Notes on Foundations of Programming Languages Computability

Sandra Alves

October 11, 2022

Abstract

The λ-calculus is Turing complete, which means that every computable function can be written as terms in the λ-calculus. In this chapter we define encodings of some useful data structures (such as boolean values, pairs, natural numbers, etc), which are useful in relating the λ-calculus with other theories for recursive functions.

1 Booleans and conditionals

The boolean values are represented as functions of two values that evaluate to one or the other of their arguments. The values true and false are defined in the λ-calculus as:

$$\begin{array}{rcl} \mathsf{true} & \equiv & \lambda xy.x \\ \mathsf{false} & \equiv & \lambda xy.y \end{array}$$

Note that

$$\begin{array}{rcl} \mathsf{true}\ M\ N & \equiv & (\lambda xy.x)MN \longrightarrow^* M \\ \mathsf{false}\ M\ N & \equiv & (\lambda xy.y)MN \longrightarrow^* N \end{array}$$

Therefore, an appropriate encoding of if is such that if L M N $\longrightarrow^*$ LMN. Thus if can be encoded as:

$$\mathsf{if} \equiv \lambda pxy.pxy$$

Having defined the truth values and a conditional, other operations on Booleans can be easily encoded:

$$\begin{array}{rcl} \mathsf{and} & \equiv & \lambda pq.\mathsf{if}\ p\ q\ \mathsf{false} \\ \mathsf{or} & \equiv & \lambda pq.\mathsf{if}\ p\ \mathsf{true}\ q \\ \mathsf{not} & \equiv & \lambda pq.\mathsf{if}\ p\ \mathsf{false}\ \mathsf{true} \end{array}$$

There are other possible encodings. For example and can be encoded in a more direct way as $\lambda mn.mnm$. Boolean values are useful to define other data structures, such as pairs. Pairs and the usual projections fst and snd can be encoded as:

$$\begin{array}{rcl} \mathsf{pair} & \equiv & \lambda xyz.zxy \\ \mathsf{fst} & \equiv & \lambda p.p\ \mathsf{true} \\ \mathsf{snd} & \equiv & \lambda p.p\ \mathsf{false} \end{array}$$

Applying snd to M and N will reduce to λf.fMN (this is called packing). When λf.fMN is applied to a two-argument function λxy.L will reduce do L[M/x][N/y] (this is called unpacking).

2 Natural numbers and arithmetic

The first encoding of natural numbers in the λ-calculus was defined by Alonzo Church and it is known as the Church numerals. Church numerals $\underline{0}, \underline{1}, \underline{2}, \ldots$, are defined as follows:

$$\begin{array}{rcl} \underline{0} & \equiv & \lambda fx.x \\ \underline{1} & \equiv & \lambda fx.fx \\ \underline{2} & \equiv & \lambda fx.f(fx) \\ & & \ldots \\ \underline{n} & \equiv & \lambda fx.f^n x \\ & & \ldots \end{array}$$

That is, the natural number n is represented by the Church numeral $\underline{n}$, which has the property that for any λ-terms F and X,

$$\underline{n}FX =_\beta F^n X$$

Arithmetic functions that work on Church numerals can also be encoded as λ-terms:

$$\begin{array}{rcl} \mathsf{add} & \equiv & \lambda mnfx.mf(nfx) \\ \mathsf{mult} & \equiv & \lambda mnfx.m(nf)x \\ \mathsf{exp} & \equiv & \lambda mnfx.nmfx \end{array}$$

It is easy to show that the functions defined above behave as expected. We show the case of add.

$$\begin{array}{rcl} \mathsf{add}\ \underline{m}\ \underline{n} & \longrightarrow^* & \lambda fx.\underline{m}f(\underline{n}fx) \\ & \longrightarrow^* & \lambda fx.f^m(f^n x) \\ & \longrightarrow & \lambda fx.f^{m+n}x \equiv \underline{m+n} \end{array}$$

Other basic operations on numerals:

$$\begin{array}{rcl} \mathsf{succ} & \equiv & \lambda nfx.f(nfx) \\ \mathsf{iszero} & \equiv & \lambda n.n(\lambda x.\mathsf{false})\mathsf{true} \end{array}$$

Encoding the predecessor function is not so straightforward. Given f, we will consider a function g working on pairs of numerals, such that $g(x,y) = (f(x), x)$, thus

$$g^{n+1}(x,x) = (f^{n+1}(x), f^n(x))$$

We can encode the predecessor function as:

$$\mathsf{pred} \equiv \lambda nfx.\mathsf{snd}\ (n\ (\mathsf{prefn} f)\ (\mathsf{pair}\ x\ x))$$

with

$$\mathsf{prefn} \equiv \lambda fp.\mathsf{pair}\ (f(\mathsf{fst}\ p))\ (\mathsf{fst}\ p)$$

Subtraction can then be defined as:

$$\mathsf{sub} \equiv \lambda mn.n\ \mathsf{pred}\ m.$$

The encodings defined above are sometimes not very intuitive or even efficient. This is somewhat related to the fact that encodings carry their on control structure.

2.1 Lists

Similar to what happens with church numerals, a list $[x_1, x_2, \ldots, x_n]$ can be represented by the term

$$\lambda fy.f\ x_1\ (f\ x_2 \ldots (f\ x_n\ y) \ldots)$$

Alternatively, lists can be represented using pairs (as done in ML or Lisp). It is possible to represent a list $[x_1, x_2, \ldots, x_n]$ as $(x_1, (x_2, (\ldots (x_n, \mathsf{nil}) \ldots))$. List can be encoded as the following λ-terms:

$$\begin{array}{rcl} \mathsf{nil} & \equiv & \mathsf{pair\ true\ true} \\ \mathsf{cons} & \equiv & \lambda xy.\mathsf{pair\ false\ (pair}\ x\ y) \end{array}$$

The encoding of lists contains a boolean flag, which indicates whether or not the list is empty. The usual functions on lists, are thus defined as:

$$\begin{array}{rcl} \mathsf{null} & \equiv & \mathsf{fst} \\ \mathsf{hd} & \equiv & \lambda z.\mathsf{fst}\ (\mathsf{snd}\ z) \\ \mathsf{tl} & \equiv & \lambda z.\mathsf{snd}\ (\mathsf{snd}\ z) \end{array}$$

A simpler encoding of the empty list is $\lambda z.z$.

3 Recursion in the λ-calculus

We now look at a key aspect in computation, which is recursion. Church numerals allows us to encode a very large class of functions on natural numbers (due to the fact that a Church number can be used to define bounded iteration). The use of high-order gives Church numerals the ability to encode even functions out of the scope of primitive recursion (for example, Ackermann's function can be encoded as $\lambda m.m(\lambda fn.nf(f\underline{1}))$succ.

In recursion theory, general recursive functions are encoded through the use of minimisation. In the λ-calculus we will define general recursion, with the use of a *fixed point combinator*, which is a term $\mathbf{Y}$ such that $\mathbf{Y}F = F(\mathbf{Y}F)$, for all terms F.

Definition 3.1 *A* fixed point *of the function* F *is any* X *such that* $F(X) = X$. *In the case above,* $X = \mathbf{Y}F$.

To encode recursion through the use of a fixed point combinator, we consider F to be the body of the recursive function and use the law $\mathbf{Y}F = F(\mathbf{Y}F)$ to unfold F as many times as needed.

Example 3.2 *Consider the following recursive definitions*

$$\begin{array}{lcl} \mathsf{fact}\ n & \equiv & \mathsf{if}\ (\mathsf{iszero}\ n)\ \underline{1}\ (\mathsf{mult}\ n\ (\mathsf{fact}(\mathsf{pre}\ n))) \\ \mathsf{append}\ x\ y & \equiv & \mathsf{if}\ (\mathsf{null}\ x)\ y\ (\mathsf{cons}\ (\mathsf{hd}\ x)\ (\mathsf{append}\ (\mathsf{tl}\ x)\ y)) \\ \mathsf{zeroes} & \equiv & \mathsf{cons}\ \ \underline{0}\ \mathsf{zeroes} \end{array}$$

Then its recursive definitions in the λ*-calculus are:*

$$\begin{array}{lcl} \mathsf{fact} & \equiv & \mathbf{Y}(\lambda fn.\mathsf{if}\ (\mathsf{iszero}\ n)\ \underline{1}\ (\mathsf{mult}\ n\ (f(\mathsf{pre}\ n)))) \\ \mathsf{append} & \equiv & \mathbf{Y}(\lambda fxy.\mathsf{if}\ (\mathsf{null}\ x)\ y\ (\mathsf{cons}\ (\mathsf{hd}\ x)\ (f\ (\mathsf{tl}\ x)\ y))) \\ \mathsf{zeroes} & \equiv & \mathbf{Y}(\lambda f.\mathsf{cons}\ \ \underline{0}\ f) \end{array}$$

It is easy to verify that.

$$\begin{array}{rcl} \mathsf{zeroes} & \equiv & \mathbf{Y}(\lambda f.\mathsf{cons}\ \ \underline{0}\ f) \\ & = & (\lambda f.\mathsf{cons}\ \ \underline{0}\ f)\mathbf{Y}(\lambda f.\mathsf{cons}\ \ \underline{0}\ f) \\ & = & (\lambda f.\mathsf{cons}\ \ \underline{0}\ f)\mathsf{zeroes} \\ & \longrightarrow & \mathsf{cons}\ \ \underline{0}\ \mathsf{zeroes} \end{array}$$

We now formalize the general usage of $\mathbf{Y}$ in the definition of recursive functions. Any recursive equation, representing an n-argument equation, $Mx_1 \ldots x_n = PM$, where P is any λ-term is defined by the λ-term

$$M \equiv \mathbf{Y}(\lambda g x_1 \ldots x_n.Pg).$$

It is easy to see that

$$\begin{array}{rcl} Mx_1 \ldots x_n & \equiv & \mathbf{Y}(\lambda g x_1 \ldots x_n.Pg)x_1 \ldots x_n \\ & = & (\lambda g x_1 \ldots x_n.Pg)Mx_1 \ldots x_n \\ & \longrightarrow^* & PM \end{array}$$

For mutually recursive definitions M and N such that

$$\begin{array}{rcl} M & = & P\ M\ N \\ N & = & Q\ M\ N \end{array}$$

We consider the fixed point of a function F on pairs such that $F(X, Y) = (P\,X\,Y, Q\,X\,Y)$. Now using the encoding of pairs, we define

$$\begin{array}{lcl} L & \equiv & \mathbf{Y}(\lambda z.\mathsf{pair}(P\ (\mathsf{fst}\ z)\ (\mathsf{snd}\ z)))(Q\ (\mathsf{fst}\ z)\ (\mathsf{snd}\ z))) \\ \\ M & \equiv & \mathsf{fst}\ L \\ N & \equiv & \mathsf{snd}\ L \end{array}$$

3.1 Some well-known fixed-point combinators

We present some fixed-point combinators which are well known in the literature. The first one is Haskell Curry's $\mathbf{Y}$ combinator, which is defined by:

$$\mathbf{Y} = \lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

We can verify easily verify the fixed point property:

$$\begin{array}{rcl} \mathbf{Y}F & \longrightarrow & (\lambda x.F(xx))(\lambda x.F(xx)) \\ & \longrightarrow & F(\lambda x.F(xx))(\lambda x.F(xx)) \\ & = & F(\mathbf{Y}F) \end{array}$$

This combinator is also known as Curry's Paradoxical combinator. If we consider Russell's Paradox where one defines the set $R \equiv \{x \mid x \notin x\}$, then $R \in R$ if and only if $R \notin R$. Now representing in the λ-calculus sets as predicates, then $M \in N$ is encoded as $N(M)$, and $\{x \mid P\}$ is encoded as $\lambda x.P$. Now one can derive Russel's Paradox because $R \equiv \lambda x.\mathbf{not}(xx)$, which implies $RR = \mathbf{not}(RR)$, which is logically a contradiction. Curry's fixed-point is defined by replacing in $RR = \mathbf{not}(RR)$, **not** by any term F.

Note that no reduction $\mathbf{Y}F \longrightarrow^* F(\mathbf{Y}F)$ is possible. The property is verified by two β-reductions, followed by a β-expansion. A stronger requirement is to have a combinator M such that $\forall F.MF \longrightarrow^* M(MF)$. A combinator that satisfies this property is Alan Turing's Θ, defined as

$$\Theta \equiv AA \quad \text{where} \quad A \equiv \lambda xy.y(xxy)$$

Now it is easy to verify that:

$$\begin{array}{rcl} \Theta F & \equiv & (\lambda xy.y(xxy))AF \\ & \longrightarrow & (\lambda y.y(AAy))F \\ & \longrightarrow & F(AAF) \\ & \equiv & F(\Theta F) \end{array}$$

Another fixed-point combinator that verifies this stronger requirement is Klop's \$ combinator:

$$\begin{array}{rcl} \$ & = & \pounds \\ \pounds & = & \lambda abcdefghijklmnopqstuvwxyzr.r(thisisafixedpointcombinator) \end{array}$$

3.2 Head normal form

We now introduce a new notion of normal form, which is related to the notion of *result* in functional programming. Note that, if we consider a recursive definition $M = PM$ then, unless P is a constant function or the identity, M does not have a normal form. In the same way, anything defined through the use of a fixed-point combinator, is not likely to have a normal form, although it can still be used to compute with. We formalize this, with the definition of *head normal form*.

Definition 3.3 *A λ-term M is a head normal form (hnf) if and only if is of the form*

$$\lambda x_1 \ldots x_n.yM_1 \ldots M_m \qquad\qquad m, n \geq 0$$

The variable y is called the head variable.

Example 3.4 *Some examples of hnfs are $\lambda x.y\Omega$, xM, $\lambda z.z(\lambda x.x)$, but not $\lambda y.(\lambda x.a)y$. In fact the redex $(\lambda x.a)y$ in the previous term is called a head-redex.*

It is obvious that a normal form is also a head normal form. Also, if

$$\lambda x_1 \dots x_n.yM_1 \dots M_m \longrightarrow^* N$$

then the term N must be of the form:

$$\lambda x_1 \dots x_n.yN_1 \dots N_m$$

with $M_i \longrightarrow^* N_i$. This in a sense, means that head normal forms fix the outer structure of the "result". Also, the reductions $M_i \longrightarrow^* N_i$ do not interfere with each other, therefore they can be done in parallel.

The notion of head normal form is related to the notion of definability.

Definition 3.5 *A term is* defined *if and only if it can be reduced to a head normal form, otherwise is* undefined.

Example 3.6 *The term Ω is an example of an undefined term, whereas $x\Omega$ is defined.*

This notion is related to Solvability in the λ-calculus:

Definition 3.7 *A term M is* solvable *if and only if there exist $x_1, \dots, x_m$, $N_1, \dots, N_n$, such that $(\lambda x_1 \dots x_m.M)N_1 \dots N_n = I$*

Example 3.8 *Take the defined term $x\Omega$. Then $(\lambda x.x\Omega)(\lambda xy.y)I \longrightarrow^* I$.*

Exercises

1 Write other (more direct) encodings of or, not and xor.

2 Write other encodings of add, mult and exp.

3 Verify the correctness of succ and iszero:

$$\begin{array}{lcl} \mathsf{succ}\ \underline{n} & \longrightarrow^* & \underline{n+1} \\ \mathsf{iszero}\ \underline{0} & \longrightarrow^* & \mathsf{true} \\ \mathsf{iszero}\ \underline{n+1} & \longrightarrow^* & \mathsf{false} \end{array}$$

4 Recall the encoding of pre. Show that:

$$\begin{array}{rcl} \mathsf{pre}\ \underline{0} & \longrightarrow^* & \underline{0} \\ \mathsf{pre}\ (\underline{n+1}) & \longrightarrow^* & \underline{n} \end{array}$$

5 Recall the following encoding of the Ackermann function:

$$\mathsf{ack} = \lambda m.m(\lambda fn.nf(f\underline{1}))\ \mathsf{suc}$$

Show that it is possible to derive the following:

$$\begin{array}{rcl} \mathsf{ack}\ \underline{0}\ \underline{n} & = & \underline{n+1} \\ \mathsf{ack}\ (\underline{m+1})\ \underline{0} & = & \mathsf{ack}\ \underline{m}\ \underline{1} \\ \mathsf{ack}\ (\underline{m+1})\ (\underline{n+1}) & = & \mathsf{ack}\ \underline{m}\ (\mathsf{ack}\ (\underline{m+1})\ n) \end{array}$$

6 Consider the following alternative encoding for numerals:

$$\begin{array}{rcl} \lceil 0 \rceil & = & \lambda x.x \\ \lceil n+1 \rceil & = & \mathsf{pair\ false}\ \lceil n \rceil \end{array}$$

For example, $\lceil 3 \rceil = [\mathsf{false}, [\mathsf{false}, [\mathsf{false}, \lambda x.x]]]$, where $[M, N]$ represents pair M N. Consider also the following definitions for the succ, pred and iszero functions, using the representation for numbers above.

$$\begin{array}{rcl} \mathsf{succ} & = & \lambda x.[\mathsf{false}, x] \\ \mathsf{pred} & = & \lambda x.x\mathsf{false} \\ \mathsf{iszero} & = & \lambda x.x\mathsf{true} \end{array}$$

(a) Prove the correctness of the previous encodings.

(b) Using the functions above and the encoding for recursion, define alternative functions for addition and multiplication.

7 Let us now consider an alternative encoding of natural numbers by Dana Scott. Consider constructors zero Z and successor Z:

$$\begin{array}{rcl} \mathsf{Z} & \equiv & \lambda fx.x \\ \mathsf{S} & \equiv & \lambda fxz.xf \end{array}$$

The natural numbers are defined as:

$$\begin{array}{rcl} \mathbf{0} & \equiv & \mathsf{Z} \\ \mathbf{1} & \equiv & \mathsf{S\ Z} \\ \mathbf{2} & \equiv & \mathsf{S(\ S\ Z)} \\ \mathbf{3} & \equiv & \mathsf{S(\ S(\ S\ Z))} \end{array}$$

(a) Using the definitions, verify that $\mathbf{2} \twoheadrightarrow \lambda xz.x(\lambda xz.x(\lambda xz.z))$;

(b) Considering Scott's encoding of predecessor $\mathsf{pred} = \lambda x.x(\lambda y.y)\mathbf{0}$, verify that $(\mathsf{pred}\ \mathbf{2}) \twoheadrightarrow \mathbf{1}$.

(c) Prove the correcteness of the encoding, by showing the equations in exercise 6.

(d) Write a similar encoding for iszero.

8 Consider the following encoding of binary trees, as well as encodings for functions isLeaf, treeLeft and treeRight:

$$\begin{array}{rcl} \mathsf{leaf}(n) & \equiv & \lambda xy.xn \\ \mathsf{node}(l, r) & \equiv & \lambda xy.ylr \end{array}$$

$$\begin{array}{rcl} \mathsf{isLeaf} & \equiv & \lambda t.t\ (\lambda z.\mathsf{true})\ (\lambda zw.\mathsf{false}) \\ \mathsf{treeLeft} & \equiv & \lambda t.t\ (\lambda x.x)\ \mathsf{true} \\ \mathsf{treeRight} & \equiv & \lambda t.t\ (\lambda x.x)\ \mathsf{false} \end{array}$$

(a) Using the encodings above and the encodings for arithmetics, define in the λ-calculus an encoding for the function leafs, which calculates the number of leafs of a tree.

(b) Without using function isLeaf, treeLeft and treeRight, define a function nodes that calculates the number of internal nodes of a tree.

9 Consider the following Haskell mutually recursive definitions for functions odd and even.

```
odd n = if n == 0 then False else even (n-1)
even n = if n == 0 then True else odd (n-1)
```

(a) Using the technique for encoding mutual recursive definitions in the λ-calculus, define encodings for functions odd and even.

(b) Prove the correctness of the encodings.

10 A term is called *defined* if and only if it can be reduced to *head normal form*. Which of the following terms are defined? (Consider $K = \lambda xy.x$.)

$$Y \quad Ynot \quad K \quad YI \quad x\Omega \quad YK \quad Y(Kx) \quad \underline{n}$$

11 A term is said to be *solvable* if and only if there exists variables $x_1, \ldots, x_m$ and terms $N_1, \ldots, N_n$ such that

$$(\lambda x_1 \ldots x_m.M)N_1 \ldots N_n = I$$

Which of the terms from the previous exercise are solvable?