CS611 Lecture 21 **Expressiveness of Lambda Calculus** 17 October 2003

Scribe: Wojciech Moczydłowski, Jr Lecturer: Radu Rugina

Last time we have defined terms of lambda calculus, small-step and large-step CBN and CBV semantics. However, there are many things we still do not have in order to be able to call lambda calculus a programming language. We will show now how to express familiar high-level constructs in lambda calculus.

- Functions with multiple arguments. Consider an extended language:

$$e \quad ::= \quad x \;\mid\; e(e_1, e_2, \ldots, e_n) \;\mid\; \lambda x_1, x_2, \ldots, x_n.e$$

  We translate terms in the extended language to the lambda calculus as follows:

  - $\lambda x_1, x_2, \ldots, x_n.e \equiv \lambda x_1.\lambda x_2.\ldots\lambda x_n.e$
  - $e_0(e_1, e_2, \ldots, e_n) \equiv e_0\ e_1\ e_2\ \ldots\ e_n$

  This translation behaves exactly as we expect:[1]

$$\begin{aligned}(\lambda x_1.\lambda x_2.e)\ e_1\ e_2 \quad &\to_1 \quad (\lambda x_2.e)[x_1 \to e_1]\ e_2 \\ &\to_1 \quad e\ [x_1 \to e_1]\ [x_2 \to e_2]\end{aligned}$$

- Booleans. We encode booleans as follows:

$$\begin{aligned}\mathit{true} \quad &\equiv \quad \lambda x.\lambda y.x \\ \mathit{false} \quad &\equiv \quad \lambda x.\lambda y.y\end{aligned}$$

  These are called Church booleans, after 20th century mathematician Alonzo Church.

  In order to be able to show that they behave as expected, we provide also *test* expression, which corresponds to the *if-then-else* construct taking three arguments:

$$\mathit{test} \quad \equiv \quad \lambda b.\lambda x.\lambda y.\ b\ x\ y$$

  And this behaves exactly as expected:

$$\begin{aligned}\mathit{test}\ \mathit{true}\ e_0\ e_1 \quad &\equiv \quad (\lambda b.\lambda x.\lambda y.\ b\ x\ y)\ \mathit{true}\ e_0\ e_1 \\ &\to_1 \quad (\lambda x.\lambda y.\ \mathit{true}\ x\ y)\ e_0\ e_1 \\ &\to_1 \quad (\lambda y.\ \mathit{true}\ e_0\ y)\ e_1 \\ &\to_1 \quad \mathit{true}\ e_0\ e_1 \\ &\equiv \quad (\lambda x.\lambda y.\ x)\ e_0\ e_1 \\ &\to_1 \quad (\lambda y.\ e_0)\ e_1 \\ &\to_1 \quad e_0\end{aligned}$$

  It's easy to see that the fact *test false* $e_0\ e_1 \to_1^* e_1$ also holds. Note that we were using CBN reduction. If we used CBV reduction, then the difference might show if the reduction starting from e_0 or e_1 wouldn't terminate.

[1] The fact that it works, however, relies heavily on the assumption that we are dealing only with combinators.

Now we can define logical connectives.

$$\mathit{or} \quad \equiv \quad \lambda x.\lambda y.\ x\ x\ y$$

Let's see if it works.

$\mathit{or\ true\ false} \equiv (\lambda x.\ \lambda y.\ x\ x\ y)\ \mathit{true\ false} \rightarrow_1^2 \mathit{true\ true\ false} \equiv (\lambda x.\ \lambda y.\ x)\ \mathit{true\ false} \rightarrow_1 (\lambda y.\ \mathit{true})\ \mathit{false} \rightarrow_1 \mathit{true}$

We can also try:

$$\mathit{or\ false\ false} \rightarrow_1^3 \mathit{false\ false\ false} \rightarrow_1^2 \mathit{false}$$

This connective behaves exactly the same in CBN and CBV semantics, if it is applied to boolean arguments.

We leave the easy check that the expressions *or false true* and *or true true* reduce to *true* to the reader.

We can also define *or* in another way:

$$\mathit{or} \equiv \lambda x.\lambda y.\ x\ \mathit{true}\ y$$

- Numbers. Almost every sensible language (including Intercal!) includes natural numbers[2]. The definition of natural numbers in lambda calculus comes also from Church, and they are called Church numerals.

$$\begin{aligned} c_0 &\equiv \lambda s.\lambda z.\ z \\ c_1 &\equiv \lambda s.\lambda z.\ s\ z \\ c_2 &\equiv \lambda s.\lambda z.\ s\ (s\ z) \\ &\vdots \\ c_n &\equiv \lambda s.\lambda z.\ s^n(z) \end{aligned}$$

The term c_i represents the number i in lambda calculus. So we represent the number i by the application of a function s to z n times. So z can be thought of as a zero and s as a successor. Let's see how we can do arithmetic!

  - $\mathit{inc} \equiv \lambda n.\lambda s.\lambda z.\ s\ (n\ s\ z)$. *inc* corresponds to the successor operation on natural numbers. To convince ourselves that it works, let's reduce $\mathit{inc}\ c_2$:

$$\mathit{inc}\ c_2 \rightarrow_\beta \lambda s.\lambda z.\ s\ (c_2\ s\ z) \rightarrow_\beta^2 \lambda s.\lambda z.\ s\ (s\ (s\ z)) \equiv c_3$$

    Note that we had to use $\rightarrow_\beta$ reduction, because the other kinds of reduction stop at WHNF.[3]
  - Addition: $\mathit{add} \equiv \lambda m.\lambda n.\lambda s.\lambda z.\ m\ s\ (n\ s\ z)$.
  - The other way to define addition is: $\mathit{add} \equiv \lambda m.\lambda n.\ m\ \mathit{inc}\ n$.
  - Testing for 0. Let's define an expression which corresponds intuitively to $\lambda m.\ \mathit{test}\ (m = 0)$:

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

    We leave the check that it works to the reader.

[2]Exercise: Find one which doesn't.

[3]This means that we are unclear about what exactly we expect from the encoding. Usually we want it to behave correctly with respect to the transitive-reflexive closure of β-reduction. Note that $\rightarrow_1 \subset \rightarrow_\beta^*$.

- Pairs. In order to define pairs we need to define the pair constructor and projections:

$$\begin{array}{rcl} \mathit{pair} & \equiv & \lambda f.\lambda s.\lambda b.\ b\ f\ s \\ \mathit{fst} & \equiv & \lambda p.\ p\ \mathit{true} \\ \mathit{snd} & \equiv & \lambda p.\ p\ \mathit{false} \end{array}$$

  To see that they work, look at the reductions below.

$$\mathit{fst}\ (\mathit{pair}\ u\ v) \to_1 \mathit{pair}\ u\ v\ \mathit{true} \to_1^3 \mathit{true}\ u\ v \to_1^3 u$$

  Here we used CBN semantics. If u and v were in WHNF, then CBV would work exactly the same.

- Subtraction. This one is tricky:

$$\begin{array}{rcl} \mathit{zz} & \equiv & \mathit{pair}\ c_0\ c_0 \\ \mathit{ss} & \equiv & \lambda p.\mathit{pair}\ (\mathit{snd}\ p)\ (\mathit{inc}\ (\mathit{snd}\ p)) \\ \mathit{dec} & \equiv & \lambda m.\mathit{fst}\ (m\ \mathit{ss}\ \mathit{zz}) \\ \mathit{sub} & \equiv & \lambda m.\lambda n.\ n\ \mathit{dec}\ m \end{array}$$

  The term *sub* represents the function $\lambda m.\lambda n.m-n$. What's going on here? We use pairs of numbers to compute the substraction. *ss* is an operation which given the pair (c_{n_1}, c_{n_2}) returns a pair (c_{n_2}, c_{n_2+1}). So by applying *ss* m times to the pair (c_0, c_0) we will get a pair (c_{m-1}, c_m). $\mathit{dec}\ c_m$ hence equals (reduces) to c_{m-1} if $m > 0$ and c_0 in other case (remember that a number n is defined to be a application n times the given function to the given argument). Seeing this, the definition of *sub* becomes trivial — we simply decrease m n times.[4]

[4]Exercise — how many reduction steps does it take to compute $\mathit{dec}\ m\ n$?