CS611 Lecture 4 — λ Encodings and Recursion — 6 September 2006

Lecturer: Dexter Kozen

1 Encoding Booleans, Natural Numbers, and Data Structures

Even though the pure λ-calculus consists only of λ-terms, we can represent and manipulate common data objects like integers, Boolean values, lists, and trees. All these things can be encoded as λ-terms.

1.1 Encoding Booleans

The Booleans are the easiest to encode, so let us start with them. We would like to define the Boolean constants *TRUE* and *FALSE* and the Boolean operators *IF*, *AND*, *OR*, *NOT*, etc. so that they behave in the expected way. There are many reasonable encodings. One good one is to define *TRUE* and *FALSE* by:

$$\begin{aligned} \mathit{TRUE} &\triangleq \lambda xy.\, x \\ \mathit{FALSE} &\triangleq \lambda xy.\, y. \end{aligned}$$

Now we would like to define the conditional test *IF*. We would like *IF* to take three arguments b, t, f, where b is a Boolean value and t, f are arbitrary λ-terms. The function should return t if $b = \mathit{TRUE}$ and f if $b = \mathit{FALSE}$.

$$\mathit{IF} \;=\; \lambda b\,t f.\begin{cases} t, & \text{if } b = \mathit{TRUE}, \\ f, & \text{if } b = \mathit{FALSE}. \end{cases}$$

Now the reason for defining *TRUE* and *FALSE* the way we did becomes clear. Since $\mathit{TRUE}\ t\ f \rightarrow t$ and $\mathit{FALSE}\, t\ f \rightarrow f$, all *IF* has to do is apply its Boolean argument to the other two arguments:

$$\mathit{IF} \triangleq \lambda b\,t f.\, b\ t\ f$$

The other Boolean operators can be defined from *IF*:

$$\begin{aligned} \mathit{AND} &\triangleq \lambda b_1\, b_2.\, \mathit{IF}\, b_1\ b_2\ \mathit{FALSE} \\ \mathit{OR} &\triangleq \lambda b_1\, b_2.\, \mathit{IF}\, b_1\ \mathit{TRUE}\, b_2 \\ \mathit{NOT} &\triangleq \lambda b_1.\, \mathit{IF}\, b_1\ \mathit{FALSE}\ \mathit{TRUE} \end{aligned}$$

Whereas these operators work correctly when given Boolean values as we have defined them, all bets are off if they are applied to any other λ-term. There is no guarantee of any kind of reasonable behavior. Basically, with the untyped λ-calculus, it is *garbage in, garbage out.*

1.2 Encoding Integers

We will encode natural numbers $\mathbb{N}$ using *Church numerals.* This is the same encoding that Alonzo Church used, although there are other reasonable encodings. The Church numeral for the number $n \in \mathbb{N}$ is denoted $\overline{n}$. It is the λ-term $\lambda f x.\, f^n\, x$, where f^n denotes the n-fold composition of f with itself:

$$\begin{aligned} \overline{0} &\triangleq \lambda f x.\, f^0\, x &&= \lambda f x.\, x \\ \overline{1} &\triangleq \lambda f x.\, f^1\, x &&= \lambda f x.\, f\ x \\ \overline{2} &\triangleq \lambda f x.\, f^2\, x &&= \lambda f x.\, f(f\ x) \\ \overline{3} &\triangleq \lambda f x.\, f^3\, x &&= \lambda f x.\, f(f(f\ x)) \\ &\ \vdots \\ \overline{n} &\triangleq \lambda f x.\, f^n\, x &&= \lambda f x.\, \underbrace{f(f(\ldots(f}_{n}\ x)\ldots) \end{aligned}$$

We can define the successor function s as

$$s \triangleq \lambda n f x.\, f(n\ f\ x).$$

That is, s on input $\overline{n}$ returns a function that takes a function f as input, applies $\overline{n}$ to it to get the n-fold composition of f with itself, then composes that with one more f to get the $(n+1)$-fold composition of f with itself. Then

$$\begin{array}{rcl} s\,\overline{n} & = & (\lambda n f x.\, f(n\ f\ x))\,\overline{n} \\ & \rightarrow & \lambda f x.\, f(\overline{n}\ f\ x) \\ & \rightarrow & \lambda f x.\, f(f^n\ x) \\ & = & \lambda f x.\, f^{n+1}\ x \\ & = & \overline{n+1}. \end{array}$$

We can perform basic arithmetic with Church numerals. For addition, we might define

$$\mathit{ADD} \triangleq \lambda m\, n\, f\, x.\, m\ f\ (n\ f\ x).$$

On input $\overline{m}$ and $\overline{n}$, this function returns

$$\begin{array}{rcl} (\lambda m\, n\, f\, x.\, m\ f\ (n\ f\ x))\ \overline{m}\,\overline{n} & \rightarrow & \lambda f\ x.\, \overline{m}\ f\ (\overline{n}\ f\ x) \\ & \rightarrow & \lambda f\ x.\, f^m\ (f^n\ x) \\ & = & \lambda f\ x.\, f^{m+n}\ x \\ & = & \overline{m+n}. \end{array}$$

Here we are composing f^m with f^n to get f^{m+n}.

Alternatively, recall that Church numerals act on a function to apply that function repeatedly, and addition can be viewed as repeated application of the successor function, so we could define

$$\mathit{ADD} \triangleq \lambda m\, n.\, m\ s\ n.$$

Similarly, multiplication is just iterated addition, and exponentiation is iterated multiplication:

$$\mathit{MUL} \triangleq \lambda m\, n.\, m\ (\mathit{ADD}\ n)\ \overline{0} \qquad \mathit{EXP} \triangleq \lambda m\, n.\, m\ (\mathit{MUL}\ n)\ \overline{1}.$$

1.3 Pairing and Projections

Logic and arithmetic are good places to start, but we still are lacking any useful data structures. For example, consider ordered pairs. It would be nice to have a pairing function *PAIR* with projections *FIRST* and *SECOND* that obeyed the following equational specifications:

$$\begin{array}{rcl} \mathit{FIRST}\ (\mathit{PAIR}\ e_1\ e_2) & = & e_1 \\ \mathit{SECOND}\ (\mathit{PAIR}\ e_1\ e_2) & = & e_2 \\ \mathit{PAIR}\ (\mathit{FIRST}\ p)\ (\mathit{SECOND}\ p) & = & p, \end{array}$$

provided p is a pair. We can take a hint from *IF*. Recall that *IF* selects one of its two branch options depending on its Boolean argument. *PAIR* can do something similar, wrapping its two arguments for later extraction by some function f:

$$\mathit{PAIR} \triangleq \lambda a b f.\, f\ a\ b.$$

Thus $\mathit{PAIR}\ e_1\ e_2 \rightarrow \lambda f.\, f\ e_1\ e_2$. To get e_1 back out, we can just apply this to *TRUE*: $(\lambda f.\, f\ e_1\ e_2)\ \mathit{TRUE} \rightarrow \mathit{TRUE}\ e_1\ e_2 \rightarrow e_1$, and similarly applying it to *FALSE* extracts e_2. Thus we can define

$$\mathit{FIRST} \triangleq \lambda p.\, p\ \mathit{TRUE} \qquad \mathit{SECOND} \triangleq \lambda p.\, p\ \mathit{FALSE}.$$

Again, if p isn't a term of the form $\mathit{PAIR}\ a\ b$, expect the unexpected.

2 Recursion and the Y Combinator

With an encoding for *IF*, we have some control over the flow of a program. We can also write simple `for` loops using the Church numerals $\overline{n}$. However, we do not yet have the ability to write an unbounded `while` loop or a recursive function.

In ML, we can write the factorial function recursively as

$$\mathsf{fun\ fact(n) = if\ n} \leq \mathsf{1\ then\ 1\ else\ n{*}fact(n\text{-}1)}$$

But how can we write this in the λ-calculus, where all the functions are anonymous? We must somehow construct a λ-term *FACT* that satisfies the equation

$$\mathit{FACT} \;=\; \lambda n.\, \mathit{IF}\ (n \leq 1)\ 1\ (\mathit{MUL}\ n\ (\mathit{FACT}\ (\mathit{SUB}\ n\ 1))) \tag{1}$$

Equivalently, we must construct a *fixpoint* of the map F defined by

$$F \;\stackrel{\triangle}{=}\; \lambda f.\, \lambda n.\, \mathit{IF}\ (n \leq 1)\ 1\ (\mathit{MUL}\ n\ (f\ (\mathit{SUB}\ n\ 1)));$$

that is, a λ-term *FACT* such that $F(\mathit{FACT}) = \mathit{FACT}$. Any solution of (1) will do; different solutions may disagree on non-integers, but one can show inductively that any solution of (1) will yield $\overline{n!}$ on input $\overline{n}$. Thus it is only a question of existence.

Now consider the λ-term

$$(\lambda x.\, F\ (x\ x))\ (\lambda x.\, F\ (x\ x)).$$

This is a fixpoint of F, since

$$(\lambda x.\, F\ (x\ x))\ (\lambda x.\, F\ (x\ x)) \;\rightarrow\; F\ ((\lambda x.\, F\ (x\ x))\ (\lambda x.\, F\ (x\ x))).$$

Moreover, this construction does not depend on the nature of F, so we can define

$$Y \;\stackrel{\triangle}{=}\; \lambda f.\, (\lambda x.\, f\ (x\ x))\ (\lambda x.\, f\ (x\ x)).$$

Then for any f, we have that $Y\,f$ is a fixpoint of f; that is, $Y\,f = f\,(Y\,f)$.

This Y is the infamous Y *combinator*, a closed λ-term that constructs solutions to recursive equations in a uniform way.

Curiously, although *every* λ-term is a fixpoint of the identity map $\lambda x.\, x$, the Y combinator produces a particularly unfortunate one, namely the divergent λ-term Ω introduced in Lecture 2.