Foundations of Programming Languages 2023

Encodings in the λ-calculus

Sandra Alves

October 2023

Church-Turing-Kleene thesis

The notion of computability has been independently formalized in several ways:

- Gödel and Herbrand formally define general recursive functions;
- Turing formalized computations as Turing machines;
- Church formalized computation as functions using the λ-calculus.

Church, Turing and Kleene proved that all these notions coincide and they define the class of all computable functions

"A function is λ-computable if and only if it is Turing computable, and if and only if it is general recursive."

The computational power of λ-calculus

Being a Turing-complete model, it is possible to encode any computable function as a λ-term.

We start by showing how to encode different data types:

- booleans
- numbers
- pairs
- lists
- recursive functions

Booleans and conditionals

The values true and false are defined in the λ-calculus as:

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

Note that

$$\begin{array}{rcl} \text{true } M \ N & = & (\lambda xy.x)MN \twoheadrightarrow M \\ \text{false } M \ N & = & (\lambda xy.y)MN \twoheadrightarrow N \end{array}$$

An appropriate encoding of if is such that if $L \ M \ N \twoheadrightarrow LMN$.
Thus if can be encoded as:

$$\text{if} = \lambda pxy.pxy$$

Other boolean operations

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

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

There are other possible encodings. For example and can be encoded in a more direct way as $\lambda mn.mnm$. (Verify!)

$$(\lambda mn.mnm) \text{ true } N \twoheadrightarrow \text{true } N \text{ true} \twoheadrightarrow N$$
$$(\lambda mn.mnm) \text{ false } N \twoheadrightarrow \text{false } N \text{ false} \twoheadrightarrow \text{false}$$

Pairs and projections

Boolean values are useful to define other data structures, such as pairs.

$$\text{pair} \quad = \quad \lambda xyz.zxy$$

Packing:

$$\text{pair } M\ N \twoheadrightarrow \lambda f.fMN$$

Unpacking:

$$(\lambda f.fMN)(\lambda xy.L) \twoheadrightarrow L[M/x][N/y]$$

What are appropriate encodings for projections?

$$\begin{aligned} \text{fst} \quad &= \quad \lambda p.p \text{ true} \\ \text{snd} \quad &= \quad \lambda p.p \text{ false} \end{aligned}$$

Numbers

Church's encoding of the natural numbers:

$$\begin{array}{rcl} \underline{0} & = & \lambda fx.x \\ \underline{1} & = & \lambda fx.fx \\ & & \dots \\ \underline{n} & = & \lambda fx.f^n x \\ & & \dots \end{array}$$

That is, the natural number n is represented by the Church numeral $\underline{n}$, which has the following property for any $F, X \in \Lambda$:

$$\underline{n}FX \twoheadrightarrow_\beta F^n X$$

One can see the natural numbers as iterators.

Arithmetic

Some basic functions on natural numbers:

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

Correctness of add:

$$\begin{aligned} \mathsf{add}\ \underline{m}\ \underline{n} &\twoheadrightarrow \lambda fx.\underline{m}f(\underline{n}fx) \\ &\twoheadrightarrow \lambda fx.f^m(f^n x) \\ &\twoheadrightarrow \lambda fx.f^{m+n}x = \underline{m+n} \end{aligned}$$

Other basic operations on numerals:

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

The predecessor function

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

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

This function can be encoded as:

$$\text{prefn} = \lambda fp.\text{pair}\ (f(\text{fst}\ p))\ (\text{fst}\ p)$$

Then

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

Subtraction can then be defined as:

$$\text{sub} = \lambda mn.n\ \text{pred}\ m.$$

Lists

Similar to what happens with Church numerals, we can encode lists:

$$[x_1, x_2, \ldots, x_n] = \lambda fy.f\ x_1\ (f\ x_2 \ldots (f\ x_n\ y) \ldots)$$

Alternatively, lists can be represented using pairs (as done in ML or Lisp).

$$[x_1, x_2, \ldots, x_n] = (x_1, (x_2, (\ldots (x_n, \mathrm{nil}) \ldots)))$$

List can simply be encode as:

$$\begin{array}{lcl} \mathrm{nil} & = & \lambda xy.y \\ \mathrm{cons} & = & \lambda ht.(\lambda z.zht) = \mathrm{pair} \end{array}$$

How can we encode hd and tl? (Exercise)

Lists

List can be also be encoded as the following λ-terms:

$$\begin{array}{lcl} \text{nil} & = & \text{pair true true} \\ \text{cons} & = & \lambda xy.\text{pair false (pair } x\ y) \end{array}$$

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:

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

A simpler encoding of the empty list is $\lambda z.z$. (Verify!)

Encoding numbers as pairs

Let us encode natural numbers as:

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

For example

$$\lceil 3 \rceil = [\text{false}, [\text{false}, [\text{false}, \lambda x.x]]]$$

where $[M, N]$ represents pair $M\ N$.

How can one define succ, pred and iszero using the representation for numbers above?

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

Recursive functions

Any recursive function F can be defined as:

$$F = \dots F \dots$$

For example `length` is implemented in Haskell as:

```
length [] = 0
length (x:xs) = 1 + length xs

length = \xs -> if xs==[] then 0
                          else (add 1 (length (tl xs)))
```

Recursive functions

We are then looking for a λ-term such that:

$$F = (\lambda f. \dots f \dots) F$$

Definition

A *fixed point* of the function F is any X such that $F(X) = X$.

Therefore, we are looking for a fixed point of $(\lambda f. \dots f \dots)$.

In the λ-calculus a term M is a fixed point of F if $F(M) =_\beta M$, and fixed point combinators can be used construct the fixed point of F.

A *fixed-point combinator* is a term $\mathbf{Y}$ such that $\mathbf{Y}F =_\beta F(\mathbf{Y}F)$, for all terms F.

Recursion in the λ-calculus

Haskell Curry's **Y** fix-point combinator is defined by:

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

We can verify easily the fixed point property:

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

This combinator is also called Curry's Paradoxical combinator:

- Consider an encoding of sets as predicates: $M \in N$ is encoded as $N(M)$, and $\{x \mid P\}$ is encoded as $\lambda x.P$.
- Taking $R = \lambda x.\mathbf{not}(xx)$, we get $RR = \mathbf{not}(RR)$, which is logically a contradiction.

Curry's fixed-point is defined by replacing **not** by any term F.

Famous fixed point combinators

Alan Turing's Θ is defined as

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

Now it is easy to verify that:

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

Another fixed-point combinator is Klop's \$ combinator:

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

(Verify!)

Examples of recursive functions

Coming back to our recursive definition:

$$M = (\lambda f. \dots f \dots)\; M$$

Using a fixed point combinator $\mathbf{Y}$ we can just define:

$$M = \mathbf{Y}(\lambda f. \dots f \dots)$$

Example:

$$\text{fact } n \quad = \quad \text{if (iszero } n)\ \underline{1}\ (\text{mult } n\ (\text{fact}(\text{pre } n)))$$

Then its recursive definition in the λ-calculus is:

$$\text{fact} \quad = \quad \mathbf{Y}(\lambda f n.\text{if (iszero } n)\ \underline{1}\ (\text{mult } n\ (f(\text{pre } n))))$$

Encoding partial recursive functions

A recursive function $f : \mathbb{N}^k \to \mathbb{N}$ is λ-definable if and only if there exists a λ-term F such that:

- If $f(n_1, \ldots, n_k) = m$, then $F \underline{n_1} \ldots \underline{n_k} =_\beta \underline{m}$.
- If $f(n_1, \ldots, n_k) = m$ is not defined, then $F \underline{n_1} \ldots \underline{n_k}$ does not have normal form.

We say that F defines function f. If F defines f then $F \underline{n_1} \ldots \underline{n_k} \twoheadrightarrow \underline{f(n_1, \ldots, n_k)}$.

The primitive recursive functions are λ-definable

The set of primitive recursive functions (PR) is defined as follows:

Function *zero*: $\mathsf{z}(x) = 0$

Function *successor*: $\mathsf{s}(x) = x + 1$

The *projections*: $\pi^n_k(x_1, \ldots, x_n) = x_k$, $1 \leq k \leq n$

Composition of primitive recursive functions f and $g_1, \ldots, g_k$:

$(f \circ (g_1, \ldots, g_k))(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$

Primitive Recursion: over primitive recursive functions f and g:

$$\begin{array}{rcl} h(x_1, \ldots, x_n, 0) & = & f(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n, y+1) & = & g(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)) \end{array}$$

Encoding partial recursive functions

Function z:

$$\mathsf{z} = \lambda n.\underline{0}$$

Function s:

$$\mathsf{s} = \lambda nfx.f(nfx)$$

Projections π_i^n:

$$\pi_i^n = \lambda x_1 \dots x_n.x_i$$

Composition:

$$W = \lambda x_1 \dots x_n.F(G_1 x_1 \dots x_n) \dots (G_k x_1 \dots x_n)$$

Encoding the primitive recursive scheme

Suppose f and g are λ-defined as F and G respectively. Now consider the following terms:

$$\begin{array}{lcl} \text{Init} & = & (0, Fx_1 \ldots x_k) \\ \text{Step} & = & \lambda p.(\text{succ}(\text{fst } p), G\ x_1 \ldots\ x_k\ (\text{fst } p)(\text{snd } p)) \end{array}$$

Then the primitive recursive scheme is function h is λ-definable as:

$$F = \lambda x x_1 \ldots x_k.\text{snd}(x \text{ Step Init})$$

This function generates the sequence of pairs:

$$(0, a_0), (1, a_1), \ldots, (n, a_n)$$

where $a_0 = f(n_1, \ldots, n_k)$ and $a_{i+1} = g(n_1, \ldots, n_k, i, a_i)$

Encoding (partial) recursive functions

Consider $f : \mathbb{N}^n \rightarrow N$, defined by minimisation:

$$f(x_1, \ldots, x_n) = \mu y.[g(x_1, \ldots, x_n, y) = 0]$$

Theorem (Kleene's normal form)

Every recursive function f can be defined as:

$$f(n_1, \ldots, n_k) = g(\mu y[h(n_1, \ldots, n_k, y) = 0])$$

where g and h are primitive recursive functions.

Encoding (partial) recursive functions

Let G and H be the terms that define the primitive recursive functions g and h, respectively. Let

$$W = \lambda y.\text{if } (\text{iszero}(Hx_1 \ldots x_k y))\ (\lambda w.Gy)\ (\lambda w.w(\text{succ } y)w)$$

Note that $x_1, \ldots x_k$ are free in W. Then the following λ-term defines f:

$$F = \lambda x_1 \ldots x_k.W\underline{0}W$$

Take any $n_1, \ldots, n_k$, then

$$F\underline{n_1} \ldots \underline{n_k} \twoheadrightarrow W[\underline{n_1}/x_1, \ldots, \underline{n_k}/x_k]\underline{0}W[\underline{n_1}/x_1, \ldots, \underline{n_k}/x_k]$$

Exercise: Verify that F behaves as expected.

Exercises

1. Write other (more direct) encodings of or, not and xor.
2. Show that the following functions on numbers are λ-definable.

   (i) the *constant* functions: $\mathsf{c}_n(x) = n$;
   (ii) the *signum* function: $\mathrm{sg}(0) = 0$ and $\mathrm{sg}(m+1) = 1$.

3. Show the correctness of the various encodings presented in this lecture.
4. Considering the following encoding of binary trees:

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

   can you define encodings for functions isLeaf, treeLeft and treeRight?