Church encoding

In mathematics, **Church encoding** is a means of representing data and operators in the lambda calculus. The **Church numerals** are a representation of the natural numbers using lambda notation. The method is named for Alonzo Church, who first encoded data in the lambda calculus this way.

Terms that are usually considered primitive in other notations (such as integers, booleans, pairs, lists, and tagged unions) are mapped to higher-order functions under Church encoding. The Church–Turing thesis asserts that any computable operator (and its operands) can be represented under Church encoding. In the untyped lambda calculus, the only primitive data type is the function.

The Church encoding is not intended as a practical implementation of primitive data types. Its use is to show that other primitive data types are not required to represent any calculation. The completeness is representational. Additional functions are needed to translate the representation into common data types, for display to people. It is not possible in general to decide if two functions are extensionally equal due to the undecidability of equivalence from Church's theorem. The translation may apply the function in some way to retrieve the value it represents, or look up its value as a literal lambda term.

Lambda calculus is usually interpreted as using intensional equality. There are potential problems with the interpretation of results because of the difference between the intensional and extensional definition of equality.

Contents

Church numerals

Church numerals are the representations of natural numbers under Church encoding. The higher-order function that represents natural number n is a function that maps any function f to its n-fold composition. In simpler terms, the "value" of the numeral is equivalent to the number of times the function encapsulates its argument.

$$f^{\circ n} = \underbrace{f \circ f \circ \cdots \circ f}_{n \text{ times}}.$$

All Church numerals are functions that take two parameters. Church numerals **0**, **1**, **2**, ..., are defined as follows in the lambda calculus.

Starting with **0** *not applying the function at all, proceed with* **1** *applying the function once, ...:*

Number	Function definition	Lambda expression
0	$\mathbf{0}\ f\ x = x$	$\mathbf{0} = \lambda f.\, \lambda x.\, x$
1	$\mathbf{1}\ f\ x = f\ x$	$\mathbf{1} = \lambda f.\, \lambda x.\, f\ x$
2	$\mathbf{2}\ f\ x = f\ (f\ x)$	$\mathbf{2} = \lambda f.\, \lambda x.\, f\ (f\ x)$
3	$\mathbf{3}\ f\ x = f\ (f\ (f\ x))$	$\mathbf{3} = \lambda f.\, \lambda x.\, f\ (f\ (f\ x))$
$\vdots$	$\vdots$	$\vdots$
n	$n\ f\ x = f^n\ x$	$n = \lambda f.\, \lambda x.\, f^{\circ n}\ x$

The Church numeral **3** represents the action of applying any given function three times to a value. The supplied function is first applied to a supplied parameter and then successively to its own result. The end result is not the numeral 3 (unless the supplied parameter happens to be 0 and the function is a successor function). The function itself, and not its end result, is the Church numeral **3**. The Church numeral **3** means simply to do anything three times. It is an ostensive demonstration of what is meant by "three times".

Calculation with Church numerals

Arithmetic operations on numbers may be represented by functions on Church numerals. These functions may be defined in lambda calculus, or implemented in most functional programming languages (see converting lambda expressions to functions).

The addition function $\mathbf{plus}(m, n) = m + n$ uses the identity $f^{\circ(m+n)}(x) = f^{\circ m}(f^{\circ n}(x))$.

$$\mathbf{plus} \equiv \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ f\ (n\ f\ x)$$

The successor function $\mathbf{succ}(n) = n + 1$ is β-equivalent to $(\mathbf{plus}\ 1)$.

$$\mathbf{succ} \equiv \lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x)$$

The multiplication function $\mathbf{mult}(m, n) = m * n$ uses the identity $f^{\circ(m*n)}(x) = (f^{\circ n})^{\circ m}(x)$.

$$\mathbf{mult} \equiv \lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ (n\ f)\ x$$

The exponentiation function $\mathbf{exp}(m, n) = m^n$ is given by the definition of Church numerals $n\ f\ x = f^n\ x$. In the definition substitute $f \to m, x \to f$ to get $n\ m\ f = m^n\ f$ and,

$$\mathbf{exp}\ \ m\ n = m^n = n\ m$$

which gives the lambda expression,

$$\mathbf{exp} \equiv \lambda m.\, \lambda n.\, n\ m$$

The $\mathbf{pred}(n)$ function is more difficult to understand.

$$\mathbf{pred} \equiv \lambda n.\, \lambda f.\, \lambda x.\, n\ (\lambda g.\, \lambda h.\, h\ (g\ f))\ (\lambda u.\, x)\ (\lambda u.\, u)$$

A Church numeral applies a function *n* times. The predecessor function must return a function that applies its parameter *n - 1* times. This is achieved by building a container around *f* and *x*, which is initialized in a way that omits the application of the function the first time. See predecessor for a more detailed explanation.

The subtraction function can be written based on the predecessor function.

$$\mathbf{minus} \equiv \lambda m.\, \lambda n.\, (n\, \mathbf{pred})\ m$$

Table of functions on Church numerals

Function	Algebra	Identity	Function definition	Lambda expressions	
Successor	$n + 1$	$f^{n+1}\ x = f(f^n x)$	$\mathbf{succ}\ \ n\ f\ x = f\ (n\ f\ x)$	$\lambda n.\, \lambda f.\, \lambda x.\, f\ (n\ f\ x)$	...
Addition	$m + n$	$f^{m+n}\ x = f^m (f^n x)$	$\mathbf{plus}\ \ m\ n\ f\ x = m\ f\ (n\ f\ x)$	$\lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ f\ (n\ f\ x)$	$\lambda m.\, \lambda n.\, n\, \mathbf{succ}\, m$
Multiplication	$m * n$	$f^{m*n}\ x = (f^m)^n\ x$	$\mathbf{multiply}\ \ m\ n\ f\ x = m\ (n\ f)\ x$	$\lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, m\ (n\ f)\ x$	$\lambda m.\, \lambda n.\, \lambda f.\, m\ (n\ f)$
Exponentiation	m^n	$n\ m\ f = m^n\ f$[1]	$\mathbf{exp}\ \ m\ n\ f\ x = (n\ m)\ f\ x$	$\lambda m.\, \lambda n.\, \lambda f.\, \lambda x.\, (n\ m)\ f\ x$	$\lambda m.\, \lambda n.\, n\ m$
Predecessor*	$n - 1$	$\mathbf{inc}^n\ \mathbf{con} = \mathbf{val}(f^{n-1} x)$	$if(n == 0)\ 0\ else\ (n - 1)$	$\lambda n.\, \lambda f.\, \lambda x.\, n\ (\lambda g.\, \lambda h.\, h\ (g\ f))\ (\lambda u.\, x)\ (\lambda u.\, u)$	
Subtraction*	$m - n$	$f^{m-n}\ x = (f^{-1})^n (f^m x)$	$\mathbf{minus}\ \ m\ n = (n\, \mathbf{pred})\ m$	...	$\lambda m.\, \lambda n.\, n\, \mathbf{pred}\, m$

* Note that in the Church encoding,

- $\mathbf{pred}(0) = 0$
- $m < n \to m - n = 0$

Translation with other representations

Most real-world languages have support for machine-native integers; the *church* and *unchurch* functions convert between nonnegative integers and their corresponding Church numerals. The functions are given here in Haskell, where the \ corresponds to the λ of Lambda calculus. Implementations in other languages are similar.

```haskell
type Church a = (a -> a) -> a -> a

church :: Integer -> Church Integer
church 0 = \f -> \x -> x
church n = \f -> \x -> f (church (n-1) f x)

unchurch :: Church Integer -> Integer
unchurch cn = cn (+ 1) 0
```

Church Booleans

Church Booleans are the Church encoding of the Boolean values *true* and *false*. Some programming languages use these as an implementation model for Boolean arithmetic; examples are Smalltalk and Pico.

Boolean logic may be considered as a choice. The Church encoding of *true* and *false* are functions of two parameters:

- *true* chooses the first parameter
- *false* chooses the second parameter

The two definitions are known as Church Booleans:

$$\mathbf{true} \equiv \lambda a.\, \lambda b.\, a$$
$$\mathbf{false} \equiv \lambda a.\, \lambda b.\, b$$

This definition allows predicates (i.e. functions returning logical values) to directly act as if-clauses. A function returning a Boolean, which is then applied to two parameters, returns either the first or the second parameter:

$$\textbf{predicate } x \textbf{ then-clause } \textbf{else-clause}$$

evaluates to *then-clause* if *predicate x* evaluates to *true*, and to *else-clause* if *predicate x* evaluates to *false*.

Because *true* and *false* choose the first or second parameter they may be combined to provide logic operators. Note that there are two version of *not*, depending on the evaluation strategy that is chosen.

$$\begin{aligned}
\textbf{and} &= \lambda p.\, \lambda q.\, p\ q\ p \\
\textbf{or} &= \lambda p.\, \lambda q.\, p\ p\ q \\
\textbf{not}_1 &= \lambda p.\, \lambda a.\, \lambda b.\, p\ b\ a \quad \textbf{(This is only a correct implementation if the evaluation strategy is applicative order.)} \\
\textbf{not}_2 &= \lambda p.\, p\ (\lambda a.\, \lambda b.\, b)\ (\lambda a.\, \lambda b.\, a) = \lambda p.\, p\, \textbf{false}\, \textbf{true} \quad \textbf{(This is only a correct implementation if the evaluation strategy is normal order.)} \\
\textbf{xor} &= \lambda a.\, \lambda b.\, a\ (\textbf{not}\ b)\ b \\
\textbf{if} &= \lambda p.\, \lambda a.\, \lambda b.\, p\ a\ b
\end{aligned}$$

Some examples:

$$\begin{aligned}
\textbf{and true false} &= (\lambda p.\, \lambda q.\, p\ q\ p)\ \ \textbf{true}\ \ \textbf{false} = \textbf{true false true} = (\lambda a.\, \lambda b.\, a)\, \textbf{false true} = \textbf{false} \\
\textbf{or true false} &= (\lambda p.\, \lambda q.\, p\ p\ q)\ (\lambda a.\, \lambda b.\, a)\ (\lambda a.\, \lambda b.\, b) = (\lambda a.\, \lambda b.\, a)\ (\lambda a.\, \lambda b.\, a)\ (\lambda a.\, \lambda b.\, b) = (\lambda a.\, \lambda b.\, a) = \textbf{true} \\
\textbf{not}_1\ \ \textbf{true} &= (\lambda p.\, \lambda a.\, \lambda b.\, p\ b\ a)(\lambda a.\, \lambda b.\, a) = \lambda a.\, \lambda b.\, (\lambda a.\, \lambda b.\, a)\ b\ a = \lambda a.\, \lambda b.\, (\lambda c.\, b)\ a = \lambda a.\, \lambda b.\, b = \textbf{false} \\
\textbf{not}_2\ \ \textbf{true} &= (\lambda p.\, p\ (\lambda a.\, \lambda b.\, b)(\lambda a.\, \lambda b.\, a))(\lambda a.\, \lambda b.\, a) = (\lambda a.\, \lambda b.\, a)(\lambda a.\, \lambda b.\, b)(\lambda a.\, \lambda b.\, a) = (\lambda b.\, (\lambda a.\, \lambda b.\, b))\ (\lambda a.\, \lambda b.\, a) = \lambda a.\, \lambda b.\, b = \textbf{false}
\end{aligned}$$

Predicates

A *predicate* is a function that returns a Boolean value. The most fundamental predicate is **IsZero**, which returns **true** if its argument is the Church numeral **0**, and **false** if its argument is any other Church numeral:

$$\textbf{IsZero} = \lambda n.\, n\ (\lambda x.\, \textbf{false})\ \textbf{true}$$

The following predicate tests whether the first argument is less-than-or-equal-to the second:

$$\textbf{LEQ} = \lambda m.\, \lambda n.\, \textbf{IsZero}\ (\textbf{minus}\ m\ n),$$

Because of the identity

$$x = y \equiv (x \leq y \wedge y \leq x)$$

The test for equality may be implemented as,

$$\textbf{EQ} = \lambda m.\, \lambda n.\, \textbf{and}\ (\textbf{LEQ}\ m\ n)\ (\textbf{LEQ}\ n\ m)$$

Church pairs

Church pairs are the Church encoding of the pair (two-tuple) type. The pair is represented as a function that takes a function argument. When given its argument it will apply the argument to the two components of the pair The definition in lambda calculus is,

$$\begin{aligned}
\textbf{pair} &\equiv \lambda x.\, \lambda y.\, \lambda z.\, z\ x\ y \\
\textbf{first} &\equiv \lambda p.\, p\ (\lambda x.\, \lambda y.\, x) \\
\textbf{second} &\equiv \lambda p.\, p\ (\lambda x.\, \lambda y.\, y)
\end{aligned}$$

For example,

$$\begin{aligned}
&\textbf{first}\ (\textbf{pair}\ a\ b) \\
=&(\lambda p.\, p\ (\lambda x.\, \lambda y.\, x))\ ((\lambda x.\, \lambda y.\, \lambda z.\, z\ x\ y)\ a\ b) \\
=&(\lambda p.\, p\ (\lambda x.\, \lambda y.\, x))\ (\lambda z.\, z\ a\ b) \\
=&(\lambda z.\, z\ a\ b)\ (\lambda x.\, \lambda y.\, x) \\
=&(\lambda x.\, \lambda y.\, x)\ a\ b = a
\end{aligned}$$

List encodings

An (immutable) list is constructed from list nodes.The basic operations on the list are;

Function	Description
nil	Construct an empty list.
isnil	Test if list is empty.
cons	Prepend a given value to a (possibly empty) list.
head	Get the first element of the list.
tail	Get the rest of the list.

Three different representations of lists are given.

- Build each list node from two pairs (to allow for empty lists).
- Build each list node from one pair
- Represent the list using the right fold function

Two pairs as a list node

A nonempty list can be implemented by a Church pair;

- *First* contains the head.

- *Second* contains the tail.

However this does not give a representation of the empty list, because there is no "null" pointer. To represent null, the pair may be wrapped in another pair, giving free values,

- *First* - Is the null pointer (empty list).
- *Second.First* contains the head.
- *Second.Second* contains the tail.

Using this idea the basic list operations can be defined like this:[2]

Expression	Description
$\mathbf{nil} \equiv \mathbf{pair}\ \ \mathbf{true}\ \ \mathbf{true}$	The first element of the pair is *true* meaning the list is null.
$\mathbf{isnil} \equiv \mathbf{first}$	Retrieve the null (or empty list) indicator
$\mathbf{cons} \equiv \lambda h.\, \lambda t.\, \mathbf{pair}\, \mathbf{false}\ (\mathbf{pair}\, h\, t)$	Create a list node, which is not null, and give it a head *h* and a tail *t*.
$\mathbf{head} \equiv \lambda z.\, \mathbf{first}\ (\mathbf{second}\, z)$	*second.first* is the head.
$\mathbf{tail} \equiv \lambda z.\, \mathbf{second}\ (\mathbf{second}\, z)$	*second.second* is the tail.

In a *nil* node *second* is never accessed, provided that **head** and **tail** are only applied to nonempty lists.

One pair as a list node

Alternatively, define[3]

$$\begin{aligned}\mathbf{cons} &\equiv \mathbf{pair}\\ \mathbf{head} &\equiv \mathbf{first}\\ \mathbf{tail} &\equiv \mathbf{second}\\ \mathbf{nil} &\equiv \mathbf{false}\\ \mathbf{isnil} &\equiv \lambda l.\, l(\lambda h.\, \lambda t.\, \lambda d.\, \mathbf{false})\, \mathbf{true}\end{aligned}$$

where the last definition is a special case of the general

$$\mathbf{process\text{-}list} \equiv \lambda l.\, l(\lambda h.\, \lambda t.\, \lambda d.\, \mathbf{head\text{-}and\text{-}tail\text{-}clause})\, \mathbf{nil\text{-}clause}$$

Represent the list using *right fold*

As an alternative to the encoding using Church pairs, a list can be encoded by identifying it with its right fold function For example, a list of three elements x, y and z can be encoded by a higher-order function that when applied to a combinator c and a value n returns c x (c y (c z n)).

$$\begin{aligned}\mathbf{nil} &\equiv \lambda c.\, \lambda n.\, n\\ \mathbf{isnil} &\equiv \lambda l.\, l\ (\lambda h.\, \lambda t.\, \mathbf{false})\ \mathbf{true}\\ \mathbf{cons} &\equiv \lambda h.\, \lambda t.\, \lambda c.\, \lambda n.\, c\ h\ (t\ c\ n)\\ \mathbf{head} &\equiv \lambda l.\, l\ (\lambda h.\, \lambda t.\, h)\ \mathbf{false}\\ \mathbf{tail} &\equiv \lambda l.\, \lambda c.\, \lambda n.\, l\ (\lambda h.\, \lambda t.\, \lambda g.\, g\ h\ (t\ c))\ (\lambda t.\, n)\ (\lambda h.\, \lambda t.\, t)\end{aligned}$$

Derivation of predecessor function

The predecessor function used in the Church encoding is,

$$\mathbf{pred}(n) = \begin{cases} 0 & \text{if } n = 0, \\ n-1 & \text{otherwise} \end{cases}.$$

To build the predecessor we need a way of applying the function 1 fewer time. A numeral n applies the function f n times to x. The predecessor function must use the numeral n to apply the function n-1 times.

Before implementing the predecessor function, here is a scheme that wraps the value in a container function. We will define new functions to use in place of f and x, called inc and init. The container function is called value. The left hand side of the table shows a numeral n applied to inc and init.

Number	Using init	Using const
0	$\mathbf{init} = \mathbf{value}\ x$	
1	$\mathbf{inc}\ \ \mathbf{init} = \mathbf{value}\ (f\, x)$	$\mathbf{inc}\ \ \mathbf{const} = \mathbf{value}\ x$
2	$\mathbf{inc}\ (\mathbf{inc}\ \ \mathbf{init}) = \mathbf{value}\ (f\, (f\, x))$	$\mathbf{inc}\ (\mathbf{inc}\ \ \mathbf{const}) = \mathbf{value}\ (f\, x)$
3	$\mathbf{inc}\ (\mathbf{inc}\ (\mathbf{inc}\ \ \mathbf{init})) = \mathbf{value}\ (f\, (f\, (f\, x)))$	$\mathbf{inc}\ (\mathbf{inc}\ (\mathbf{inc}\ \ \mathbf{const})) = \mathbf{value}\ (f\, (f\, x))$
$\vdots$	$\vdots$	$\vdots$
n	$n\,\mathbf{inc}\ \ \mathbf{init} = \mathbf{value}\ (f^n\ x) = \mathbf{value}\ (n\ f\ x)$	$n\,\mathbf{inc}\ \ \mathbf{const} = \mathbf{value}\ (f^{n-1}\ x) = \mathbf{value}\ ((n-1)\ f\ x)$

The general recurrence rule is,

$$\mathbf{inc}\ (\mathbf{value}\ v) = \mathbf{value}\ (f\, v)$$

If there is also a function to retrieve the value from the container (called extract),

$$\mathbf{extract}\ (\mathbf{value}\ v) = v$$

Then extract may be used to define the samenum function as,

$$\mathbf{samenum} = \lambda n.\, \lambda f.\, \lambda x.\, \mathbf{extract}\ (n\,\mathbf{inc}\,\mathbf{init}) = \lambda n.\, \lambda f.\, \lambda x.\, \mathbf{extract}\ (\mathbf{value}\ (n\ f\ x)) = \lambda n.\, \lambda f.\, \lambda x.\, n\ f\ x = \lambda n.\, n$$

The samenum function is not intrinsically useful. However, as inc delegates calling of f to its container argument, we can arrange that on the first application inc receives a special container that ignores its argument allowing to skip the first application of f. Call this new initial container const. The right hand side of the above table shows the expansions of n inc const. Then by replacing init with const in the expression for thesame function we get the predecessor function,

$$\mathbf{pred} = \lambda n.\, \lambda f.\, \lambda x.\, \mathbf{extract}\ (n\, \mathbf{inc}\, \mathbf{const}) = \lambda n.\, \lambda f.\, \lambda x.\, \mathbf{extract}\ (\mathbf{value}\ ((n-1)\ f\ x)) = \lambda n.\, \lambda f.\, \lambda x.\, (n-1)\ f\ x = \lambda n.\, (n-1)$$

As explained below the functionsinc, init, const, value and extract may be defined as,

$$\begin{aligned} \mathbf{value} &= \lambda v.\, (\lambda h.\, h\ v) \\ \mathbf{extract}\ k &= k\ \lambda u.\, u \\ \mathbf{inc} &= \lambda g.\, \lambda h.\, h\ (g\ f) \\ \mathbf{init} &= \lambda h.\, h\ x \\ \mathbf{const} &= \lambda u.\, x \end{aligned}$$

Which gives the lambda expression for pred as,

$$\mathbf{pred} = \lambda n.\, \lambda f.\, \lambda x.\, n\ (\lambda g.\, \lambda h.\, h\ (g\ f))\ (\lambda u.\, x)\ (\lambda u.\, u)$$

Value container

The value container applies a function to its valueIt is defined by,

$$\mathbf{value}\ v\ h = h\ v$$

so,

$$\mathbf{value} = \lambda v.\, (\lambda h.\, h\ v)$$

Inc

The inc function should take a value containing v, and return a new value containing $f\ v$.

$$\mathbf{inc}\ (\mathbf{value}\ v) = \mathbf{value}\ (f\ v)$$

Letting g be the value container,

$$g = \mathbf{value}\ v$$

then,

$$g\ f = \mathbf{value}\ v\ f = f\ v$$

so,

$$\begin{aligned} &\mathbf{inc}\ g = \mathbf{value}\ (g\ f) \\ &\mathbf{inc} = \lambda g.\, \lambda h.\, h\ (g\ f) \end{aligned}$$

Extract

The value may be extracted by applying the identity function,

$$I = \lambda u.\, u$$

Using I,

$$\mathbf{value}\ v\ I = v$$

so,

$$\mathbf{extract}\ k = k\ I$$

Const

To implement pred the init function is replaced with the const that does not apply f. We need const to satisfy,

$$\begin{aligned} &\mathbf{inc}\ \ \mathbf{const} = \mathbf{value}\ x \\ &\lambda h.\, h\ (\mathbf{const}\ \ f) = \lambda h.\, h\ x \end{aligned}$$

Which is satisfied if,

$$\mathbf{const}\ \ f = x$$

Or as a lambda expression,

$$\mathbf{const} = \lambda u.\, x$$

Another way of defining pred

Pred may also be defined using pairs:

$$\begin{aligned} \mathbf{f} &= \lambda p.\ \ \mathbf{pair}\ (\mathbf{second}\ p)\ (\mathbf{succ}\ (\mathbf{second}\ p)) \\ \mathbf{zero} &= (\lambda f.\, \lambda x.\ \ x) \\ \mathbf{pc0} &= \mathbf{pair}\ \ \mathbf{zero}\ \ \mathbf{zero} \\ \mathbf{pred} &= \lambda n.\ \ \mathbf{first}\ \ (n\ f\ \ \mathbf{pc0}) \end{aligned}$$

This is a simpler definition, but leads to a more complex expression for pred.The expansion for **pred three**:

$$\begin{aligned} \mathbf{pred\,three} &= \mathbf{first}\ \ (\mathbf{f}\ \ (\mathbf{f}\ \ (\mathbf{f}\ \ (\mathbf{pair}\ \ \mathbf{zero}\ \ \mathbf{zero})))) \\ &= \mathbf{first}\ \ (\mathbf{f}\ \ (\mathbf{f}\ \ (\mathbf{pair}\ \ \mathbf{zero}\ \ \mathbf{one}))) \\ &= \mathbf{first}\ \ (\mathbf{f}\ \ (\mathbf{pair}\ \ \mathbf{one}\ \ \mathbf{two})) \\ &= \mathbf{first}\ \ (\mathbf{pair}\ \ \mathbf{two}\ \ \mathbf{three}) \\ &= \mathbf{two} \end{aligned}$$

Division

Division of natural numbers may be implemented by[4]

$$n/m = \mathbf{if}\ \ n \geq m\ \ \mathbf{then}\ \ 1 + (n-m)/m\ \ \mathbf{else}\ \ 0$$

Calculating $n - m$ takes many beta reductions. Unless doing the reduction by hand, this doesn't matter that much, but it is preferable to not have to do this calculation twice. The simplest predicate for testing numbers is *IsZero* so consider the condition.

$$\mathbf{IsZero}\ \ (\mathbf{minus}\ \ n\ m)$$

But this condition is equivalent to $n \leq m$, not $n < m$. If this expression is used then the mathematical definition of division given above is translated into function on Church numerals as,

$$\mathbf{divide1}\ \ n\ m\ f\ x = (\lambda d.\, \mathbf{IsZero}\ \ d\ (0\ f\ x)\ (f\ (\mathbf{divide1}\ \ d\ m\ f\ x)))\ (\mathbf{minus}\ \ n\ m)$$

As desired, this definition has a single call to $\mathbf{minus}\ n\ m$. However the result is that this formula gives the value of $(n-1)/m$.

This problem may be corrected by adding 1 to n before calling *divide*. The definition of *divide* is then,

$$\mathbf{divide}\ n = \mathbf{divide1}\ (\mathbf{succ}\ n)$$

divide1 is a recursive definition. The Y combinator may be used to implement the recursion. Create a new function called *div* by;

- In the left hand side $\mathbf{divide1} \rightarrow \mathbf{div}\ c$
- In the right hand side $\mathbf{divide1} \rightarrow c$

to get,

$$\mathbf{div} = \lambda c.\ \lambda n.\ \lambda m.\ \lambda f.\ \lambda x.\ (\lambda d.\ \mathbf{IsZero}\ d\ (0\ f\ x)\ (f\ (c\ d\ m\ f\ x)))\ (\mathbf{minus}\ n\ m)$$

Then,

$$\mathbf{divide} = \lambda n.\ \mathbf{divide1}\ (\mathbf{succ}\ n)$$

where,

$$\begin{aligned}
\mathbf{divide1} &= Y\ \mathbf{div} \\
\mathbf{succ} &= \lambda n.\ \lambda f.\ \lambda x.\ f\ (n\ f\ x) \\
Y &= \lambda f.\ (\lambda x.\ f\ (x\ x))\ (\lambda x.\ f\ (x\ x)) \\
0 &= \lambda f.\ \lambda x.\ x \\
\mathbf{IsZero} &= \lambda n.\ n\ (\lambda x.\ \mathbf{false})\ \mathbf{true} \\
\\
\mathbf{true} &\equiv \lambda a.\ \lambda b.\ a \\
\mathbf{false} &\equiv \lambda a.\ \lambda b.\ b \\
\\
\mathbf{minus} &= \lambda m.\ \lambda n.\ n\ \mathbf{pred}\ m \\
\mathbf{pred} &= \lambda n.\ \lambda f.\ \lambda x.\ n\ (\lambda g.\ \lambda h.\ h\ (g\ f))\ (\lambda u.\ x)\ (\lambda u.\ u)
\end{aligned}$$

Gives,

$$\mathbf{divide} = \lambda n.((\lambda f.(\lambda x.x\ x)\ (\lambda x.f\ (x\ x)))\ (\lambda c.\lambda n.\lambda m.\lambda f.\lambda x.(\lambda d.(\lambda n.n\ (\lambda x.(\lambda a.\lambda b.b))\ (\lambda a.\lambda b.a))\ d\ ((\lambda f.\lambda x.x)\ f\ x)\ (f\ (c\ d\ m\ f\ x)))\ ((\lambda m.\lambda n.n(\lambda n.\lambda f.\lambda x.n\ (\lambda g.\lambda h.h\ (g\ f))\ (\lambda u.x)\ (\lambda u.u))m)\ n\ m)))$$

Or as text, using \ for λ,

```
divide = (\n.((\f.(\x.x x) (\x.f (x x))) (\c.\n.\m.\f.\x.(\d.(\n.n (\x.(\a.\b.b)) (\a.\b.a)) d ((\f.\x.x) f x) (f (c d m f x))) ((\m.\n.n (\n.\f.\x.n (\g.\h.h
(g f)) (\u.x) (\u.u)) m) n m))) ((\n.\f.\x. f (n f x)) n))
```

For example, 9/3 is represented by

```
divide (\f.\x.f (f (f (f (f (f (f (f (f x)))))))) (\f.\x.f (f (f x)))
```

Using a lambda calculus calculator, the above expression reduces to 3, using normal order

```
\f.\x.f (f (f (x)))
```

Signed numbers

One simple approach for extending Church Numerals to signed numbers is to use a Church pair, containing Church numerals representing a positive and a negative value.[5] The integer value is the difference between the two Church numerals.

A natural number is converted to a signed number by

$$\mathbf{convert}_s = \lambda x.\ \mathbf{pair}\ x\ 0$$

Negation is performed by swapping the values.

$$\mathbf{neg}_s = \lambda x.\ \mathbf{pair}\ (\mathbf{second}\ x)\ (\mathbf{first}\ x)$$

The integer value is more naturally represented if one of the pair is zero. The *OneZero* function achieves this condition,

$$\mathbf{OneZero} = \lambda x.\ \mathbf{IsZero}\ (\mathbf{first}\ x)\ x\ (\mathbf{IsZero}\ (\mathbf{second}\ x)\ x\ (\mathbf{OneZero}\ \ \mathbf{pair}\ (\mathbf{pred}\ (\mathbf{first}\ x))\ (\mathbf{pred}\ (\mathbf{second}\ x))))$$

The recursion may be implemented using the Y combinator,

$$\mathbf{OneZ} = \lambda c.\ \lambda x.\ \mathbf{IsZero}\ (\mathbf{first}\ x)\ x\ (\mathbf{IsZero}\ (\mathbf{second}\ x)\ x\ (c\ \mathbf{pair}\ (\mathbf{pred}\ (\mathbf{first}\ x))\ (\mathbf{pred}\ (\mathbf{second}\ x))))$$

$$\mathbf{OneZero} = Y\ \mathbf{OneZ}$$

Plus and minus

Addition is defined mathematically on the pair by

$$x + y = [x_p, x_n] + [y_p, y_n] = x_p - x_n + y_p - y_n = (x_p + y_p) - (x_n + y_n) = [x_p + y_p, x_n + y_n]$$

The last expression is translated into lambda calculus as,

$$\mathbf{plus}_s = \lambda x.\ \lambda y.\ \mathbf{OneZero}\ (\mathbf{pair}\ (\mathbf{plus}\ (\mathbf{first}\ x)\ (\mathbf{first}\ y))\ (\mathbf{plus}\ (\mathbf{second}\ x)\ (\mathbf{second}\ y)))$$

Similarly subtraction is defined,

$$x - y = [x_p, x_n] - [y_p, y_n] = x_p - x_n - y_p + y_n = (x_p + y_n) - (x_n + y_p) = [x_p + y_n, x_n + y_p]$$

giving,

$$\mathbf{minus}_s = \lambda x.\, \lambda y.\, \mathbf{OneZero}\ (\mathbf{pair}\ (\mathbf{plus}\ (\mathbf{first}\ x)\ (\mathbf{second}\ y))\ (\mathbf{plus}\ (\mathbf{second}\ x)\ (\mathbf{first}\ y)))$$

Multiply and divide

Multiplication may be defined by

$$x * y = [x_p, x_n] * [y_p, y_n] = (x_p - x_n) * (y_p - y_n) = (x_p * y_p + x_n * y_n) - (x_p * y_n + x_n * y_p) = [x_p * y_p + x_n * y_n, x_p * y_n + x_n * y_p]$$

The last expression is translated into lambda calculus as,

$$\mathbf{mult}_s = \lambda x.\, \lambda y.\, \mathbf{pair}\ (\mathbf{plus}\ (\mathbf{mult}\ (\mathbf{first}\ x)\ (\mathbf{first}\ y))\ (\mathbf{mult}\ (\mathbf{second}\ x)\ (\mathbf{second}\ y)))\ (\mathbf{plus}\ (\mathbf{mult}\ (\mathbf{first}\ x)\ (\mathbf{second}\ y))\ (\mathbf{mult}\ (\mathbf{second}\ x)\ (\mathbf{first}$$

A similar definition is given here for division, except in this definition, one value in each pair must be zero (see *OneZero* above). The *divZ* function allows us to ignore the value that has a zero component.

$$\mathbf{divZ} = \lambda x.\, \lambda y.\, \mathbf{IsZero}\ y\ \mathbf{0}\ (\mathbf{divide}\ x\ y)$$

divZ is then used in the following formula, which is the same as for multiplication, but with *mult* replaced by *divZ*.

$$\mathbf{divide}_s = \lambda x.\, \lambda y.\, \mathbf{pair}\ (\mathbf{plus}\ (\mathbf{divZ}\ (\mathbf{first}\ x)\ (\mathbf{first}\ y))\ (\mathbf{divZ}\ (\mathbf{second}\ x)\ (\mathbf{second}\ y)))\ (\mathbf{plus}\ (\mathbf{divZ}\ (\mathbf{first}\ x)\ (\mathbf{second}\ y))\ (\mathbf{divZ}\ (\mathbf{second}\ x)\ (\mathbf{firs$$

Rational and real numbers

Rational and real numbers may also be encoded in lambda calculus. Rational numbers may be encoded as a pair of signed numbers. Real numbers may be encoded by a limiting process that guarantees that the difference from the real value differs by a number which may be made as small as we need.[6] [7] The references given describe software that could, in theory, be translated into lambda calculus. Once real numbers are defined, complex numbers are naturally encoded as a pair of real numbers.

The data types and functions described above demonstrate that any data type or calculation may be encoded in lambda calculus. This is the Church-Turing thesis.

See also

- Lambda calculus
- System F for Church numerals in a typed calculus
- Mogensen–Scott encoding
- Von Neumann definition of ordinals — another way to encode natural numbers: as sets

Notes

1. This formula is the definition of a Church numeral n with f -> m, x -> f.
2. Pierce, Benjamin C. (2002). *Types and Programming Languages*. MIT Press. p. 500. ISBN 978-0-262-16209-8
3. Tromp, John (2007). "14. Binary Lambda Calculus and Combinatory Logic". In Calude, Cristian S. *Randomness And Complexity From Leibniz To Chaitin* (https://books.google.com/books?id=flPlCgAAQBAJ&pg=PP1). World Scientific. pp. 237–262. ISBN 978-981-4474-39-9
   As PDF: Tromp, John (14 May 2014). "Binary Lambda Calculus and Combinatory Logic" (https://tromp.github.io/cl/LC.pdf) (PDF). Retrieved 2017-11-24.
4. Allison, Lloyd. "Lambda Calculus Integers" (http://www.csse.monash.edu.au/~lloyd/tildeFP/Lambda/Examples/const-int/)
5. Bauer, Andrej. "Andrej's answer to a question; "Representing negative and complex numbers using lambda calculus"" (http://cs.stackexchange.com/questions/2272/representing-negative-and-complex-numbers-using-lambda-calculus)
6. "Exact real arithmetic" (http://www.haskell.org/haskellwiki/Exact_real_arithmetic). *Haskell*.
7. Bauer, Andrej. "Real number computational software" (https://github.com/andrejbauer/marshall)

References

- Directly Reflective Meta-Programming
- Church numerals and booleans explained by Robert Cartwright at Rice University
- Theoretical Foundations For Practical 'Totally Functional Programming' (Chapters 2 and 5) All about Church and other similar encodings, including how to derive them and operations on them, from first principles
- Some interactive examples of Church numerals
- Lambda Calculus Live Tutorial: Boolean Algebra