CMPSC 461: Programming Language Concepts
Lecture Note 2: Encodings

1 Introduction

Even though all values in the λ-calculus are functions, it would be nice to somehow have objects which could be worked with like integers and boolean values, and control-flow constructs. In this note, we will build such constructs from the core λ-calculus.

2 Encoding Booleans

We first encode Boolean values ($\texttt{TRUE}$, $\texttt{FALSE}$) and functions on them (such as $\texttt{AND}$, $\texttt{IF}$), such that the expected behavior holds, including statements such as

- $\texttt{AND}\ \texttt{TRUE}\ \texttt{FALSE} = \texttt{FALSE}$
- $\texttt{IF}\ \texttt{TRUE}\ t\ f = t$, where t and f are arbitrary λ-terms

Notice that here we use the *prefix* form ($\texttt{AND}\ \texttt{TRUE}\ \texttt{FALSE}$), rather than the *infix* form ($\texttt{TRUE}\ \texttt{AND}\ \texttt{FALSE}$) to emphasize that the operation $\texttt{AND}$ is a function with two parameters. Moreover, if-statement in a functional programming language is a function with three parameters: a branch condition; values returned when the branch condition evaluates to $\texttt{TRUE}$ and $\texttt{FALSE}$ respectively.

One way to encode the Boolean values is:

- $\texttt{TRUE} \triangleq \lambda x\ y.\ x$
- $\texttt{FALSE} \triangleq \lambda x\ y.\ y$

To encode function $\texttt{IF}$, we notice that it should be of the form:

$$\texttt{IF} = \lambda b\ t\ f.\ \texttt{if}\ b = \texttt{TRUE}\ \texttt{then}\ t\ \texttt{else}\ f$$

Now, the definitions of Boolean values become handy, because $\texttt{TRUE}\ t\ f$ evaluates to t, and $\texttt{FALSE}\ t\ f$ evaluates to f. Hence, we only need the term $b\ t\ f$ in the body of $\texttt{IF}$:

$$\texttt{IF} \triangleq \lambda b\ t\ f.\ b\ t\ f$$

With the encoding of $\texttt{IF}$, we can easily define other Boolean operations. For example, we can encode the operation $\texttt{AND}$:

$$\texttt{AND} = \lambda b_1\ b_2.\ \texttt{IF}\ b_1\ (\texttt{IF}\ b_2\ \texttt{TRUE}\ \texttt{FALSE})\ \texttt{FALSE}$$

With β-reduction, that is equivalent to:

$$\texttt{AND} = \lambda b_1\ b_2.\ b_1\ (b_2\ \texttt{TRUE}\ \texttt{FALSE})\ \texttt{FALSE}$$

Now we check that such encodings work as expected by one example.

$$\begin{aligned}
\texttt{AND}\ \texttt{TRUE}\ \texttt{FALSE} &= (\lambda b_1\ b_2.\ b_1\ (b_2\ \texttt{TRUE}\ \texttt{FALSE})\ \texttt{FALSE})\ \texttt{TRUE}\ \texttt{FALSE} \quad \text{(by definition)} \\
&= \texttt{TRUE}\ (\texttt{FALSE}\ \texttt{TRUE}\ \texttt{FALSE})\ \texttt{FALSE} \quad (\beta\text{-reduction}) \\
&= (\lambda x\ y.\ x)\ (\texttt{FALSE}\ \texttt{TRUE}\ \texttt{FALSE})\ \texttt{FALSE} \quad \text{(by definition)} \\
&= \texttt{FALSE}\ \texttt{TRUE}\ \texttt{FALSE} \quad (\beta\text{-reduction}) \\
&= (\lambda x\ y.\ y)\ \texttt{TRUE}\ \texttt{FALSE} \quad \text{(by definition)} \\
&= \texttt{FALSE} \quad (\beta\text{-reduction})
\end{aligned}$$

3 Encoding integers

To encode numbers, we'll use Church numerals. Here, the number n is represented as a higher-order function $\underline{n}$ which, given another function, returns the n-fold composition of that other function: $\underline{n}(f) \mapsto f^n$. The function $\underline{n}$ (we use an underline to distinguish a Church numeral from its counterpart in mathematics) encodes number n. In other words,

$$\underline{n} \triangleq \lambda f\ z.\ f^n\ z = \lambda f\ z.\ \underbrace{f(f(\ldots f}_{n \text{ times}}(z)))$$

For example,

$$\begin{aligned} \underline{0} &\triangleq \lambda f\ z.\ z \\ \underline{1} &\triangleq \lambda f\ z.\ f\ z \\ \underline{2} &\triangleq \lambda f\ z.\ f\ (f\ z) \end{aligned}$$

For Church numerals, we can define a successor function `SUCC` which takes one Church number and returns its successor. To see how the encoding works, we observe that by definition, `SUCC` should take some Church number $\underline{n}$ as input, and return the encoding of $n+1$, which is in the form of $\lambda f\ z.\ t$ for some t yet to be defined. So the goal is write down some t such that $t = f^{n+1}\ z$ given input $\underline{n}$.

To define the missing piece t, we notice that for any $\underline{n}$, $\underline{n}\ f\ z = f^n\ z$. So given an input $\underline{n}$, we can apply f to $\underline{n}\ f\ z$ to get $f^{n+1}\ z$ (i.e., $f\ (\underline{n}\ f\ z) = f^{n+1}\ z$). Hence, we simply define t as $f\ (n\ f\ z)$ where n is the input. In summary, we have the following definition:

$$\texttt{SUCC} \triangleq \lambda n.\ \lambda f\ z.\ f\ (n\ f\ z)$$

We can also define simple arithmetic, such as `PLUS`. One way of doing that is:

$$\texttt{PLUS} \triangleq \lambda n_1\ n_2.\ (n_1\ \texttt{SUCC}\ n_2)$$

Why this encoding works? Intuitively, a Church number $\underline{n}$ is a function that takes another function f and a term t as inputs, and returns the result of applying f to t for n times. Therefore, we can compute compute $\underline{n_1 + n_2}$ via $\underline{n_1}$ `SUCC` $\underline{n_2}$ (i.e., increment $\underline{n_2}$ by one for n_1 times).

Following this idea, we can easily encode multiplication `MULT` as:

$$\texttt{MULT} \triangleq \lambda n_1\ n_2.\ (n_1\ (\texttt{PLUS}\ n_2)\ \underline{0})$$

Finally, we check that `PLUS` $\underline{1}\ \underline{2} = \underline{3}$ based on the definition above.

$$\begin{aligned} \texttt{PLUS}\ \underline{1}\ \underline{2} &= (\lambda n_1\ n_2.\ (n_1\ \texttt{SUCC}\ n_2))\ \underline{1}\ \underline{2} \quad \text{(by definition)} \\ &= \underline{1}\ \texttt{SUCC}\ \underline{2} \quad (\beta\text{-reduction}) \\ &= (\lambda f\ z.\ f\ z)\ \texttt{SUCC}\ \underline{2} \quad \text{(by definition)} \\ &= \texttt{SUCC}\ \underline{2} \quad (\beta\text{-reduction}) \\ &= (\lambda n.\ \lambda f\ z.\ f\ (n\ f\ z))\ \underline{2} \quad \text{(by definition)} \\ &= \lambda f\ z.\ f\ (\underline{2}\ f\ z) \quad (\beta\text{-reduction}) \\ &= \lambda f\ z.\ f\ \big((\lambda f\ z.\ f\ (f\ z))\ f\ z\big) \quad \text{(by definition)} \\ &= \lambda f\ z.\ f\ (f\ (f\ z)) \quad (\beta\text{-reduction}) \\ &= \underline{3} \quad \text{(by definition)} \end{aligned}$$