CS704: Lecture 5
Encodings and Recursion in λ-calculus

Thomas Reps
[Based on notes taken by David Guild on January 29, 2010]

Abstract

This lecture presents the use of encodings to represent data in pure λ-calculus, including encoding schemes for Boolean values, natural numbers, and simple data structures (constructed from a pairing constructor). It also presents how recursive functions can be encoded in λ-calculus.

1 Review

Last time, we gave a brief introduction to encodings. Recall that an encoding of an abstract data type (consisting of a certain kind of data, plus one or more operations on that kind of data) involves a family of combinators to represent the data items, together with additional combinators to represent the operations. A combinator that encodes some operator O on encoded data values must, on encoded data values, mimic the action of operator O on (unencoded) data values. (The result of applying the combinator that encodes O to a λ-calculus term or terms that do not represent encoded data values is allowed to be anything—it is allowed to be equal to some encoded data value or not equal to any encoded data value.)

To distinguish between values and encoded values, we write the data value as V and the encoded value (a combinator) as $\underline{V}$. For instance, we defined the combinator $\underline{T} \stackrel{\text{def}}{=} \lambda x.\lambda y.x$, as the representation of the Boolean value T.

Encodings for the Booleans, and the if-then-else operator, are repeated here.

$$\begin{aligned} \underline{T} &= \lambda x.\lambda y.x \\ \underline{F} &= \lambda x.\lambda y.y \\ \underline{\text{if-then-else}} &= \lambda b.\lambda m.\lambda n.bmn \end{aligned}$$

2 Operations on Booleans

As mentioned above, the key requirement on a λ-calculus term that encodes an operator is that its application to a λ-calculus term or terms that represent valid data must yield a term that is equal to the encoding of the correct result. The effect of the operator on terms that do not represent encoded data values, including terms that represent other data types, is unspecified. Thus, we will make the assumption that each term supplied as an argument to an operator is a term that encodes the intended kind of data value.

Let's construct the $\underline{\text{not}}$ combinator. We already know what $\underline{T}$ and $\underline{F}$ are, and a little thinking shows us that $\lambda b.(\lambda x.\lambda y.byx)$ gives us the proper behavior. The parentheses are to emphasize that the combinator takes a Boolean b and constructs a brand new combinator, starting with $\lambda x.\lambda y$, as the return value; the Boolean value b is used to select either x or y as the body of the new combinator.

This method of encoding operators is relatively easy. The operator needs a λ for each input variable, plus the λs that appear in the encoded result. The trick is to use the *action* of the input data combinators to construct the body term of the combinator that is returned as the result. For example, one combinator that encodes the "or" operations is defined as follows:

$$\underline{\text{or}} \stackrel{\text{def}}{=} \lambda a.\lambda b.\lambda x.\lambda y.ax(bxy). \tag{1}$$

In this combinator, the body reduces to x unless both arguments equal $\underline{F}$, in which case the body reduces to y (and the return value is equal to $\underline{F}$).

Eqn. (1) is not the only way to encode "or". Notice that the terms $\lambda a.\lambda b.aab$ and $\lambda a.\lambda b.a\underline{T}b$ are also both $\underline{\text{or}}$ combinators as well! These examples illustrate that there are many combinators for any given operation. In fact, by performing β-expansions, one can generate an infinite number of combinators for any given operation.

3 Data Structures

The next step is to add a mechanism for creating structured data (lists, trees, etc.) The simplest such structure requires the ability to pair two elements (which can encode any kind of data, including other pairs). We will denote the pair consisting of M and N by "(M, N)". We define the pair that consists of arbitrary data elements M and N as follows:

$$\underline{(M, N)} \stackrel{\text{def}}{=} \lambda x.xMN.$$

Thus, the constructor $\underline{\text{pair}}$ can be defined as $\lambda m.\lambda n.(\lambda x.xmn)$.

Next, we need a way to extract data from a pair. Given our method for encoding pairs, the two destruction operations $\underline{\text{first}}$ and $\underline{\text{second}}$ are quite simple to define:

$$\underline{\text{first}} \stackrel{\text{def}}{=} \lambda p.p\underline{T}$$
$$\underline{\text{second}} \stackrel{\text{def}}{=} \lambda p.p\underline{F}.$$

Once we have $\underline{\text{pair}}$, $\underline{\text{first}}$, and $\underline{\text{second}}$, we can build more complex data structures. For instance, a LISP "cons-cell" is a pair of pointers; to encode lists of type t, the first pointer of a pair is to a value of type t, and the second is to the rest of the list. However, in LISP one can encode other kinds of structures, such as trees with nodes of arity k, by using lists of length k to encode a k-tuple of pointers. Such constructions works equally well in λ-calculus. We also need some recognizable list-terminator element and a way to test whether an element of a pair is the list-terminator element. (We could use $\underline{0}$ and $\underline{\text{isZero}}$, which are defined in §4.)

It can be shown that λ-calculus is Turing-complete. Although we do not yet have everything we would need to prove that fact, we can sketch how to encode the tape of a Turing machine. We use $\underline{\text{pair}}$ to represent the tape as follows: there would be an outer pair, consisting of the symbol under the tape head along with an inner pair (whose two elements are both linked lists). The two lists of the inner pair would contain the symbols to the left and right of the tape head, respectively. We can further construct combinators that act on this structure to simulate moving left or right on the tape, using $\underline{\text{if-then-else}}$, $\underline{\text{pair}}$, $\underline{\text{first}}$, and $\underline{\text{second}}$.

4 Church Numerals

There are several possible ways to encode numbers. For instance, we could encode them in unary by using a list of length k to encode k. We could also encode them in binary by using a list of length $O(\log k)$ to encode k. In these cases, the combinators for $\underline{\text{isZero}}$, $\underline{\text{plus}}$, $\underline{\text{times}}$, etc. would have to interpret such representations in the standard way.

Alonzo Church defined a different encoding for the natural numbers, which we present below. He defined the encoding of the first few numbers as follows:

$$\begin{aligned}
\underline{0} &\stackrel{\text{def}}{=} \lambda f.\lambda x.x \\
\underline{1} &\stackrel{\text{def}}{=} \lambda f.\lambda x.fx \\
\underline{2} &\stackrel{\text{def}}{=} \lambda f.\lambda x.f(fx) \\
\underline{3} &\stackrel{\text{def}}{=} \lambda f.\lambda x.f(f(fx)).
\end{aligned}$$

This pattern continues, so the numeral $\underline{n}$ has n copies of f in it. (Consequently, the Church numerals are also a kind of unary notation for encoding the natural numbers.) We will write this as $\underline{n} \stackrel{\text{def}}{=} \lambda f.\lambda x.f^n x$, where $f^n x$ is meta-notation that denotes f applied n times to x. That is, $f^n x$ denotes $\underbrace{f(f(\ldots f}_{n \text{ times}}(x)\ldots))$. (Note that when $f^n x$ is expanded, the applications associate to the right.)

We define the $\underline{\text{isZero}}$ combinator, which accepts a single Church numeral and returns (an encoding of) a Boolean value, as follows:

$$\underline{\text{isZero}} \stackrel{\text{def}}{=} \lambda n.n(\lambda x.\underline{F})\underline{T}.$$

It is clear that this reduces to $\underline{T}$ when given $\underline{0}$, but how does it reduce to $\underline{F}$ otherwise? Look at the outermost f in the body of a non-zero λ-term. It will be bound to $\lambda x.\underline{F}$, and thus will take the entire rest of the term and replace it with $\underline{F}$. This is an example of what might be called "stuttering," where the combinator performs the same thing repeatedly (after performing something special for the non-repeated case).

The successor function, also known as the "add-1" function, can be defined as follows:

$$\underline{\text{succ}} \stackrel{\text{def}}{=} \lambda n.\lambda f.\lambda x.nf(fx).$$

The predecessor function, or "subtract-1" function, is significantly more complicated (and again uses a form of stuttering). First, we define the predecessor of 0 to be 0, because we are encoding the natural numbers, and thus do not have -1. The $\underline{\text{pred}}$ combinator should have the form $\lambda n.\ldots$, where n is a Church numeral of the form $\lambda f.\lambda x.f^n x$. We would like to find a way to just remove one of the fs from n; however, λ-calculus does not allow us to "edit" λ-calculus terms. Instead, we have to use n to build the appropriate answer from the ground up by iterating some process for n rounds. We will use pairs to do so: at each stage we will have a pair that consists of a function and a value. After the i^{th} round, the second component will contain $\underline{\text{pred}}\ i$, and thus after the n^{th} round, the second component has the desired answer. The initial pair—for the 0^{th} round—therefore must have the form $(?, \underline{0})$ (where "?" just means that we have not yet figured out the required form for the first component).

On each round, we will create the new second component by applying the previous first component to the previous second component. After the 1^{st} round, we again want the second component to hold $\underline{0}$. Thus, the the initial pair's first component could hold $\lambda x.\underline{0}$ (or $\lambda x.x$.).

Thereafter, we want the first component to add 1 to the second component; hence, from the 1^{st} round on, the first component should hold $\underline{\text{succ}}$.

In other words, using "$\lambda(f, v).\ldots$" as meta-notation that provides access to the individual components of an input pair (f, v), our combinator should perform

$$\lambda n.\underline{\text{second}}((\lambda(f, v).(\underline{\text{succ}}, (f\ v)))^n(\lambda x.\underline{0}, \underline{0})),$$

which can also be expressed as

$$\lambda n.\underline{\text{second}}(n\ (\lambda(f,v).(\underline{\text{succ}},(f\ v)))\ (\lambda x.\underline{0},\underline{0})).$$

Finally, we can express this in standard λ-calculus notation by rewriting $\lambda(f,v)\ldots.\,f\ldots v\ldots$ as

$$\lambda p.\ldots(\underline{\text{first}}\ p)\ldots(\underline{\text{second}}\ p)\ldots$$

to obtain

$$\underline{\text{pred}} \stackrel{\text{def}}{=} \lambda n.\underline{\text{second}}(n\ (\lambda p.\underline{\text{pair}}\ \underline{\text{succ}}\ ((\underline{\text{first}}\ p)(\underline{\text{second}}\ p)))\ (\underline{\text{pair}}\ (\lambda x.\underline{0})\ \underline{0})).$$

5 Recursion in λ-calculus

Recall the prototypical recursive function fact, which computes the value of x factorial. In C or Java, we can write fact as follows:

```
int fact(int x) {
   return(x == 0 ?  1 :  x * fact(x-1));
}
```

Our goal is to be able to write a functional expression in λ-calculus that is the λ-calculus analog of `fact`.

In λ-calculus, the problem that we have to finesse is that functional expressions do not have permanent names. Recursion in C relies on functions having permanent names; for instance, in the body of `fact`, the name `fact` is used to specify what function to call at the recursive call-site. We now show how a special combinator will let us side-step the problem of λ-calculus expressions not having permanent names. (As we will see, the combinator's bound variables essentially "issue" temporary names that suffice.)

The first step is to consider the following λ-calculus *equality*, in which "fact" appears on both the left-hand side and right-hand side:

$$\text{fact}(x) = \textbf{if}\ (\text{isZero}\ x)\ \textbf{then}\ 1\ \textbf{else}\ x * \text{fact}(x-1)\ \textbf{fi} \tag{2}$$

Such an equality can be considered an *implicit definition* of fact: to solve Eqn. (2) we must find a λ-calculus term such that, for all x, the left-hand-side λ-calculus term equals the right-hand-side λ-calculus term. In other words, Eqn. (2) is an equation that we are trying to solve for fact.

The next step is to rewrite Eqn. (3) as follows:

$$\text{fact} = \lambda x.\textbf{if}\ (\text{isZero}\ x)\ \textbf{then}\ 1\ \textbf{else}\ x * \text{fact}(x-1)\ \textbf{fi} \tag{3}$$

We now present a method that can be used solve *any recursive equation in λ-calculus*—i.e., any λ-calculus equation of the form

$$f = \lambda x.\ \underbrace{\ldots f\ldots f\ldots f\ldots f\ldots}_{\text{any number of occurrences of } f} \tag{4}$$

Theorem 5.1 *Every λ-calculus equation of the form shown in Eqn. (4) has a solution.* □

The proof of Thm. 5.1 proceeds by performing on Eqn. (4) exactly the steps that we use below to solve Eqn. (3).

To solve Eqn. (3), we introduce another combinator, which we will call Fact, that acts as a "check-your-work combinator". In particular, we define Fact by abstracting on the name "fact" in the right-hand side of Eqn. (3):

$$\text{Fact} = \lambda g.\lambda x.\textbf{if } (\text{isZero } x) \textbf{ then } 1 \textbf{ else } x * g(x-1) \textbf{ fi}$$

Notice that we have the property

$$\text{fact} = \text{Fact fact}.$$

This gives us another definition of fact: we seek a λ-term f such that $f = \text{Fact } f$. In other words, fact is a λ-calculus term that is a *fixed point* of Fact.

The next step is to hypothesize that there is a magic combinator, Y', that when supplied with Fact immediately creates the desired fixed-point solution. For this to be the case, we must have

$$Y' \text{ Fact} = \text{Fact}(Y' \text{ Fact}). \tag{5}$$

Eqn. (5) is a λ-calculus *equality*, and consequently in general we might have to use both β-expansions as well as β-reductions to obtain the right-hand side from the left-hand side. However, one way for Eqn. (5) to hold would be for the right-hand side to be obtainable from the left-hand side by a sequence of β-reductions only:

$$Y' \text{ Fact} \longrightarrow_\beta^* \text{Fact}(Y' \text{ Fact}). \tag{6}$$

For Eqn. (6) to hold, the sequence of β-reductions must recreate an occurrence of Y' on the right-hand side. However, we have seen something very much like this already with the λ-calculus term $(\lambda x.xx)(\lambda x.xx)$: in one β-reduction, it recreates itself. The way this happens is that the λx in the first half of $(\lambda x.xx)(\lambda x.xx)$ is bound to the second-half term (which is identical to the first-half term). Consequently, the λx in the first half is bound to "half" of $(\lambda x.xx)(\lambda x.xx)$, and thus the application xx causes $(\lambda x.xx)(\lambda x.xx)$ to be recreated.

Thus, we will make the guess that Y' consists of two identical terms $\theta\theta$, where the first θ term uses the second to recreate the whole. In particular, we have

$$\theta\theta\text{Fact} \longrightarrow_\beta^* \text{Fact}(\theta\theta\text{Fact}). \tag{7}$$

Eqn. (7) can be solved by choosing

$$\theta \stackrel{\text{def}}{=} (\lambda t.\lambda f.f(ttf)).$$

Thus,

$$Y' \stackrel{\text{def}}{=} (\lambda t.\lambda f.f(ttf))(\lambda t.\lambda f.f(ttf)),$$

and the solution to Eqn. (3) is the term fact $\stackrel{\text{def}}{=} Y'$ Fact; i.e.,

$$\text{fact} \stackrel{\text{def}}{=} (\lambda t.\lambda f.f(ttf))(\lambda t.\lambda f.f(ttf))(\lambda g.\lambda x.\textbf{if } (\text{isZero } x) \textbf{ then } 1 \textbf{ else } x * g(x-1) \textbf{ fi}).$$

The term Y' is Turing's fixed-point combinator.

Remark. Earlier we said that the way we would finesse the problem of functional expressions not having permanent names is for the fixed-point combinator's bound variables to "issue" temporary names. Unlike C and Java, where the "full" name `fact` is available, Y'—like $(\lambda x.xx)(\lambda x.xx)$—only has a name for *half* of Y': namely, the λt in the first occurrence of $(\lambda t.\lambda f.f(ttf))$, which gets bound to the second occurrence of $(\lambda t.\lambda f.f(ttf))$; however, the full Y' term can be recreated by an occurrence of self-application (i.e., tt). □

Remark. It is a consequence of Thm. 5.1 that the equation $x = x + 1$ has a solution in λ-calculus. However, we know that there is no solution to $x = x + 1$ in ordinary arithmetic. What is going on?

Answer: the λ-calculus term that solves $x = x + 1$ is not a Church numeral; hence there is no contradiction with what we know from ordinary arithmetic. In particular, the solution to $x = x + 1$ is the λ-calculus term "Y' $\lambda y.y + 1$". □