Chapter 4

The Lambda-Calculus

The original motivation of Alonzo Church for inventing the λ-calculus was to provide a type-free foundation for mathematics (alternate to set theory) based on higher-order logic and the notion of *function* in the early 1930's (1932, 1933).

This attempt to provide such a foundation for mathematics failed due to a form of Russell's paradox.

Church was clever enough to turn the technical reason for this failure, the existence of fixed-point combinators, into a success, namely to view the λ-calculus as a formalism for defining the notion of *computability* (1932,1933,1935).

The λ-calculus is indeed one of the first computation models, slightly preceding the Turing machine.

Kleene proved in 1936 that all the computable functions (recursive functions) in the sense of Herbrand and Gödel are definable in the λ-calculus, showing that the λ-calculus has *universal computing power*.

In 1937, Turing proved that Turing machines compute the same class of computable functions. (This paper is very hard to read, in part because the definition of a Turing machine is not included in this paper).

In short, the λ-calculus and Turing machines have *the same computing power*.

Here we have to be careful. To be precise we should have said that all the *total* computable functions (total recursive functions) are definable in the λ-calculus.

In fact, it is also true that all the *partial* computable functions (partial recursive functions) are definable in the λ-calculus but this requires more care.

Since the λ-calculus does not have any notion of tape, register, or any other means of storing data, it quite amazing that the λ-calculus has so much computing power.

The λ-calculus is based on three concepts:

(1) Application.

(2) Abstraction (also called λ-abstraction).

(3) β-reduction (and β-conversion).

If f is a function, say the exponential function $f\colon \mathbb{N} \to \mathbb{N}$ given by $f(n) = 2^n$, and if n a natural number, then the result of applying f to a natural number, say 5, is written as

$$(f5)$$

and is called an *application*.

Here we can agree that f and 5 do not have the same *type*, in the sense that f is a function and 5 is a number, so applications such as $(f\,f)$ or (55) do not make sense, but the λ-calculus is *type-free* so expressions such as $(f\,f)$ as allowed.

This may seem silly, and even possibly undesirable, but allowing self application turns out to a major reason for the computing power of the λ-calculus.

Given an expression M containing a variable x, say

$$M(x) = x^2 + x + 1,$$

as x ranges over $\mathbb{N}$, we obtain the function respresented in standard mathematical notation by $x \mapsto x^2 + x + 1$.

If we supply the input value 5 for x, then the value of the function is $5^2 + 5 + 1 = 31$.

Church introduced the notation

$$\lambda x.\,(x^2 + x + 1)$$

for this function.

Here, we have an *abstraction*, in the sense that the static expression $M(x)$ for x fixed becomes an "abstract" function denoted $\lambda x.\,M$.

It would be pointless to only have the two concepts of application and abstraction.

The glue between these two notions is a form of evaluation called *β-reduction*.[1]

Given a λ-abstraction $\lambda x.\,M$ and some other term N (thought of as an argument), we have the "evaluation" rule, we say *β-reduction*,

$$(\lambda x.\,M)N \xrightarrow{+}_{\beta} M[x := N],$$

where $M[x := N]$ denotes the result of substituting N for all occurrences of x in M.

[1] Apparently, Church was fond of Greek letters.

For example, if $M = \lambda x.\,(x^2 + x + 1)$ and $N = 2y + 1$, we have

$$(\lambda x.\,(x^2 + x + 1))(2y + 1) \xrightarrow{+}_{\beta} (2y + 1)^2 + 2y + 1 + 1.$$

Observe that β-reduction is a *purely formal* operation (plugging N wherever x occurs in M), and that the expression $(2y + 1)^2 + 2y + 1 + 1$ is *not* instantly simplified to $4y^2 + 6y + 3$.

In the λ-calculus, the natural numbers as well as the arithmetic operations $+$ and $\times$ need to be represented as λ-terms in such a way that they "evaluate" correctly using only β-conversion.

In this sense, the λ-calculus is an incredibly low-level programming language. Nevertheless, the λ-calculus is the core of various *functional programming languages* such as *OCaml*, *ML*, *Miranda* and *Haskell*, among others.

We now proceed with precise definitions and results. But first we ask the reader not to think of functions as the functions we encounter in analysis or algebra. Instead think of functions as *rules for computing* (by moving and plugging arguments around), a more combinatory (which does not mean combinatorial) viewpoint.

4.1 Syntax of the Lambda-Calculus

We begin by defining the *lambda-calculus*, also called *untyped lambda-calculus* or *pure lambda-calculus*, to emphasize that the terms of this calculus are not typed.

This formal system consists of

1. A set of terms, called *λ-terms*.
2. A notion of reduction, called *β-reduction*, which allows a term M to be transformed into another term N in a way that mimics a kind of evaluation.

First we define (pure) λ-terms.

We have a countable set of variables $\{x_0, x_1, \ldots, x_n \ldots\}$ that correspond to the atomic λ-terms.

Definition 4.1. The *λ-terms* M are defined inductively as follows:

(1) If x_i is a variable, then x_i is a λ-term.

(2) If M and N are λ-terms, then (MN) is a λ-term called an *application*.

(3) If M is a λ-term, and x is a variable, then the expression $(\lambda x.\, M)$ is a λ-term called a *λ-abstraction*.

The fact that self-application is allowed in the untyped λ-calculus is what gives it its computational power (through fixed-point combinators, see Section 4.5).

Definition 4.2. The *depth* $d(M)$ of a λ-term M is defined inductively as follows.

1. If M is a variable x, then $d(x) = 0$.
2. If M is an application $(M_1 M_2)$, then $d(M) = \max\{d(M_1), d(M_2)\} + 1$.
3. If M is a λ-abstraction $(\lambda x.\, M_1)$, then $d(M) = d(M_1) + 1$.

It is pretty clear that λ-terms have representations as (ordered) labeled trees.

Definition 4.3. Given a λ-term M, the *tree* tree(M) representing M is defined inductively as follows:

1. If M is a variable x, then tree(M) is the one-node tree labeled x.

2. If M is an application $(M_1 M_2)$, then tree(M) is the tree with a binary root node labeled . , and with a left subtree tree(M_1) and a right subtree tree(M_2).

3. If M is a λ-abstraction $(\lambda x.\, M_1)$, then tree(M) is the tree with a unary root node labeled λx, and with one subtree tree(M_1).

Definition 4.3 is illustrated in Figure 4.1.

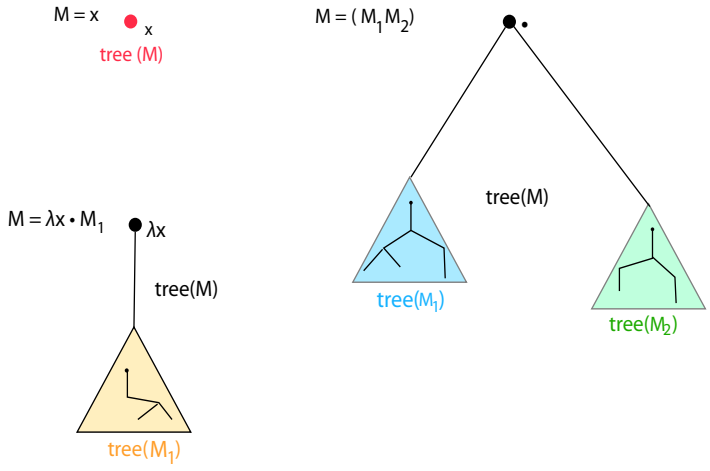

Figure 4.1: The tree tree(M) associated with a pure λ-term M.

Obviously, the depth $d(M)$ of λ-term is the depth of its tree representation tree(M).

Unfortunately λ-terms contain a profusion of parentheses so some conventions are commonly used:

(1) A term of the form

$$(\cdots((FM_1)M_2)\cdots M_n)$$

is abbreviated (association to the left) as

$$FM_1\cdots M_n.$$

(2) A term of the form

$$(\lambda x_1.\,(\lambda x_2.\,(\cdots(\lambda x_n.\,M)\cdots)))$$

is abbreviated (association to the right) as

$$\lambda x_1\cdots x_n.\,M.$$

Matching parentheses may be dropped or added for convenience. Here are some examples of λ-terms (and their abbreviation):

y	y
(yx)	yx
$(\lambda x.\,(yx))$	$\lambda x.\,yx$
$((\lambda x.\,(yx))z)$	$(\lambda x.\,yx)z$
$(((\lambda x.\,(\lambda y.\,(yx)))z)w)$	$(\lambda xy.\,yx)zw.$

Note that $\lambda x.\,yx$ is an abbreviation for $(\lambda x.\,(yx))$, not $((\lambda x.\,y)x)$.

The variables occurring in a λ-term are free or bound.

Definition 4.4. For any λ-term M, the set $FV(M)$ of *free variables* of M and the set $BV(M)$ of *bound variables* in M are defined inductively as follows:

(1) If $M = x$ (a variable), then

$$FV(x) = \{x\}, \quad BV(x) = \emptyset.$$

(2) If $M = (M_1 M_2)$, then

$$\begin{aligned} FV(M) &= FV(M_1) \cup FV(M_2) \\ BV(M) &= BV(M_1) \cup BV(M_2). \end{aligned}$$

(3) if $M = (\lambda x.\, M_1)$, then

$$\begin{aligned} FV(M) &= FV(M_1) - \{x\} \\ BV(M) &= BV(M_1) \cup \{x\}. \end{aligned}$$

If $x \in FV(M_1)$, we say that the occurrences of the variable x *occur in the scope of* λ.

A λ-term M is *closed* or a *combinator* if $FV(M) = \emptyset$, that is, if it has no free variables.

For example

$$FV\big((\lambda x.\, yx)z\big) = \{y, z\}, \quad BV\big((\lambda x.\, yx)z\big) = \{x\},$$

and

$$FV\big((\lambda xy.\, yx)zw\big) = \{z, w\}, BV\big((\lambda xy.\, yx)zw\big) = \{x, y\}.$$

Before proceeding with the notion of substitution we must address an issue with bound variables.

The point is that bound variables are really *place-holders* so they can be renamed freely without changing the reduction behavior of the term as long as they do not clash with free variables.

For example, the terms $\lambda x.\,(x(\lambda y.\,x(yx))$ and $\lambda x.\,(x(\lambda z.\,x(zx))$ should be considered as equivalent.

Similarly, the terms $\lambda x.\,(x(\lambda y.\,x(yx))$ and $\lambda w.\,(w(\lambda z.\,w(zw))$ should be considered as equivalent.

One way to deal with this issue is to use the tree representation of λ-terms given in Definition 4.3.

For every leaf labeled with a bound variable x, we draw a backpointer to an ancestor of x determined as follows.

Given a leaf labeled with a bound variable x, climb up to the closest ancestor labeled λx, and draw a backpointer to this node. Then all bound variables can be erased.

An example is shown in Figure 4.2 for the term $M = \lambda x.\, x(\lambda y.\, (x(yx)))$.

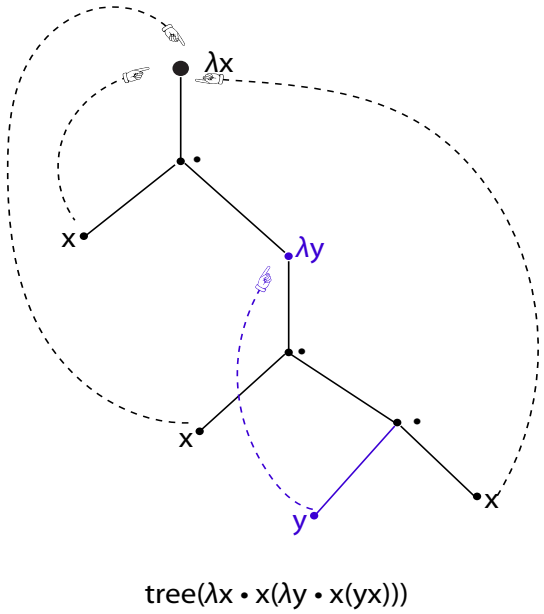

Figure 4.2: The tree representation of a λ-term with backpointers.

A clever implementation of the idea of backpointers is the formalism of *de Bruijn indices*; see Pierce [**?**] (Chapter 6) or Barendregt [**?**] (Appendix C).

Church introduced the notion of *α-conversion* to deal with this issue. First we need to define substitutions.

A *substitution* φ is a finite set of pairs
$\varphi = \{(x_1, N_1), \ldots, (x_n, N_n)\}$, where the x_i are distinct variables and the N_i are λ-terms.

We write

$$\varphi = [N_1/x_1, \ldots, N_n/x_n] \quad \text{or}$$
$$\varphi = [x_1 := N_1, \ldots, x_n := N_n].$$

The second notation indicates more clearly that each term N_i is substituted for the variable x_i, and it seems to have been almost universally adopted.

Given a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, for any variable x_i, we denote by φ_{-x_i} the new substitution where the pair (x_i, N_i) is replaced by the pair (x_i, x_i) (that is, the new substitution leaves x_i unchanged).

Definition 4.5. Given any λ-term M and any substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, we define the λ-term $M[\varphi]$, *the result of applying the substitution φ to M*, as follows:

(1) If $M = y$, with $y \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = y = M$.

(2) If $M = x_i$ for some $i \in \{1, \ldots, n\}$, then $M[\varphi] = N_i$.

(3) If $M = (PQ)$, then $M[\varphi] = (P[\varphi]Q[\varphi])$.

(4) If $M = \lambda x.\, N$ and $x \neq x_i$ for $i = 1, \ldots, n$, then $M[\varphi] = \lambda x.\, N[\varphi]$,

(5) If $M = \lambda x.\, N$ and $x = x_i$ for some $i \in \{1, \ldots, n\}$, then
$M[\varphi] = \lambda x.\, N[\varphi]_{-x_i}$.

The term M is *safe* for the substitution
$\varphi = [x_1 := N_1, \ldots, x_n := N_n]$ if
$BV(M) \cap (FV(N_1) \cup \cdots \cup FV(N_n)) = \emptyset$, that is, if the free variables in the substitution *do not* become bound.

Note that Clause (5) ensures that a substitution *only substitutes the terms N_i for the variables x_i free in M*. Thus if M is a *closed* term, then for *every* substitution φ, we have $M[\varphi] = M$.

There is a problem with the present definition of a substitution in Cases (4) and (5), which is that the result of substituting a term N_i containing the variable x free causes this variable to become bound after the substitution.

We say that x is *captured*. We should only apply a substitution φ to a term M if M is safe for φ.

To remedy this problem, Church defined *α-conversion*.

Definition 4.6. The binary relation $\longrightarrow_\alpha$ on λ-terms called *immediate α-conversion*[2] is the smallest relation satisfying the following properties: for all λ-terms M, N, P, Q:

$$\lambda x.\, M \longrightarrow_\alpha \lambda y.\, M[x := y], \text{for all } y \notin FV(M) \cup BV(M)$$

$$\text{if} \quad M \longrightarrow_\alpha N \text{ then } MQ \longrightarrow_\alpha NQ \text{ and } PM \longrightarrow_\alpha PN$$
$$\text{if} \quad M \longrightarrow_\alpha N \text{ then } \lambda x.\, M \longrightarrow_\alpha \lambda x.\, N.$$

The least equivalence relation

$$\equiv_\alpha = (\longrightarrow_\alpha \cup \longrightarrow_\alpha^{-1})^*$$

containing $\longrightarrow_\alpha$ (the reflexive and transitive closure of $\longrightarrow_\alpha \cup \longrightarrow_\alpha^{-1}$) is called *$\alpha$-conversion*.

Here $\longrightarrow_\alpha^{-1}$ denotes the converse of the relation $\longrightarrow_\alpha$, that is, $M \longrightarrow_\alpha^{-1} N$ iff $N \longrightarrow_\alpha M$.

[2] We told you that Church was fond of Greek letters.

For example,

$$\begin{aligned}\lambda f x.\, f(f(x)) = \lambda f.\, \lambda x.\, f(f(x)) &\longrightarrow_\alpha \lambda f.\, \lambda y.\, f(f(y)) \\ &\longrightarrow_\alpha \lambda g.\, \lambda y.\, g(g(y)) \\ &= \lambda g y.\, g(g(y)).\end{aligned}$$

Now given a λ-term M and a substitution $\varphi = [x_1 := N_1, \ldots, x_n := N_n]$, before applying φ to M we *first perform some α-conversion* to obtain a term $M' \equiv_\alpha M$ whose set of bound variables $BV(M')$ is disjoint from $FV(N_1) \cup \cdots \cup FV(N_n)$ so that *M' is safe for φ*, and the result of the substitution is $M'[\varphi]$.

For example,

$$\begin{aligned}\big(\lambda y z.\, (xy)z\big)[x := yz] &\equiv_\alpha \big(\lambda u v.\, (xu)v\big)[x := yz] \\ &= \lambda u v.\, ((yz)u)v.\end{aligned}$$

From now on, we consider two λ-terms M and M' such that $M \equiv_\alpha M'$ as identical (to be rigorous, we deal with equivalence classes of terms with respect to α-conversion).

Even the experts are lax about α-conversion so we happily go along with them. The convention is that *bound variables are always renamed to avoid clashes* (with free or bound variables).

Note that the representation of λ-terms as trees with back-pointers also ensures that substitutions are safe. However, this requires some extra effort.

No matter what, it takes some effort to deal properly with bound variables.

4.2 β-Reduction and β-Conversion; the Church–Rosser Theorem

The computational engine of the λ-calculus is β-reduction.

Definition 4.7. The relation $\longrightarrow_\beta$, called *immediate β-reduction*, is the smallest relation satisfying the following properties for all λ-terms M, N, P, Q:

$(\lambda x.\, M)N \longrightarrow_\beta M[x := N]$, where M is safe for $[x := N]$

if $M \longrightarrow_\beta N$ then $MQ \longrightarrow_\beta NQ$ and $PM \longrightarrow_\beta PN$
if $M \longrightarrow_\beta N$ then $\lambda x.\, M \longrightarrow_\beta \lambda x.\, N$.

The transitive closure of $\longrightarrow_\beta$ is denoted by $\overset{+}{\longrightarrow}_\beta$, the reflexive and transitive closure of $\longrightarrow_\beta$ is denoted by $\overset{*}{\longrightarrow}_\beta$, and we define *$\beta$-conversion*, denoted by $\overset{*}{\longleftrightarrow}_\beta$, as the smallest equivalence relation

$$\overset{*}{\longleftrightarrow}_\beta = (\longrightarrow_\beta \cup \longrightarrow_\beta^{-1})^*$$

containing $\longrightarrow_\beta$.

A subterm of the form $(\lambda x.\, M)N$ occurring in another term is called a *β-redex*.

A λ-term M is a *β-normal form* if there is no λ-term N such that $M \longrightarrow_\beta N$, equivalently if M contains *no* β-redex.

For example,

$$\begin{aligned}(\lambda xy.\, x)uv = \big((\lambda x.\, (\lambda y.\, x)u\big)v &\longrightarrow_\beta ((\lambda y.\, x)[x := u])v \\ &= (\lambda y.\, u)v \longrightarrow_\beta u[y := v] \\ &= u\end{aligned}$$

and

$$\begin{aligned}(\lambda xy.\, y)uv = \big((\lambda x.\, (\lambda y.\, y)u\big)v &\longrightarrow_\beta ((\lambda y.\, y)[x := u])v \\ &= (\lambda y.\, y)v \longrightarrow_\beta y[y := v] \\ &= v.\end{aligned}$$

This shows that $\lambda xy.\, x$ behaves like the projection onto the first argument and $\lambda xy.\, y$ behaves like the projection onto the second.

More interestingly, if we let

$$\boldsymbol{\omega} = \lambda x.\,(xx),$$

then

$$\begin{aligned}\boldsymbol{\Omega} = \boldsymbol{\omega}\boldsymbol{\omega} = (\lambda x.\,(xx))(\lambda x.\,(xx)) &\longrightarrow_\beta (xx)[x := \lambda x.\,(xx)] \\ &= \boldsymbol{\omega}\boldsymbol{\omega} = \boldsymbol{\Omega}.\end{aligned}$$

The above example shows that β-reduction sequences may be infinite. This is a curse and a miracle of the λ-calculus!

There are even β-reductions where the evolving term grows in size:

$$\begin{aligned}(\lambda x.\,xxx)(\lambda x.\,xxx) &\xrightarrow{+}_\beta (\lambda x.\,xxx)(\lambda x.\,xxx)(\lambda x.\,xxx) \\ &\xrightarrow{+}_\beta (\lambda x.\,xxx)(\lambda x.\,xxx)(\lambda x.\,xxx)(\lambda x.\,xxx) \\ &\qquad\qquad\qquad\qquad\qquad\qquad \xrightarrow{+}_\beta \cdots\end{aligned}$$

In general, a λ-term contains many different β-redex.

One then might wonder if there is any sort of relationship between any two terms M_1 and M_2 arising through two β-reduction sequences $M \overset{*}{\longrightarrow}_\beta M_1$ and $M \overset{*}{\longrightarrow}_\beta M_2$ starting with the same term M.

The answer is given by the following famous theorem.

Theorem 4.1. *(Church–Rosser Theorem) The following two properties hold:*

(1) The λ-calculus is **confluent***: for any three λ-terms M, M_1, M_2, if $M \overset{*}{\longrightarrow}_\beta M_1$ and $M \overset{*}{\longrightarrow}_\beta M_2$, then there is some λ-term M_3 such that $M_1 \overset{*}{\longrightarrow}_\beta M_3$ and $M_2 \overset{*}{\longrightarrow}_\beta M_3$. See Figure 4.3.*

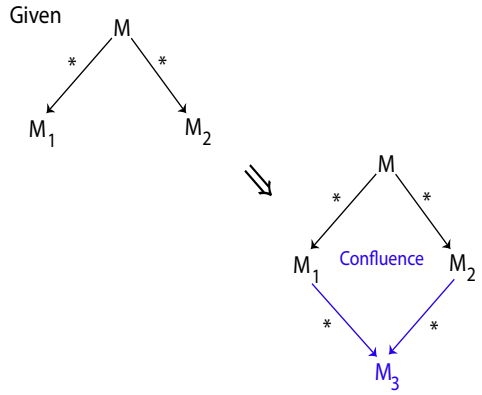

Figure 4.3: The confluence property

.

(2) The λ-calculus has the **Church–Rosser property***: for any two λ-terms M_1, M_2, if $M_1 \overset{*}{\longleftrightarrow}_\beta M_2$, then there is some λ-term M_3 such that $M_1 \overset{*}{\longrightarrow}_\beta M_3$ and $M_2 \overset{*}{\longrightarrow}_\beta M_3$. See Figure 4.4.*

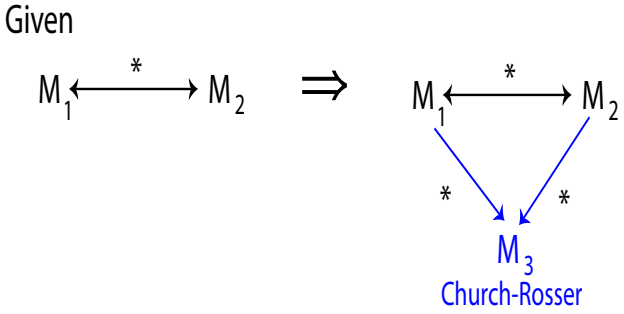

Figure 4.4: The Church–Rosser property.

Furthermore (1) and (2) are equivalent, and if a λ-term M β-reduces to a β-normal form N, then N is unique (up to α-conversion).

Another immediate corollary of the Church–Rosser theorem is that if $M \overset{*}{\longleftrightarrow}_\beta N$ and if N is a β-normal form, then in fact $M \overset{*}{\longrightarrow}_\beta N$. We leave this fact as an exerise

This fact will be useful in showing that the recursive functions are computable in the λ-calculus.

4.3 Some Useful Combinators

In this section we provide some evidence for the expressive power of the λ-calculus.

First we make a remark about the representation of functions of several variables in the λ-calculus.

The λ-calculus makes the implicit assumption that a function has a single argument.

This is the idea behind application: given a term M viewed as a function and an argument N, the term $(M\,N)$ represents the result of applying M to the argument N, *except that the actual evaluation is suspended*.

Evaluation is performed by β-conversion. To deal with functions of several arguments we use a method known as *Currying* (after Haskell Curry).

In this method, a function of n arguments is viewed as a function of one argument *taking a function of $n-1$ arguments as argument*.

Consider the case of two arguments, the general case being similar.

Consider a function $f\colon \mathbb{N}\times\mathbb{N}\to\mathbb{N}$. For any fixed x, we define the function $F_x\colon \mathbb{N}\to\mathbb{N}$ given by

$$F_x(y) = f(x,y) \qquad y\in\mathbb{N}.$$

Using the λ-notation we can write

$$F_x = \lambda y.\, f(x,y),$$

and then the function $x\mapsto F_x$, which is a function from $\mathbb{N}$ to the *set of functions* $[\mathbb{N}\to\mathbb{N}]$ (also denoted $\mathbb{N}^{\mathbb{N}}$), is denoted by the λ-term

$$F = \lambda x.\, F_x = \lambda x.\,(\lambda y.\, f(x,y)).$$

And indeed,

$$(FM)N \xrightarrow{+}_{\beta} F_M N \xrightarrow{+}_{\beta} f(M,N).$$

Remark: Currying is a way to realizing the isomorphism between the sets of functions
$[\mathbb{N} \times \mathbb{N} \to \mathbb{N}]$ and $[\mathbb{N} \to [\mathbb{N} \to \mathbb{N}]]$ (or in the standard set-theoretic notation, between $\mathbb{N}^{\mathbb{N}\times\mathbb{N}}$ and $(\mathbb{N}^{\mathbb{N}})^{\mathbb{N}}$.

Does this remind you of the identity

$$(m^n)^p = m^{n*p}?$$

It should.

The function space $[\mathbb{N} \to \mathbb{N}]$ is called an *exponential*.

Proposition 4.2. *If* $\mathbf{I}, \mathbf{K}, \mathbf{K}_*$, *and* $\mathbf{S}$ *are the combinators defined by*

$$\begin{aligned}\mathbf{I} &= \lambda x.\, x\\ \mathbf{K} &= \lambda xy.\, x\\ \mathbf{K}_* &= \lambda xy.\, y\\ \mathbf{S} &= \lambda xyz.\, (xz)(yz),\end{aligned}$$

then for all λ*-terms* M, N, P, *we have*

$$\begin{aligned}\mathbf{I}M &\xrightarrow{+}_\beta M\\ \mathbf{K}MN &\xrightarrow{+}_\beta M\\ \mathbf{K}_*MN &\xrightarrow{+}_\beta N\\ \mathbf{S}MNP &\xrightarrow{+}_\beta (MP)(NP)\\ \mathbf{KI} &\xrightarrow{+}_\beta \mathbf{K}_*\\ \mathbf{SKK} &\xrightarrow{+}_\beta \mathbf{I}.\end{aligned}$$

For example,

$$\begin{aligned}\mathbf{S}MNP = \big(\lambda xyz.\,(xz)(yz)\big)MNP &\longrightarrow_\beta \\ \big(\big(\lambda yz.\,(xz)(yz)\big)[x := M]\big)NP & \\ = \big(\lambda yz.\,(Mz)(yz)\big)NP &\longrightarrow_\beta \\ \big(\big(\lambda z.\,(Mz)(yz)\big)[y := N]\big)P & \\ = \big(\lambda z.\,(Mz)(Nz)\big)\big)P &\longrightarrow_\beta \\ \big((Mz)(Nz)\big)[z := P] = (MP)(NP). &\end{aligned}$$

The need for a conditional construct $\mathsf{if\ then\ else}$ such that $\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ yields P and $\mathsf{if}\ \mathbf{F}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ yields Q is indispensable to write nontrivial programs.

There is a trick to encode the boolean values $\mathbf{T}$ and $\mathbf{F}$ in the λ-calculus to mimick the above behavior of $\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q$, provided that B is a truth value.

Since everything in the λ-calculus is a function, the booleans values $\mathbf{T}$ and $\mathbf{F}$ are encoded as λ-terms.

At first, this seems quite odd, but what counts is the behavior of $\mathsf{if}\ \mathbf{B}\ \mathsf{then}\ P\ \mathsf{else}\ Q$, and it works!

The truth values $\mathbf{T}, \mathbf{F}$ and the conditional construct $\mathsf{if}\ B\ \mathsf{then}\ P\ \mathsf{else}\ Q$ can be encoded in the λ-calculus as follows.

Proposition 4.3. *Consider the combinators given by* $\mathbf{T} = \mathbf{K}, \mathbf{F} = \mathbf{K}_*$, *and*

$$\mathsf{if\ then\ else}\ = \lambda bxy.\, bxy.$$

Then for all λ-terms we have

$$\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{+}_\beta P$$
$$\mathsf{if}\ \mathbf{F}\ \mathsf{then}\ P\ \mathsf{else}\ Q \xrightarrow{+}_\beta Q.$$

The boolean operations $\wedge, \vee, \neg$ can be defined in terms of $\mathsf{if\ then\ else}$.

For example,

$$\mathsf{And}\ b_1 b_2 = \mathsf{if}\ b_1 \mathsf{then}\ (\mathsf{if}\ b_2\ \mathsf{then}\ \mathbf{T}\ \mathsf{else}\ \mathbf{F})\ \mathsf{else}\ \mathbf{F}.$$

Remark: If B is a term different from $\mathbf{T}$ or $\mathbf{F}$, then $\mathsf{if}\ \mathbf{B}\ \mathsf{then}\ P\ \mathsf{else}\ Q$ may not reduce at all, or reduce to something different from P or Q.

The problem is that the conditional statement that we designed only works properly if the input B is of the correct type, namely a boolean.

If we give garbage as input, then we can't expect a correct result.

The λ-calculus being type-free, it is unable to check for the validity of the input. In this sense this is a defect, but it also accounts for its power.

The ability to construct ordered pairs is also crucial.

Proposition 4.4. *For any two λ-terms M and N consider the combinator $\langle M, N\rangle$ and the combinator π_1 and π_2 given by*

$$\begin{aligned}\langle M, N\rangle &= \lambda z.\, zMN = \lambda z.\, \mathsf{if}\ z\ \mathsf{then}\ M\ \mathsf{else}\ N\\ \pi_1 &= \lambda z.\, z\mathbf{K}\\ \pi_2 &= \lambda z.\, z\mathbf{K}_*.\end{aligned}$$

Then

$$\begin{aligned}\pi_1\langle M, N\rangle &\xrightarrow{+}_\beta M\\ \pi_2\langle M, N\rangle &\xrightarrow{+}_\beta N\\ \langle M, N\rangle\mathbf{T} &\xrightarrow{+}_\beta M\\ \langle M, N\rangle\mathbf{F} &\xrightarrow{+}_\beta N.\end{aligned}$$

For example,

$$\begin{aligned}\pi_1\langle M, N\rangle &= \big(\lambda z.\, z\mathbf{K}\big)\big(\lambda z.\, zMN\big)\\ &\longrightarrow_\beta \big(z\mathbf{K}\big)[z := \lambda z.\, zMN] = \big(\lambda z.\, zMN\big)\mathbf{K}\\ &\longrightarrow_\beta (zMN)[z := \mathbf{K}] = \mathbf{K}MN \xrightarrow{+}_\beta M,\end{aligned}$$

by Proposition 4.2.

4.4 Representing the Natural Numbers

Historically the natural numbers were first represented in the λ-calculus by Church in the 1930's.

Later in 1976 Barendregt came up with another representation which is more convenient to show that the recursive functions are λ-definable. We start with Church's representation.

First, given any two λ-terms F and M, for any natural number $n \in \mathbb{N}$, we define $F^n(M)$ inductively as follows:

$$F^0(M) = M$$
$$F^{n+1}(M) = F(F^n(M)).$$

Definition 4.8. (Church Numerals) The *Church numerals* $\mathbf{c}_0, \mathbf{c}_1, \mathbf{c}_2, \ldots$ are defined by

$$\mathbf{c}_n = \lambda f x.\, f^n(x).$$

$\mathbf{c}_0 = \lambda f x.\, x = \mathbf{K}_*$,
$\mathbf{c}_1 = \lambda f x.\, f x$,
$\mathbf{c}_2 = \lambda f x.\, f(f x)$, *etc.*

The Church numerals are β-normal forms.

Observe that

$$\mathbf{c}_n F z = (\lambda f x.\, f^n(x)) F z \xrightarrow{+}_\beta F^n(z). \qquad (\dagger)$$

This shows that $\mathbf{c}_n$ *iterates* n *times the function represented by the term* F *on initial input* z.

This is the trick behind the definition of the Church numerals.

Definition 4.9. The *iteration combinator* **Iter** is given by

$$\mathbf{Iter} = \lambda n f x.\, n f x.$$

Observe that

$$\mathbf{Iter}\, \mathbf{c}_n\, F\, X \xrightarrow{+}_\beta F^n X,$$

that is, the result of iterating F for n steps starting with the initial term X.

Remark: The combinator **Iter** is actually equal to the combinator

$$\mathsf{if\ then\ else} = \lambda bxy.\, bxy$$

of Definition 4.3.

Remarkably, if n (or b) is a boolean, then this combinator behaves like a conditional, but if n (or b) is a Church numeral, then it behaves like an iterator.

A closely related combinator is **Fold**, defined by

$$\mathbf{Fold} = \lambda xfn.\, nxf.$$

The only difference is that the abstracted variables are listed in the order x, f, n, instead of n, f, x.

This version of an iterator is used when the Church numerals are defined as $\lambda xf.\, f^n(x)$ instead of $\lambda fx.\, f^n(x)$, where x and f are permuted in the λ-binder.

Let us show how some basic functions on the natural numbers can be defined.

We begin with the constant function $\mathbf{Z}$ given by $\mathbf{Z}(n) = 0$ for all $n \in \mathbb{N}$.

We claim that $\mathbf{Z_c} = \lambda x.\, \mathbf{c_0}$ works.

Indeed, we have

$$\mathbf{Z_c}\, \mathbf{c}_n = (\lambda x.\, \mathbf{c_0})\mathbf{c}_n \longrightarrow_\beta \mathbf{c_0}[x := \mathbf{c}_n] = \mathbf{c_0}$$

since $\mathbf{c_0}$ is a closed term.

The successor function **Succ** is given by

$$\mathbf{Succ}(n) = n + 1.$$

We claim that

$$\mathbf{Succ_c} = \lambda n f x.\, f(nfx)$$

computes **Succ**.

Indeed we have

$$\begin{aligned}\mathbf{Succ_c}\ \mathbf{c}_n &= (\lambda nfx.\, f(nfx))\mathbf{c}_n \\ &\longrightarrow_\beta \big(\lambda fx.\, f(nfx)\big)[n := \mathbf{c}_n] = \lambda fx.\, f(\mathbf{c}_n fx) \\ &\longrightarrow_\beta \lambda fx.\, f(f^n(x)) \\ &= \lambda fx.\, f^{n+1}(x) = \mathbf{c}_{n+1}.\end{aligned}$$

The function **IsZero** which tests whether a natural number is equal to 0 is defined by the combinator

$$\mathbf{IsZero_c} = \lambda x.\, x(\mathbf{K}\,\mathbf{F})\mathbf{T}.$$

Addition and multiplication are a little more tricky to define.

Proposition 4.5. *(J.B. Rosser) Define* **Add** *and* **Mult** *as the combinators given by*

$$\begin{aligned}\mathbf{Add} &= \lambda mnfx.\, mf(nfx)\\ \mathbf{Mult} &= \lambda xyz.\, x(yz).\end{aligned}$$

We have

$$\begin{aligned}\mathbf{Add}\,\mathbf{c}_m\mathbf{c}_n &\xrightarrow{+}_\beta \mathbf{c}_{m+n}\\ \mathbf{Mult}\,\mathbf{c}_m\mathbf{c}_n &\xrightarrow{+}_\beta \mathbf{c}_{m*n}\end{aligned}$$

for all $m, n \in \mathbb{N}$.

Proof. We have

$$\begin{aligned}\mathbf{Add}\,\mathbf{c}_m\mathbf{c}_n &= (\lambda mnfx.\, mf(nfx))\mathbf{c}_m\mathbf{c}_n\\ &\xrightarrow{+}_\beta (\lambda fx.\, \mathbf{c}_m f(\mathbf{c}_n fx))\\ &\xrightarrow{+}_\beta \lambda fx.\, f^m(f^n(x))\\ &= \lambda fx.\, f^{m+n}(x) = \mathbf{c}_{m+n}.\end{aligned}$$

For multiplication we need to prove by induction on m that

$$(\mathbf{c}_n x)^m(y) \xrightarrow{*}_\beta x^{m*n}(y). \qquad (*)$$

□

As an exercise the reader should prove that addition and multiplication can also be defined in terms of **Iter** (see Definition 4.9) by

$$\begin{aligned}\mathbf{Add} &= \lambda mn.\, \mathbf{Iter}\, m\, \mathbf{Succ_c}\, n\\ \mathbf{Mult} &= \lambda mn.\, \mathbf{Iter}\, m\, (\mathbf{Add}\, n)\, \mathbf{c}_0.\end{aligned}$$

The above expressions are close matches to the primitive recursive definitions of addition and multiplication.

A function that plays an important technical role is the predecessor function **Pred** defined such that

$$\begin{aligned}\mathbf{Pred}(0) &= 0\\ \mathbf{Pred}(n+1) &= n.\end{aligned}$$

It turns out that it is quite tricky to define this function in terms of the Church numerals.

Church and his students struggled for a while until Kleene found a solution in his famous 1936 paper.

The story goes that Kleene found his solution when he was sittting in the dentist's chair!

The trick is to make use of pairs. Kleene's solution is

$$\mathbf{Pred_K} = \lambda n.\, \pi_2(\mathbf{Iter}\, n\, \lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle\, \langle \mathbf{c}_0, \mathbf{c}_0\rangle).$$

The reason this works is that we can prove that

$$(\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^0 \langle \mathbf{c}_0, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_0, \mathbf{c}_0\rangle,$$

and by induction that

$$(\lambda z.\, \langle \mathbf{Succ_c}(\pi_1 z), \pi_1 z\rangle)^{n+1} \langle \mathbf{c}_0, \mathbf{c}_0\rangle \xrightarrow{+}_\beta \langle \mathbf{c}_{n+1}, \mathbf{c}_n\rangle.$$

Here is another tricky solution due to J. Velmans (according to H. Barendregt):

$$\mathbf{Pred_c} = \lambda xyz.\, x(\lambda pq.\, q(py))(\mathbf{K}z)\mathbf{I}.$$

The ability to construct pairs together with the **Iter** combinator allows the definition of a large class of functions, because **Iter** is "type-free" in its second and third arguments so it really allows higher-order primitive recursion.

For example, the factorial function defined such that

$$\begin{aligned} 0! &= 1 \\ (n+1)! &= (n+1)n! \end{aligned}$$

can be defined.

First we define h by

$$h = \lambda xn.\, \mathbf{Mult}\, \mathbf{Succ_c} n\, x$$

and then

$$\begin{aligned}\mathbf{fact} = \lambda n.\, \pi_1(\mathbf{Iter}\, n\, \lambda z.\, \langle h(\pi_1 z)\, (\pi_2 z),\\ \mathbf{Succ_c}(\pi_2 z)\rangle\, \langle \mathbf{c}_1, \mathbf{c}_0\rangle).\end{aligned}$$

Barendregt came up with another way of representing the natural numbers that makes things easier.

Definition 4.10. (Barendregt Numerals) The *Barendregt numerals* $\mathbf{b}_n$ are defined as follows:

$$\begin{aligned} \mathbf{b}_0 &= \mathbf{I} = \lambda x.\, x \\ \mathbf{b}_{n+1} &= \langle \mathbf{F}, \mathbf{b}_n \rangle. \end{aligned}$$

The Barendregt numerals are β-normal forms.

The Barendregt numerals are tuples, which makes operating on them simpler than the Church numerals which encode n as the composition f^n.

Proposition 4.6. *The functions* **Succ**, **Pred** *and* **IsZero** *are defined in terms of the Barendregt numerals by the combinators*

$$\begin{aligned}\mathbf{Succ_b} &= \lambda x.\, \langle \mathbf{F}, x\rangle\\ \mathbf{Pred_b} &= \lambda x.\, (x\mathbf{F})\\ \mathbf{IsZero_b} &= \lambda x.\, (x\mathbf{T}),\end{aligned}$$

and we have

$$\begin{aligned}\mathbf{Succ_b}\,\mathbf{b}_n &\xrightarrow{+}_\beta \mathbf{b}_{n+1}\\ \mathbf{Pred_b}\,\mathbf{b}_0 &\xrightarrow{+}_\beta \mathbf{b}_0\\ \mathbf{Pred_b}\,\mathbf{b}_{n+1} &\xrightarrow{+}_\beta \mathbf{b}_n\\ \mathbf{IsZero_b}\,\mathbf{b}_0 &\xrightarrow{+}_\beta \mathbf{T}\\ \mathbf{IsZero_b}\,\mathbf{b}_{n+1} &\xrightarrow{+}_\beta \mathbf{F}.\end{aligned}$$

Since there is an obvious bijection between the Church combinators and the Barendregt combinators there should be combinators effecting the translations.

Proposition 4.7. *The combinator T given by*

$$T = \lambda x.\, (x\mathbf{Succ_b})\mathbf{b}_0$$

has the property that

$$T\,\mathbf{c}_n \xrightarrow{+}_\beta \mathbf{b}_n \qquad \textit{for all } n \in \mathbb{N}.$$

There is also a combinator defining the inverse map but it is defined recursively and we don't know how to express recursive definitions in the λ-calculus.

This is achieved by using fixed-point combinators.

4.5 Fixed-Point Combinators and Recursively Defined Functions

Fixed-point combinators are the key to the definability of recursive functions in the λ-calculus. We begin with the $\mathbf{Y}$-combinator due to Curry.

Proposition 4.8. *(Curry $\mathbf{Y}$-combinator) If we define the combinator $\mathbf{Y}$ as*

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

then for any λ-term F we have

$$F(\mathbf{Y}F) \overset{*}{\longleftrightarrow}_\beta \mathbf{Y}F.$$

Observe that $F(\mathbf{Y}F)$ and $\mathbf{Y}F$ both β-reduce to $F(WW)$, where $W = \lambda x.\, F(xx)$, but neither $F(\mathbf{Y}F) \overset{+}{\longrightarrow}_\beta \mathbf{Y}F$ nor $\mathbf{Y}F \overset{+}{\longrightarrow}_\beta F(\mathbf{Y}F)$.

This is a slight disadvantage of the Curry $\mathbf{Y}$-combinator.

Turing came up with another fixed-point combinator that does not have this problem.

Proposition 4.9. *(Turing $\boldsymbol{\Theta}$-combinator) If we define the combinator $\boldsymbol{\Theta}$ as*

$$\boldsymbol{\Theta} = (\lambda xy.\, y(xxy))(\lambda xy.\, y(xxy)),$$

then for any λ-term F we have

$$\boldsymbol{\Theta} F \xrightarrow{+}_{\beta} F(\boldsymbol{\Theta} F).$$

Now we show how to use the fixed-point combinators to represent recursively-defined functions in the λ-calculus.

For example, there is a combinator G such that

$$GX \xrightarrow{+}_{\beta} X(XG) \quad \text{for all } X.$$

Informally the idea is to consider the "functional" $F = \lambda gx.\, x(xg)$, and to find a fixed-point of this functional.

Pick

$$G = \mathbf{\Theta}\lambda gx.\, x(xg) = \mathbf{\Theta}F.$$

Since by Proposition 4.9 we have $G = \mathbf{\Theta}F \xrightarrow{+}_\beta F(\mathbf{\Theta}F) = FG$, and we also have

$$FG = (\lambda gx.\, x(xg))G \longrightarrow_\beta \lambda x.\, x(xG),$$

so

$G \xrightarrow{+}_\beta \lambda x.\, x(xG)$, which implies

$$GX \xrightarrow{+}_\beta (\lambda x.\, x(xG))X \longrightarrow_\beta X(XG).$$

In general, if we want to define a function G recursively such that

$$GX \xrightarrow{+}_\beta M(X, G)$$

where $M(X, G)$ is λ-term containing recursive occurrences of G, we let $F = \lambda gx.\, M(x, g)$ and

$$G = \mathbf{\Theta} F.$$

Then we have

$$\begin{aligned} G \xrightarrow{+}_\beta FG = (\lambda gx.\, M(x, g))G &\longrightarrow_\beta \lambda x.\, M(x, g)[g := G] \\ &= \lambda x.\, M(x, G), \end{aligned}$$

so

$$\begin{aligned} GX \xrightarrow{+}_\beta (\lambda x.\, M(x, G))X &\longrightarrow_\beta M(x, G)[x := X] \\ &= M(X, G), \end{aligned}$$

as desired.

As another example, here is how the factorial function can be defined (using the Church numerals). Let

$$F = \lambda gn.\, \mathsf{if}\ \mathbf{IsZero_c}\, n\ \mathsf{then}\ \mathbf{c}_1\ \mathsf{else}\ \mathbf{Mult}\, n\, g(\mathbf{Pred_c}\, n).$$

Then the term $G = \mathbf{\Theta} F$ defines the factorial function. The verification of the above fact is left as an exercise.

As usual with recursive definitions there is no guarantee that the function that we obtain terminates for all input.

For example, if we consider

$$F = \lambda gn.\, \mathsf{if}\ \mathbf{IsZero_c}\, n\ \mathsf{then}\ \mathbf{c}_1\ \mathsf{else}\ \mathbf{Mult}\, n\, g(\mathbf{Succ_c}\, n)$$

then for $n \geq 1$ the reduction behavior is

$$G\mathbf{c}_n \xrightarrow{+}_\beta \mathbf{Mult}\, \mathbf{c}_n\, G\, \mathbf{c}_{n+1},$$

which does not terminate.

We leave it as an exercise to show that the inverse of the function T mapping the Church numerals to the Barendregt numerals is given by the combinator

$$\mathbf{\Theta}(\lambda f x.\ \mathsf{if}\ \mathbf{IsZero_b}\, x\ \mathsf{then}\ \mathbf{c}_0\ \mathbf{else}\ \mathbf{Succ_c}(f(\mathbf{Pred_b}\, x)).$$

It is remarkable that the λ-calculus allows the implementation of *arbitrary recursion* without a stack, just using λ-terms as the data-structure and β-reduction.

This does not mean that this evaluation mechanism is efficient but this is another story (as well as evaluation strategies, which have to do with parameter-passing strategies, call-by-name, call-by-value).

Now we have all the ingredients to show that all the (total) computable functions are definable in the λ-calculus.

4.6 λ-Definability of the Computable Functions

We begin by reviewing the definition of the computable functions (recursive functions)
(à la Herbrand–Gödel–Kleene).

For our purposes it suffices to consider functions (partial or total) $f\colon \mathbb{N}^n \to \mathbb{N}$ as opposed to the more general case of functions $f\colon (\Sigma^*)^n \to \Sigma^*$ defined on strings.

Definition 4.11. The *base functions* are the functions Z, S, P_i^n defined as follows:

(1) The constant *zero function* Z such that

$$Z(n) = 0, \qquad \text{for all } n \in \mathbb{N}.$$

(2) The *successor function* S such that

$$S(n) = n + 1, \qquad \text{for all } n \in \mathbb{N}.$$

(3) For every $n \geq 1$ and every i with $1 \leq i \leq n$, the *projection function* P_i^n such that

$$P_i^n(x_1, \ldots, x_n) = x_i, \qquad x_1, \ldots, x_n \in \mathbb{N}.$$

Next comes (extended) composition.

Definition 4.12. Given any partial or total function $g\colon \mathbb{N}^m \to \mathbb{N}$ $(m \geq 1)$ and any m partial or total functions $h_i\colon \mathbb{N}^n \to \mathbb{N}$ $(n \geq 1)$, the *composition of g and $h_1, \ldots, h_m$*, denoted $g \circ (h_1, \ldots, h_m)$, is the partial or total function function $f\colon \mathbb{N}^n \to \mathbb{N}$ given by

$$f(x_1, \ldots, x_n) = g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n)),$$
$$x_1, \ldots, x_n \in \mathbb{N}.$$

If g or any of the h_i are partial functions, then $f(x_1, \ldots, x_n)$ is defined if and only if *all $h_i(x_1, \ldots, x_n)$ are defined and $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ is defined*.

Note that even if g "ignores" one of its arguments, say the ith one, $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ is undefined if $h_i(x_1, \ldots, x_n)$ is undefined.

Definition 4.13. Given any partial or total functions $g\colon \mathbb{N}^m \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+2} \to \mathbb{N}$ $(m \geq 1)$, the partial or total function function $f\colon \mathbb{N}^{m+1} \to \mathbb{N}$ is defined by *primitive recursion* from g and h if f is given by:

$$\begin{aligned} f(0, x_1, \ldots, x_m) &= g(x_1, \ldots, x_m) \\ f(n+1, x_1, \ldots, x_m) &= h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) \end{aligned}$$

for all $n, x_1, \ldots, x_m \in \mathbb{N}$. If $m = 0$, then g is some fixed natural number and we have

$$\begin{aligned} f(0) &= g \\ f(n+1) &= h(f(n), n). \end{aligned}$$

It can be shown that if g and h are total functions, then so if f.

Note that the second clause of the definition of primitive recursion is

$$f(n{+}1, x_1, \ldots, x_m) = h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) \qquad (*_1)$$

but in an earlier definition it was

$$f(n{+}1, x_1, \ldots, x_m) = h(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m), \qquad (*_2)$$

with the first two arguments of h permuted.

Since

$$h{\circ}(P_2^{m+2}, P_1^{m+2}, P_3^{m+2}, \ldots, P_{m+2}^{m+2})(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m) = h(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m)$$

and

$$h{\circ}(P_2^{m+2}, P_1^{m+2}, P_3^{m+2}, \ldots, P_{m+2}^{m+2})(f(n, x_1, \ldots, x_m), n, x_1, \ldots, x_m) = h(n, f(n, x_1, \ldots, x_m), x_1, \ldots, x_m),$$

the two definitions are equivalent.

In this section we chose version $(*_1)$ because it matches the treatment in Barendregt [**?**] and will make it easier for the reader to follow Barendregt [**?**] if they wish.

The last operation is *minimization* (sometimes called minimalization).

Definition 4.14. Given any partial or total function $g\colon \mathbb{N}^{m+1} \to \mathbb{N}$ $(m \geq 0)$, the partial or total function function $f\colon \mathbb{N}^m \to \mathbb{N}$ is defined as follows: for all $x_1, \ldots, x_m \in \mathbb{N}$,

$$f(x_1, \ldots, x_m) = \text{the least } n \in \mathbb{N} \text{ such that } g(n, x_1, \ldots, x_m) = 0,$$

and undefined if there is no n such that $g(n, x_1, \ldots, x_m) = 0$.

We say that f is *defined by minimization from* g, and we write

$$f(x_1, \ldots, x_m) = \mu x[g(x, x_1, \ldots, x_m) = 0].$$

For short, we write $f = \mu g$.

Even if g is a total function, f may be undefined for some (or all) of its inputs.

Definition 4.15. (Herbrand–Gödel–Kleene) The set of *partial computable* (or *partial recursive*) functions is the smallest set of partial functions (defined on $\mathbb{N}^n$ for some $n \geq 1$) which contains the base functions and is closed under

(1) Composition.

(2) Primitive recursion.

(3) Minimization.

The set of *computable* (or *recursive*) functions is the subset of partial computable functions that are total functions (that is, defined for all input).

We proved earlier the Kleene normal form, which says that *every* partial computable function $f\colon \mathbb{N}^m \to \mathbb{N}$ is computable as

$$f = g \circ \mu h,$$

for some *primitive recursive functions* $g\colon \mathbb{N} \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+1} \to \mathbb{N}$.

The significance of this result is that f is built up from *total functions* using composition and primitive recursion, and only a single minimization is needed at the end.

Before stating our main theorem, we need to define what it means for a (numerical) function to be definable in the λ-calculus. This requires some care to handle partial functions.

Since there are combinators for translating Church numerals to Barendregt numerals and vice-versa, it does not matter which numerals we pick.

We pick the Church numerals because primitive recursion is definable without using a fixed-point combinator.

Definition 4.16. A function (partial or total) $f : \mathbb{N}^n \to \mathbb{N}$ is *λ-definable* if for all $m_1, \ldots, m_n \in \mathbb{N}$, there is a combinator (a closed λ-term) F with the following properties:

(1) The value $f(m_1, \ldots, m_n)$ is defined if and only if $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to a β-normal form (necessarily unique by the Church–Rosser theorem).

(2) If $f(m_1, \ldots, m_n)$ is defined, then

$$F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n} \overset{*}{\longleftrightarrow}_\beta \mathbf{c}_{f(m_1,\ldots,m_n)}.$$

In view of the Church–Rosser theorem (Theorem 4.1) and the fact that $\mathbf{c}_{f(m_1,\ldots,m_n)}$ is a β-normal form, we can replace

$$F\mathbf{c}_{m_1}\cdots\mathbf{c}_{m_n} \overset{*}{\longleftrightarrow}_\beta \mathbf{c}_{f(m_1,\ldots,m_n)}$$

by

$$F\mathbf{c}_{m_1}\cdots\mathbf{c}_{m_n} \overset{*}{\longrightarrow}_\beta \mathbf{c}_{f(m_1,\ldots,m_n)}$$

Note that the termination behavior of f on inputs $m_1, \ldots, m_n$ has to *match* the reduction behavior of $F\mathbf{c}_{m_1}\cdots\mathbf{c}_{m_n}$, namely $f(m_1,\ldots,m_n)$ is undefined if *no* reduction sequence from $F\mathbf{c}_{m_1}\cdots\mathbf{c}_{m_n}$ reaches a β-normal form.

Condition (2) ensures that if $f(m_1,\ldots,m_n)$ is defined, then the correct value $\mathbf{c}_{f(m_1,\ldots,m_n)}$ is computed by some reduction sequence from $F\mathbf{c}_{m_1}\cdots\mathbf{c}_{m_n}$.

If we only care about total functions then we require that $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to a β-normal for *all* $m_1, \ldots, m_n$ and (2).

A stronger and more elegant version of λ-definabilty that better captures when a function is undefined for some input is considered in Section 4.7 .

We have the following remarkable theorems.

Theorem 4.10. *If a total function $f : \mathbb{N}^n \to \mathbb{N}$ is λ-definable, then it is (total) computable. If a partial function $f : \mathbb{N}^n \to \mathbb{N}$ is λ-definable, then it is partial computable.*

Although Theorem 4.10 is intuitively obvious since computation by β-reduction sequences are "clearly" computable, a detailed proof is long and very tedious.

One has to define primitive recursive functions to mimick β-conversion, *etc*.

Most books sweep this issue under the rug.

Barendregt observes that the "λ-calculus is recursively axiomatized," which implies that the graph of the function beeing defined is recursively enumerable, but no details are provided; see Barendregt [**?**] (Chapter 6, Theorem 6.3.13).

Kleene (1936) provides a detailed and very tedious proof. This is an amazing paper, but very hard to read.

Theorem 4.11. *(Kleene, 1936) If a (total) function $f : \mathbb{N}^n \to \mathbb{N}$ is computable, then it is λ-definable. If a (partial) function $f : \mathbb{N}^n \to \mathbb{N}$ is is partial computable, then it is λ-definable.*

Proof. First we assume all functions to be total. There are several steps.

Step 1. The base functions are λ-definable.

We already showed that $\mathbf{Z_c}$ computes Z and that $\mathbf{Succ_c}$ computes S. Observe that $\mathbf{U}_i^n$ given by

$$\mathbf{U}_i^n = \lambda x_1 \cdots x_n . \, x_i$$

computes P_i^n.

Step 2. Closure under composition.

If g is λ-defined by the combinator G and $h_1, \ldots, h_m$ are λ-defined by the combinators $H_1, \ldots, H_m$, then $g \circ (h_1, \ldots, h_m)$ is λ-defined by

$$F = \lambda x_1 \cdots x_n . G(H_1 x_1 \cdots x_n) \ldots (H_m x_1 \cdots x_n).$$

Since the functions are total, there is no problem.

Step 3. Closure under primitive recursion.

We could use a fixed-point combinator but the combinator **Iter** and pairing do the job.

If f is defined by primitive recursion from g and h, and if G λ-defines g and H λ-defines h, then f is λ-defined by

$$F = \lambda n x_1 \cdots x_m.\, \pi_1\big(\mathbf{Iter}\; n\; \lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, x_1 \cdots x_m, \\ \mathbf{Succ_c}(\pi_2 z)\rangle\; \langle G x_1 \cdots x_m, \mathbf{c}_0\rangle\big).$$

The reason F works is that we can prove by induction that

$$\big(\lambda z.\, \langle H\, \pi_1 z\, \pi_2 z\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m},\; \mathbf{Succ_c}(\pi_2 z)\rangle\big)^n \\ \langle G \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_m}, \mathbf{c}_0\rangle \\ \xrightarrow{+}_\beta \langle \mathbf{c}_{f(n,n_1,\ldots,n_m)}, \mathbf{c}_n\rangle.$$

We can also show that primitive recursion can be achieved using a fixed-point combinator. Define the combinators J and F by

$$J = \lambda f x x_1 \cdots x_m.\ \mathsf{if}\ \mathbf{IsZero_c}\ x\ \mathsf{then}\ G x_1 \cdots x_m \\ \mathsf{else}\ H(f(\mathbf{Pred_c}\ x)\ x_1 \cdots x_m)(\mathbf{Pred_c}\ x)\ x_1 \cdots x_m,$$

and

$$F = \mathbf{\Theta} J.$$

Then F λ-defines f, and since the functions are total, there is no problem.

This method must be used if we use the Barendregt numerals.

Step 4. Closure under minimization.

Suppose f is total and defined by minimization from g and that g is λ-defined by G.

Define the combinators J and F by

$$J = \lambda f x x_1 \cdots x_m.\, \mathsf{if}\ \mathbf{IsZero_c}\, G x\, x_1 \cdots x_m\ \mathsf{then}\ x \\ \mathsf{else}\ f(\mathbf{Succ_c}\, x)\, x_1 \cdots x_m$$

and

$$F = \mathbf{\Theta} J.$$

It is not hard to check that

$$F\, \mathbf{c}_n\, \mathbf{c}_{n_1} \ldots \mathbf{c}_{n_n} \xrightarrow{+}_\beta \begin{cases} \mathbf{c}_n & \text{if } g(n, n_1, \ldots, n_m) = 0 \\ F\, \mathbf{c}_{n+1}\, \mathbf{c}_{n_1} \cdots \mathbf{c}_{n_n} & \text{otherwise,} \end{cases}$$

and we can use this to prove that F λ-defines f.

Since we assumed that f is total, some least n will be found. We leave the details as an exercise.

This finishes the proof that every total computable function is λ-definable.

To prove the result for the partial computable functions we appeal to the Kleene normal form: every partial computable function $f\colon \mathbb{N}^m \to \mathbb{N}$ is computable as

$$f = g \circ \mu h,$$

for some *primitive recursive functions* $g\colon \mathbb{N} \to \mathbb{N}$ and $h\colon \mathbb{N}^{m+1} \to \mathbb{N}$.

Then our previous proof yields combinators G and H that λ-define g and h.

The minimization of h may fail but since g is a total function of a single argument, $f(n_1, \ldots, n_m)$ is defined iff $g(\mu n[h(n, n_1, \ldots, n_m) = 0])$ is defined so it should be clear that F computes f, but the reader may want to provide a rigorous argument.

A detailed proof is given in Hindley and Seldin [**?**] (Chapter 4, Theorem 4.18). □

Combining Theorem 4.10 and Theorem 4.11 we have established the remarkable result that the set of λ-definable total functions is exactly the set of (total) computable functions, and similarly for partial functions.

So the λ-calculus has universal computing power.

Remark: With some work, it is possible to show that lists can be represented in the λ-calculus.

Since a Turing machine tape can be viewed as a list, it should be possible (but very tedious) to simulate a Turing machine in the λ-calculus.

This simulation should be somewhat analogous to the proof that a Turing machine computes a computable function (defined à la Herbrand–Gödel–Kleene).

Since the λ-calculus has the same power as Turing machines we should expect some undecidabity results analogous to the undecidability of the halting problem or Rice's theorem.

We state the following analog of Rice's theorem without proof. It is a corollary of a theorem known as the Scott–Curry theorem.

Theorem 4.12. *(D. Scott) Let $\mathcal{A}$ be any nonempty set of λ-terms not equal to the set of all λ-terms. If $\mathcal{A}$ is closed under β-reduction, then it is not computable (not recursive).*

Theorem 4.12 is proven in Barendregt [**?**] (Chapter 6, Theorem 6.6.2) and Barendregt [**?**].

As a corollary of Theorem 4.12 it is undecidable whether a λ-term has a β-normal form, a result originally proved by Church.

This is an analog of the undecidability of the halting problem, but it seems more spectacular because the syntax of λ-terms is really very simple.

The problem is that β-reduction is very powerful and elusive.

4.7 Definability of Functions in Typed Lambda-Calculi

This is a *supplementary optional section* that requires knowledge of the simply-typed lambda calculus.

In the pure λ-calculus, some λ-terms have no β-normal form, and worse, it is undecidable whether a λ-term has a β-normal form.

In contrast, by Theorem **??**, *every* raw λ-term that type-checks in the simply-typed λ-calculus has a β-normal form.

Thus it is natural to ask whether the natural numbers are definable in the simply-typed λ-calculus because if the answer is positive, then the numerical functions definable in the simply-typed λ-calculus are guaranteed to be total.

This indeed possible. If we pick any base type σ, then we can define typed Church numerals $\mathbf{c}_n$ as terms of type $\mathsf{Nat}_\sigma = (\sigma \to \sigma) \to (\sigma \to \sigma)$, by

$$\mathbf{c}_n = \lambda f\colon (\sigma \to \sigma).\, \lambda x\colon \sigma.\, f^n(x).$$

The notion of λ-definable function is defined just as before.

Then we can define **Add** and **Mult** as terms of type $\mathsf{Nat}_\sigma \to (\mathsf{Nat}_\sigma \to \mathsf{Nat}_\sigma)$ essentially as before, but surprise, not much more is definable.

Among other things, strong typing of terms restricts the iterator combinator too much.

It was shown by Schwichtenberg and Statman that the numerical functions definable in the simply-typed λ-calculus are the extended polynomials; see Statman [**?**] and Troelstra and Schwichtenberg [**?**].

The extended polynomials are the smallest class of numerical functions closed under composition containing

1. The constant functions 0 and 1.
2. The projections.
3. Addition and multiplication.
4. The function $\mathbf{IsZero_c}$.

Is there a way to get a larger class of total functions?

There are indeed various ways of doing this.

One method is to add the natural numbers and the booleans as data types to the simply-typed λ-calculus, and to also add product types, an iterator combinator, and some new reduction rules.

This way we obtain a system equivalent to *Gödel's system T*. A large class of numerical total functions containing the primitive recursive functions is definable in this system; see Girard–Lafond–Taylor [**?**].

Although theoretically interesting, this is not a practical system.

Another wilder method is to allow more general types to the simply-typed λ-calculus, the so-called *second-order types* or *polymorphic types*.

In addition to base types, we allow *type variables* (often denoted $X, Y, \ldots$) ranging over simple types and new types of the form $\forall X.\, \sigma$.[3]

[3]Barendregt and others used Greek letters to denote type variables but we find this confusing.

For example, $\forall X.\,(X \to X)$ is such a new type, and so is

$$\forall X.\,(X \to ((X \to X) \to X)).$$

Actually, the second-order types that we just defined are special cases of the QBF (quantified boolean formulae) arising in complexity theory restricted to implication and universal quantifiers; see Section **??**.

Remarkably, the other connectives $\wedge, \vee, \neg$ and $\exists$ are definable in terms of $\to$ (as a logical connective, $\Rightarrow$) and $\forall$; see Troelstra and Schwichtenberg [**?**] (Chapter 11).

Remark: The type

$$\mathsf{Nat} = \forall X.\,(X \to ((X \to X) \to X)).$$

can be chosen to represent the type of the natural numbers.

The type of the natural numbers can also be chosen to be

$$\forall X.\,((X \to X) \to (X \to X)).$$

This makes essentially no difference but the first choice has some technical advantages.

There is also a new form of *type abstraction*, $\Lambda X.\, M$, and of *type application*, $M\sigma$, where M is a λ-term and σ is a type. There are two new typing rules:

$$\frac{\Gamma \rhd M : \sigma}{\Gamma \rhd (\Lambda X.\, M) : \forall X.\, \sigma} \quad \text{(type abstraction)}$$

provided that X does not occur free in any of the types in Γ, and

$$\frac{\Gamma \rhd M : \forall X.\, \sigma}{\Gamma \rhd (M\tau) : \sigma[X := \tau]} \quad \text{(type application)}$$

where τ is any type (and no capture of variable takes place).

From the point of view where types are viewed as propositions and λ-terms are viewed as proofs, type abstraction is an introduction rule and type application is an elimination rule, both for the second-order quantifier $\forall$.

We also have a new reduction rule

$$(\Lambda X.\, M)\sigma \longrightarrow_{\beta\forall} M[X := \sigma]$$

that corresponds to a new form of redundancy in proofs having to do with a $\forall$-elimination immediately following a $\forall$-introduction.

Here in the substitution $M[X := \tau]$, all free occurrences of X in M and the types in M are replaced by τ.

For example,

$$\begin{gathered}(\Lambda X.\, \lambda f\colon (X \to X).\, \lambda x\colon X.\, \lambda g\colon \forall Y.\, (Y \to Y).\, gX\, fx) \\ [X := \tau] \\ = \lambda f\colon (\tau \to \tau).\, \lambda x\colon \tau.\, \lambda g\colon \forall Y.\, (Y \to Y).\, g\tau\, xf.\end{gathered}$$

This new typed λ-calculus is the *second-order polymorphic lambda calculus*. It was invented by Girard (1972) who named it *system F*; see Girard [**?**, **?**], and it is denoted $\boldsymbol{\lambda 2}$ by Barendregt.

From the point of view of logic, Girard's system is a proof system for *intuitionistic second-order propositional logic*.

We define $\overset{+}{\longrightarrow}_{\boldsymbol{\lambda 2}}$ and $\overset{*}{\longrightarrow}_{\boldsymbol{\lambda 2}}$ as the relations

$$\overset{+}{\longrightarrow}_{\boldsymbol{\lambda 2}} = (\longrightarrow_{\beta} \cup \longrightarrow_{\beta\forall})^{+}$$
$$\overset{*}{\longrightarrow}_{\boldsymbol{\lambda 2}} = (\longrightarrow_{\beta} \cup \longrightarrow_{\beta\forall})^{*}.$$

A variant of system F was also introduced independently by John Reynolds (1974) but for very different reasons.

The intuition behind terms of type $\forall X.\, \sigma$ is that a term M of type $\forall X.\, \sigma$ is a sort of *generic function* such that for *any* type τ, the function $M\tau$ is a *specialized version* of type $\sigma[X := \tau]$ of M.

For example, M could be the function that appends an element to a list, and for specific types such as the natural numbers Nat, strings String, trees Tree, *etc.*, the functions $M\mathsf{Nat}$, $M\mathsf{String}$, $M\mathsf{Tree}$, are the specialized versions of M to lists of elements having the specific data types $\mathsf{Nat}, \mathsf{String}, \mathsf{Tree}$.

For example, if σ is any type, we have the closed term

$$\mathbf{A}_\sigma = \lambda x\colon \sigma.\, \lambda f\colon (\sigma \to \sigma).\, fx,$$

of type $\sigma \to ((\sigma \to \sigma) \to \sigma)$, such that for every term F of type $\sigma \to \sigma$ and every term a of type σ,

$$\mathbf{A}_\sigma aF \xrightarrow{+}_{\boldsymbol{\lambda 2}} Fa.$$

Since $\mathbf{A}_\sigma$ has the same behavior for *all* types σ, it is natural to define the generic function $\mathbf{A}$ given by

$$\mathbf{A} = \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, fx,$$

which has type $\forall X.\, (X \to ((X \to X) \to X))$, and then $\mathbf{A}\sigma$ has the same behavior as $\mathbf{A}_\sigma$.

We will see shortly that $\mathbf{A}$ is the Church numeral $\mathbf{c}_1$ in $\boldsymbol{\lambda 2}$.

Remarkably, system F is *strongly normalizing*, which means that *every λ-term typable in system F has a β-normal form*.

The proof of this theorem is hard and was one of Girard's accomplishments in his dissertation, Girard [**?**].

The Church–Rosser property also holds for system F. The proof technique used to prove that system F is strongly normalizing is thoroughly analyzed in Gallier [**?**].

We stated earlier that deciding whether a simple type σ is provable, that is, whether there is a closed λ-term M that type-checks in the simply-typed λ-calculus such that the judgement $\triangleright M : \sigma$ is provable is a hard problem.

Indeed Statman proved that this problem is P-space complete; see Statman [**?**] and Section **??**.

It is natural so ask whether it is decidable whether given any second-order type σ, there is a closed λ-term M that type-checks in system F such that the judgement $\triangleright M : \sigma$ is provable (if σ is viewed as a second-order logical formula, the problem is to decide whether σ is provable).

Surprisingly the answer is *no*; this problem (called *inhabitation*) is undecidable.

This result was proven by Löb around 1976, see Barendregt [**?**].

This undecidability result is troubling and at first glance seems paradoxical.

Indeed, viewed as a logical formula, a second-order type σ is a QBF (a quantified boolean formula), and if we assign the truth values **F** and **T** to the boolean variables in it, we can decide whether such a proposition is valid in exponential time and polynomial space (in fact, we will see that later QBF validity is P-space complete).

This seems in contradiction with the fact that provability is undecidable.

But the proof system corresponding to system F is an *intuitionistic proof system*, so there are (non-quantifed) propositions that are valid in the truth-value semantics but not provable in intuitionistic propositional logic.

The set of second-order propositions provable in intuitionistic second-order logic is a *proper* subset of the set of valid QBF (under the truth-value semantics), and it is *not computable*. So there is no paradox after all.

Going back to the issue of computability of numerical functions, a version of the *Church numerals* can be defined as

$$\mathbf{c}_n = \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f^n(x). \qquad (*_{c1})$$

Observe that $\mathbf{c}_n$ has type Nat.

Also note that variables x and f now appear in the order x, f in the λ-binder, as opposed to f, x as in Definition 4.8.

Inspired by the definition of **Succ** given in Section 4.4, we can define the successor function on the natural numbers as

$$\mathbf{Succ} = \lambda n\colon \mathsf{Nat}.\, \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f(nX\, xf).$$

Note how n, which is of type
$\mathsf{Nat} = \forall X.\, (X \to ((X \to X) \to X))$, is applied to the type variable X in order to become a term nX of type $X \to ((X \to X) \to X)$, so that $nX\, xf$ has type X, thus $f(nX\, xf)$ also has type X.

For every type σ, every term F of type $\sigma \to \sigma$ and every term a of type σ, we have

$$\begin{aligned}\mathbf{c}_n\sigma\, aF &= \big(\Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, f^n(x)\big)\sigma\, aF \\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} \big(\lambda x\colon \sigma.\, \lambda f\colon (\sigma \to \sigma).\, f^n(x)\big)aF \\ &\xrightarrow{+}_{\boldsymbol{\lambda 2}} F^n(a);\end{aligned}$$

that is,

$$\mathbf{c}_n\sigma\, aF \xrightarrow{+}_{\boldsymbol{\lambda 2}} F^n(a). \qquad (*_{c2})$$

So $\mathbf{c}_n\sigma$ iterates F n times starting with a. As a consequence,

$$\mathbf{Succ}\, \mathbf{c}_n \xrightarrow{+}_{\boldsymbol{\lambda 2}} \mathbf{c}_{n+1}.$$

We can also define addition of natural numbers as

$$\mathbf{Add} = \lambda m\colon \mathsf{Nat}.\, \lambda n\colon \mathsf{Nat}.\, \Lambda X.\, \lambda x\colon X.\, \lambda f\colon (X \to X).\, (mX\, f(nX\, xf))f.$$

Note how m and n, which are of type $\mathsf{Nat} = \forall X.\, (X \to ((X \to X) \to X))$, are applied to the type variable X in order to become terms mX and nX of type $X \to ((X \to X) \to X)$, so that $nX\, xf$ has type X, thus $f(nX\, xf)$ also has type X, and $mX\, f(nX\, xf)$ has type $(X \to X) \to X$, and finally $(mX\, f(nX\, xf))f$ has type X.

Many of the constructions that can be performed in the pure λ-calculus can be mimicked in system F, which explains its expressive power.

For example, for any two second-order types σ and τ, we can define a *pairing function* $\langle -, - \rangle$ (to be very precise, $\langle -, - \rangle_{\sigma,\tau}$) given by

$$\langle -, - \rangle = \lambda u\colon \sigma.\, \lambda v\colon \tau.\, \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fuv,$$

of type $\sigma \to \big(\tau \to \big(\forall X.\, ((\sigma \to (\tau \to X)) \to X)\big)\big)$.

Given any term M of type σ and any term N of type τ, we have

$$\langle -, - \rangle_{\sigma,\tau} MN \xrightarrow{*}_{\boldsymbol{\lambda 2}} \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, fMN.$$

Thus we define $\langle M, N\rangle$ as

$$\langle M, N\rangle = \Lambda X.\, \lambda f\colon \sigma \to (\tau \to X).\, f M N,$$

and the type

$$\forall X.\, ((\sigma \to (\tau \to X)) \to X)$$

of $\langle M, N\rangle$ is denoted by $\sigma \times \tau$.

As a logical formula it is equivalent to $\sigma \wedge \tau$, which means that if we view σ and τ as (second-order) propositions, then

$$\sigma \wedge \tau \equiv \forall X.\, ((\sigma \to (\tau \to X)) \to X)$$

is provable intuitionistically.

This is a special case of the result that we mentioned earlier: the connectives $\wedge, \vee, \neg$ and $\exists$ are definable in terms of $\to$ (as a logical connective, $\Rightarrow$) and $\forall$.

Proposition 4.13. *The connectives* $\wedge, \vee, \neg, \bot$ *and* $\exists$ *are definable in terms of* $\to$ *and* $\forall$*, which means that the following equivalences are provable intuitionistically, where* X *is not free in* σ *or* τ*:*

$$\begin{aligned}
\sigma \wedge \tau &\equiv \forall X.\, \big((\sigma \to (\tau \to X)) \to X\big) \\
\sigma \vee \tau &\equiv \forall X.\, \big((\sigma \to X) \to ((\tau \to X) \to X)\big) \\
\bot &\equiv \forall X.\, X \\
\neg\sigma &\equiv \sigma \to \forall X.\, X \\
\exists Y.\, \sigma &\equiv \forall X.\, \big((\forall Y.\, (\sigma \to X)) \to X\big).
\end{aligned}$$

Remark: The rule of type application implies that $\bot \to \sigma$ is intuitionistically provable for *all* propositions (types) σ.

So in second-order logic there is no difference between minimal and intuitionistic logic.

We also have two *projections* π_1 and π_2 (to be very precise $\pi_1^{\sigma\times\tau}$ and $\pi_2^{\sigma\times\tau}$) given by

$$\pi_1 = \lambda g\colon \sigma \times \tau.\, g\sigma(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, x)$$
$$\pi_2 = \lambda g\colon \sigma \times \tau.\, g\tau(\lambda x\colon \sigma.\, \lambda y\colon \tau.\, y).$$

It is easy to check that π_1 has type $(\sigma \times \tau) \to \sigma$ and that π_2 has type $(\sigma \times \tau) \to \tau$.

The reader should check that for any M of type σ and any N of type τ we have

$$\pi_1\langle M, N\rangle \xrightarrow{+}_{\boldsymbol{\lambda 2}} M \quad \text{and} \quad \pi_2\langle M, N\rangle \xrightarrow{+}_{\boldsymbol{\lambda 2}} N.$$

The *booleans* can be defined as

$$\mathbf{T} = \Lambda X.\, \lambda x\colon X.\, \lambda y\colon X.\, x$$
$$\mathbf{F} = \Lambda X.\, \lambda x\colon X.\, \lambda y\colon X.\, y,$$

both of type $\mathsf{Bool} = \forall X.\, (X \to (X \to X))$.

We also define $\mathsf{if\ then\ else}$ as

$$\mathsf{if\ then\ else} = \Lambda X.\, \lambda z\colon \mathsf{Bool}.\, zX$$

of type $\forall X.\, \mathsf{Bool} \to (X \to (X \to X))$.

It is easy that for any type σ and any two terms M and N of type σ we have

$$(\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma \xrightarrow{+}_{\boldsymbol{\lambda 2}} M$$
$$(\mathsf{if}\ \mathbf{F}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma \xrightarrow{+}_{\boldsymbol{\lambda 2}} N,$$

where we write $(\mathsf{if}\ \mathbf{T}\ \mathsf{then}\ M\ \mathsf{else}\ N)\sigma$ instead of $(\mathsf{if\ then\ else})\ \sigma\, \mathbf{T} M N$ (and similarly for the other term).

Lists, trees, and other inductively data stuctures are also representable in system F; see Girard–Lafond–Taylor [**?**].

We can also define an *iterator* **Iter** given by

$$\mathbf{Iter} = \Lambda X.\, \lambda u\colon X.\, \lambda f\colon (X \to X).\, \lambda z\colon \mathsf{Nat}.\, zX\, uf$$

of type $\forall X.\, (X \to ((X \to X) \to (\mathsf{Nat} \to X)))$.

The idea is that given f of type $\sigma \to \sigma$ and u of type σ, the term $\mathbf{Iter}\,\sigma\, uf\mathbf{c}_n$ iterates f n times over the input u.

It is easy to show that for any term t of type Nat we have

$$\begin{aligned}\mathbf{Iter}\,\sigma\, uf\mathbf{c}_0 &\xrightarrow{+}_{\boldsymbol{\lambda 2}} u\\ \mathbf{Iter}\,\sigma\, uf(\mathbf{Succ_c}\, t) &\xrightarrow{+}_{\boldsymbol{\lambda 2}} f(\mathbf{Iter}\,\sigma\, uft),\end{aligned}$$

and that

$$\mathbf{Iter}\,\sigma\, uf\mathbf{c}_n \xrightarrow{+}_{\boldsymbol{\lambda 2}} f^n(u).$$

Then mimicking what we did in the pure λ-calculus, we can show that *the primitive recursive functions are λ-definable in system F*.

Actually, higher-order primitive recursion is definable. So, for example, Ackermann's function is definable.

Remarkably, the class of numerical functions definable in system F is a class of (total) computable functions much bigger than the class of primitive recursive functions.

This class of functions was characterized by Girard as *the functions that are provably-recursive in a formalization of arithmetic known as intuitionistic second-order arithmetic*; see Girard [**?**], Troelstra and Schwichtenberg [**?**] and Girard–Lafond–Taylor [**?**].

It can also be shown (using a diagonal argument) that there are (total) computable functions not definable in system F.

From a theoretical point of view, every (total) function that we will ever want to compute is definable in system F.

However, from a practical point of view, programming in system F is very tedious and usually leads to very inefficient programs.

Nevertheless polymorphism is an interesting paradigm which had made its way in certain programming languages.

Type systems even more powerful than system F have been designed, the ultimate system being the *calculus of constructions* due to Huet and Coquand, but these topics are beyond the scope of these notes.

One last comment has to do with the use of the simply-typed λ-calculus as a the core of a programming language.

In the early 1970's Dana Scott defined a system named LCF based on the the simply-typed λ-calculus and obtained by adding the natural numbers and the booleans as data types, product types, and a fixed-point operator.

Robin Milner then extended LCF, and as a by-product, defined a programming language known as ML, which is the ancestor of most functional programming languages.

A masterful and thorough exposition of type theory and its use in programming language design is given in Pierce [**?**].

We now revisit the problem of defining the partial computable functions.

4.8 Head Normal-Forms and the Partial Computable Functions

One defect of the proof of Theorem 4.11 in the case where a computable function is partial is the use of the Kleene normal form.

The difficulty has to do with composition.

Given a partial computable function g λ-defined by a closed term G and a partial computable function h λ-defined by a closed term H (for simplicity we assume that both g and h have a single argument), it would be nice if the composition $h \circ g$ was represented by $\lambda x.\, H(Gx)$.

This is true if both g and h are total, but false if either g or h is partial as shown by the following example from Barendregt [**?**] (Chapter 2, §2).

If g is the function undefined everywhere and h is the constant function 0, then g is λ-defined by $G = \mathbf{K}\mathbf{\Omega}$ and h is λ-defined by $H = \mathbf{K}\,\mathbf{c}_0$, with $\mathbf{\Omega} = (\lambda x.\,(xx))(\lambda x.\,(xx))$.

We have

$$\lambda x.\,H(Gx) = \lambda x.\,\mathbf{K}\,\mathbf{c}_0(\mathbf{K}\mathbf{\Omega}x) \xrightarrow{+}_\beta \lambda x.\,\mathbf{K}\mathbf{c}_0\mathbf{\Omega} \xrightarrow{+}_\beta \lambda x.\,\mathbf{c}_0,$$

but $h \circ g = g$ is the function undefined everywhere, and $\lambda x.\,\mathbf{c}_0$ represents the total function h, so $\lambda x.\,H(Gx)$ *does not* λ-define $h \circ g$.

It turns out that the λ-definability of the partial computable functions can be obtained in a more elegant fashion without having recourse to the Kleene normal form by capturing the fact that a function is undefined for some input is a more subtle way.

The key notion is the notion of *head normal form*, which is more general than the notion of β-normal form.

As a consequence, there a *fewer* λ-terms having *no* head normal form than λ-terms having *no* β-normal form, and we capture a stronger form of divergence.

Recall that a λ-term is either a variable x, or an application (MN), or a λ-abstraction $(\lambda x.\, M)$.

We can sharpen this characterization as follows.

Proposition 4.14. *The following properties hold:*

(1) Every application term M is of the form

$$M = (N_1 N_2 \cdots N_{n-1}) N_n, \quad n \geq 2,$$

where N_1 is not an application term.

(2) Every abstraction term M is of the form

$$M = \lambda x_1 \cdots x_n.\, N, \quad n \geq 1,$$

where N is not an abstraction term.

(3) Every λ-term M is of one of the following two forms:

$$M = \lambda x_1 \cdots x_n.\, x M_1 \cdots M_m, \quad m, n \geq 0 \qquad \text{(a)}$$

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0) M_1 \cdots M_m, \quad m \geq 1, n \geq 0, \qquad \text{(b)}$$

where x is a variable.

The terms, $\mathbf{I}, \mathbf{K}, \mathbf{K}_*, \mathbf{S}$, the Church numerals $\mathbf{c}_n$, $\mathsf{if\,then\,else}$, $\langle M, N\rangle$, π_1, π_2, $\mathbf{Iter}$, $\mathbf{Succ_c}$, $\mathbf{Add}$ and $\mathbf{Mult}$ as in Proposition 4.5, are λ-terms of type (a).

However, $\mathbf{Pred_K}$, $\mathbf{\Omega} = (\lambda x.\,(xx))(\lambda x.\,(xx))$, $\mathbf{Y}$ (the Curry $\mathbf{Y}$-combinator), $\mathbf{\Theta}$ (the Turing $\mathbf{\Theta}$-combinator) are of type (b).

Proposition 4.14 motivates the following definition.

Definition 4.17. A λ-term M is a *head normal form* (for short *hnf*) if it is of the form (a), namely

$$M = \lambda x_1 \cdots x_n.\, x M_1 \cdots M_m, \quad m, n \geq 0,$$

where x is a variable called the *head variable*.

A λ-term M *has a head normal form* if there is some head normal form N such that $M \xrightarrow{*}_\beta N$.

In a term M of the form (b),

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0) M_1 \cdots M_m, \quad m \geq 1, n \geq 0,$$

the subterm $(\lambda x.\, M_0)M_1$ is called the *head redex* of M.

In addition to the terms of type (a) that we listed after Proposition 4.14, the term $\lambda x.\, x\mathbf{\Omega}$ is a head normal form.

It is the head normal form of the term $\lambda x.\, (\mathbf{I}x)\mathbf{\Omega}$, which has *no* β-normal form.

Not every term has a head normal form. For example, the term

$$\mathbf{\Omega} = (\lambda x.\, (xx))(\lambda x.\, (xx))$$

has no head normal form.

Every β-normal form must be a head normal form, but the converse is false as we saw with

$$M = \lambda x.\, x\mathbf{\Omega},$$

which is a head normal form but has no β-normal form.

Note that a head redex of a term is a leftmost redex, but not conversely, as shown by the term $\lambda x.\, x((\lambda y.\, y)x)$.

A term may have more than one head normal form but here is a way of obtaining a head normal form (if there is one) in a systematic fashion.

Definition 4.18. The relation $\longrightarrow_h$, called *one-step head reduction*, is defined as follows: For any two terms M and N, if M contains a head redex $(\lambda x.\, M_0)M_1$, which means that M is of the form

$$M = \lambda x_1 \cdots x_n.\, (\lambda x.\, M_0)M_1 \cdots M_m, \quad m \geq 1, n \geq 0,$$

then $M \longrightarrow_h N$ with

$$N = \lambda x_1 \cdots x_n.\, (M_0[x := M_1])M_2 \cdots M_m.$$

We denote by $\overset{+}{\longrightarrow}_h$ the transitive closure of $\longrightarrow_h$ and by $\overset{*}{\longrightarrow}_h$ the reflexive and transitive closure of $\longrightarrow_h$.

Given a term M containing a head redex, the *head reduction sequence of M* is the uniquely determined sequence of one-step head reductions

$$M = M_0 \longrightarrow_h M_1 \longrightarrow_h \cdots \longrightarrow_h M_n \longrightarrow_h \cdots .$$

If the head reduction sequence reaches a term M_n which is a head normal form we say that the sequence *terminates*, and otherwise we say that M has an *infinite* head reduction.

The following result is shown in Barendregt [**?**] (Chapter 8, §3).

Theorem 4.15. *(Wadsworth) A λ-term M has a head normal form if and only if the head reduction sequence terminates.*

In some intuitive sense, a λ-term M that does not have any head normal form has a strong divergence behavior with respect to β-reduction.

Remark: There is a notion more general than the notion of head normal form which comes up in functional languages (for example, Haskell). A λ-term M is a *weak head normal form* if it of one of the two forms

$$\lambda x.\, N \quad \text{or} \quad y N_1 \cdots N_m$$

where y is a variable

These are exactly the terms that do not have a redex of the form $(\lambda x.\, M_0) M_1 N_1 \cdots N_m$.

Every head normal form is a weak head normal form, but there are many more weak head normal forms than there are head normal forms since a term of the form $\lambda x.\, N$ where N is *arbitrary* is a weak head normal form, but not a head normal form unless N is of the form $\lambda x_1 \cdots x_n.\, x M_1 \cdots M_m$, with $m, n \geq 0$.

Reducing to a weak head normal form is a *lazy evaluation strategy*.

There is also another useful notion which turns out to be equivalent to having a head normal form.

Definition 4.19. A closed λ-term M is *solvable* if there are closed terms $N_1, \ldots, N_n$ such that

$$M N_1 \cdots N_n \xrightarrow{*}_\beta \mathbf{I}.$$

A λ-term M with free variables $x_1, \ldots, x_m$ is *solvable* if the closed term $\lambda x_1 \cdots x_m.\, M$ is solvable. A term is *unsolvable* if it is not solvable.

The following result is shown in Barendregt [**?**] (Chapter 8, §3).

Theorem 4.16. *(Wadsworth) A λ-term M has a head normal form if and only if is it solvable.*

Actually, the proof that having a head normal form implies solvable is not hard.

We are now ready to revise the notion of λ-definability of numerical functions.

Definition 4.20. A function (partial or total) $f\colon \mathbb{N}^n \to \mathbb{N}$ is *strongly λ-definable* if for all $m_1, \ldots, m_n \in \mathbb{N}$, there is a combinator (a closed λ-term) F with the following properties:

(1) If the value $f(m_1, \ldots, m_n)$ is defined, then $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ reduces to the β-normal form $\mathbf{c}_{f(m_1,\ldots,m_n)}$.

(2) If $f(m_1, \ldots, m_n)$ is undefined, then $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ has no head normal form, or equivalently, is unsolvable.

Observe that in Case (2), when the value $f(m_1, \ldots, m_n)$ is undefined, the divergence behavior of $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ is stronger than in Definition 4.16.

Not only $F\mathbf{c}_{m_1} \cdots \mathbf{c}_{m_n}$ has no β-normal form, but actually it has *no head normal form*.

The following result is proven in Barendregt [**?**] (Chapter 8, §4).

The proof does not use the Kleene normal form. Instead, it makes clever use of the term **KII**. Another proof is given in Krivine [**?**] (Chapter II).

Theorem 4.17. *Every partial or total computable function is strongly λ-definable. Conversely, every strongly λ-definable function is partial computable.*

Making sure that a composition $g \circ (h_1, \ldots, h_m)$ is defined for some input $x_1, \ldots, x_n$ iff all the $h_i(x_1, \ldots, x_n)$ and $g(h_1(x_1, \ldots, x_n), \ldots, h_m(x_1, \ldots, x_n))$ are defined is tricky.

The term **KII** comes to the rescue!

If g is strongly λ-definable by G and the h_i are strongly λ-definable by H_i, then it can be shown that the combinator F given by

$$F = \lambda x_1 \cdots x_n.\, (H_1 x_1 \cdots x_n \mathbf{KII}) \cdots (H_m x_1 \cdots x_n \mathbf{KII}) \\ (G(H_1 x_1 \cdots x_n) \cdots (G(H_m x_1 \cdots x_n))$$

strongly λ-defines F; see Barendregt [**?**] (Chapter 8, Lemma 8.4.6).