mathit{true} \\ & \to_1^* & \mathit{true}\ u\ v \\ & \to_1^* & u \end{array}$$

Using CBV, *fst* evaluates to the same result (if evaluation of v terminates), but the intermediate steps differ:

$$\begin{array}{rcl} (\lambda p.\, p\ \mathit{true})((\lambda f.\, \lambda s.\, \lambda b.\, b\ f\ s)u\ v) & \to_1^* & (\lambda p.\, p\ \mathit{true})(\lambda b.\, b\ u\ v) \\ & \to_1^* & \mathit{true}\ u\ v \end{array}$$

This evaluates to the same result (u).

We can similarly show that *snd* selects the second element of the pair.

4 Natural Numbers

We define natural numbers as follows:

$$\begin{array}{rcl} c_0 & = & \lambda s.\, \lambda z.\, z \\ c_1 & = & \lambda s.\, \lambda z.\, s\ z \\ c_2 & = & \lambda s.\, \lambda z.\, s(s\ z) \\ c_3 & = & \lambda s.\, \lambda z.\, s(s(s\ z)) \\ & \vdots & \end{array}$$

Number n is represented in λ-calculus by the combinator c_n. The combinators defined above are known as *Church numerals*. Intuitively, the s and z parameters stand for "successor" and "zero", and c_n applies n times s to z. We next look at operations on natural numbers.

4.1 The Increment Function

$$\mathit{inc} = \lambda n.\, \lambda s.\, \lambda z.\, s(n\ s\ z)$$

The above expression can be read as: "apply n times the successor function s to z, then apply s to the result".

For example:

$$\begin{array}{rcll} \mathit{inc}\ c_2 & = & (\lambda n.\, \lambda s.\, \lambda z.\, s(n\ s\ z))c_2 & \\ & \to_1 & \lambda s.\, \lambda z.\, s(c_2\ s\ z) & \\ & = & \lambda s.\, \lambda z.\, s((\lambda s'.\, \lambda z'.\, s'(s'\ z'))\ s\ z) & (*) \\ & \to_1^* & \lambda s.\, \lambda z.\, s(s(s\ z)) & (**) \\ & = & c_3 & \end{array}$$

Note: here that expression (*) does not reduce to expression (**) using either CBV or CBN operational semantics, as neither CBV nor CBN β-reduces expressions inside abstractions. To reduce expression (*) to (**), we need an evaluation order different than CBV or CBN, which will be be discussed later in the course.

In general, a proof by induction is required to show that $\mathit{inc}(c_n) = c_{n+1}$.

4.2 Addition

$$add = \lambda m.\, \lambda n.\, \lambda s.\, \lambda z.\, m\ s(n\ s\ z)$$

For example,

$$\begin{aligned} add\ c_2\ c_3 &\to_1^* \lambda s.\, \lambda z.\, c_2\ s(c_3\ s\ z) \\ &\to_1^* \lambda s.\, \lambda z.\, s(s(c_3\ s\ z)) \\ &\to_1^* \lambda s.\, \lambda z.\, s(s(s(s(s\ z)))) \\ &= c_5 \end{aligned}$$

Note: As in the case of *inc*, the above reductions do not correspond to the CBV or CBN evaluation orders, because they reduce expressions inside abstractions.

Another way to express *add* is:

$$add = \lambda m.\, \lambda n.\, m\ inc\ n$$

Intuitively, *add* applies m times *inc* to n.

4.3 Multiplication

$$mul = \lambda m.\, \lambda n.\, m(add\ n)c_0$$

Intuitively, *mul* applies m times *add* n (the function that adds n to its argument) to zero.

4.4 Subtraction

The definition of subtraction is not as simple as that of addition. Subtraction can be expressed as follows:

$$\begin{aligned} zz &= pair\ c_0\ c_0 \\ ss &= \lambda p.\, pair(snd\ p)(inc(snd\ p)) \\ dec &= \lambda m.\, fst(m\ ss\ zz) \\ sub &= \lambda m.\, \lambda n.\, n\ dec\ m \end{aligned}$$

Here's why this works. First, note that $ss(c_i, c_j)$ evaluates to $(c_j, c_j + 1)$. Hence, if we start with $zz = (c_0, c_0)$ and we apply m times ss, we get the following sequence of pairs:

$$\begin{gathered} (c_0, c_0) \\ \downarrow \\ (c_0, c_0 + 1) \\ \downarrow \\ (c_0 + 1, c_0 + 2) \\ \downarrow \\ \vdots \\ \downarrow \\ (m - 1, m) \end{gathered}$$

Finally, the *dec* function selects the first element of the the final pair $(m - 1, m)$, and *sub* applies n times *dec* to m.

Note that *dec* applied to c_0 evaluates to c_0.

4.5 Comparison operators

We would like to have some means to compare natural numbers. ie., we want to have constructs that are semantically similar to expressions such as $m = n$, $m \leq n$ and so on. We present here one such comparison operator, which tests if a number is zero or not.

We define *testz* as follows:

$$testz = \lambda m.\, m(\lambda x.\, \mathit{false})\, true$$

Intuitively, *testz* applies m times ($\lambda x.\mathit{false}$) to *true*. Thus, if invoked on c_0 it evaluates to *true* and if invoked on c_i ($i > 0$), it evaluates to *false* .