

## COMP 617 RAP Seminar, Fall 2006

Presenter: Seth Fogarty

Scribe: Gregory Malecha

09/13/2006

This lecture focused on the encoding of values as functions in the un-typed  $\lambda$ -calculus.

**Notation:** The  $\llbracket \cdot \rrbracket$  is a way of quoting program text and means the “representation” or “encoding” of something. For example,  $\llbracket b \rrbracket$  should be read as “the representation of b”.

## 1 Encoding Booleans

The representation of booleans is based closely on the **if-then-else** construct in OCaml. Logically, booleans are represented as functions of two arguments, which return the first argument if the value is true, and the second if the value is false. Concretely, this leads to the following definitions.

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

Based on this representation, the **if-then-else** construction can be represented as follows:

$$\llbracket \mathbf{if\ b\ then\ x\ else\ y} \rrbracket \equiv \llbracket b \rrbracket \llbracket x \rrbracket \llbracket y \rrbracket$$

Note that this equivalence only exists when evaluating using lazy semantics.

Based on this definition for booleans, we can construct basic functions on booleans, for example the binary and function, which can be formulated in pseudo-code as

$$\text{AND}(m,n) = \text{if } m \text{ then } n \text{ else false}$$

Since returning the false in this line implies that  $m$  has the value of false, returning  $m$  is equivalent. Writing this in the  $\lambda$ -calculus yields the following:

$$\llbracket \text{AND} \rrbracket \equiv \lambda m. \lambda n. m\ n\ m$$

This can be checked through hand-evaluation of each case,  $\{\text{TT}, \text{TF}, \text{FT}, \text{FF}\}$ . For example, the evaluation for  $(\llbracket \text{AND} \rrbracket \llbracket \mathbf{T} \rrbracket \llbracket \mathbf{F} \rrbracket)$  is given by:

$$\begin{aligned}&\llbracket \text{AND} \rrbracket \llbracket \mathbf{T} \rrbracket \llbracket \mathbf{F} \rrbracket \\ &\equiv (\lambda m. \lambda n. m\ n\ m) \llbracket \mathbf{T} \rrbracket \llbracket \mathbf{F} \rrbracket \\ &\rightarrow_{\beta} (\lambda n. \llbracket \mathbf{T} \rrbracket\ n\ \llbracket \mathbf{T} \rrbracket) \llbracket \mathbf{F} \rrbracket \\ &\rightarrow_{\beta} \llbracket \mathbf{T} \rrbracket \llbracket \mathbf{F} \rrbracket \llbracket \mathbf{T} \rrbracket \\ &\equiv (\lambda x. \lambda y. x) \llbracket \mathbf{F} \rrbracket \llbracket \mathbf{T} \rrbracket \\ &\rightarrow_{\beta} (\lambda y. \llbracket \mathbf{F} \rrbracket) \llbracket \mathbf{T} \rrbracket \\ &\rightarrow_{\beta} \llbracket \mathbf{F} \rrbracket\end{aligned}$$

## 2 Encoding the Natural Numbers

When considering the encoding of natural numbers, we again consider the ways in which we use numbers. A simple use of numbers is iterative application. This leads to the representation of numbers as binary functions which apply the first argument to the second the number of times that the number represents. This representation was first developed by Church, and so we refer to it as Church numerals. For example, the formulations of 0, 1, 2, and 3 are given below.

$$\begin{aligned} \llbracket 0 \rrbracket &= \lambda f. \lambda a. a \\ \llbracket 1 \rrbracket &= \lambda f. \lambda a. f a \\ \llbracket 2 \rrbracket &= \lambda f. \lambda a. f (f a) \\ \llbracket 3 \rrbracket &= \lambda f. \lambda a. f (f (f a)) \end{aligned}$$

The most primitive function on natural numbers is the successor function which adds 1 to the given number. Based on the Church representation of numbers, we can formulate the successor function as:

$$\llbracket \text{succ} \rrbracket = \lambda n. \lambda f. \lambda a. f (n f a)$$

Once again, we can check this by analysis of an example.

$$\begin{aligned} &\llbracket \text{succ} \rrbracket \llbracket 1 \rrbracket \\ \equiv & (\lambda n. \lambda f. \lambda a. f (n f a)) \llbracket 1 \rrbracket \\ \rightarrow_{\beta} & \lambda f. \lambda a. f (\llbracket 1 \rrbracket f a) \end{aligned}$$

If we continued to evaluate this using full  $\beta$ -reduction rules, then we would obtain the exact program represented by  $\llbracket 2 \rrbracket$ ; however, it is sufficient that this function behaves in exactly the same way as  $\llbracket 2 \rrbracket$ , the text of the program is not significant.