This chapter is experimental. It needs more explanation, and the material should be structured better into definitions and propositions with proofs, and more examples.

### ldf.1 Introduction

At first glance, the lambda calculus is just a very abstract calculus of expressions that represent functions and applications of them to others. Nothing in the syntax of the lambda calculus suggests that these are functions of particular kinds of objects, in particular, the syntax includes no mention of natural numbers. Its basic operations—application and lambda abstractions—are operations that apply to any function, not just functions on natural numbers.

Nevertheless, with some ingenuity, it is possible to define arithmetical functions, i.e., functions on the natural numbers, in the lambda calculus. To do this, we define, for each natural number \( n \in \mathbb{N} \), a special \( \lambda \)-term \( n \), the *Church numeral* for \( n \). (Church numerals are named for Alonzo Church.)

**Definition ldf.1.** If \( n \in \mathbb{N} \), the corresponding *Church numeral* \( \overline{n} \) represents \( n \):

\[
\overline{n} \equiv \lambda f x. f^n(x)
\]

Here, \( f^n(x) \) stands for the result of applying \( f \) to \( x \) \( n \) times. For example, \( \overline{0} \) is \( \lambda f x. x \), and \( \overline{3} \) is \( \lambda f x. f(f(f x)) \).

The Church numeral \( \overline{n} \) is encoded as a lambda term which represents a function accepting two arguments \( f \) and \( x \), and returns \( f^n(x) \). Church numerals are evidently in normal form.

A representation of natural numbers in the lambda calculus is only useful, of course, if we can compute with them. Computing with Church numerals in the lambda calculus means applying a \( \lambda \)-term \( F \) to such a Church numeral,
and reducing the combined term $F \pi$ to a normal form. If it always reduces to a normal form, and the normal form is always a Church numeral $\pi$, we can think of the output of the computation as being the number $m$. We can then think of $F$ as defining a function $f: \mathbb{N} \rightarrow \mathbb{N}$, namely the function such that $f(n) = m$ iff $F \pi \rightarrow \pi$. Because of the Church-Rosser property, normal forms are unique if they exist. So if $F \pi \rightarrow \pi$, there can be no other term in normal form, in particular no other Church numeral, that $F \pi$ reduces to.

Conversely, given a function $f: \mathbb{N} \rightarrow \mathbb{N}$, we can ask if there is a term $F$ that defines $f$ in this way. In that case we say that $F$ $\lambda$-defines $f$, and that $f$ is $\lambda$-definable. We can generalize this to many-place and partial functions.

**Definition ldf.2.** Suppose $f: \mathbb{N}^k \rightarrow \mathbb{N}$. We say that a lambda term $F$ $\lambda$-defines $f$ if for all $n_0, \ldots, n_{k-1}$,

$$F \pi_0 \pi_1 \ldots \pi_{k-1} \rightarrow f(n_0, n_1, \ldots, n_{k-1})$$

if $f(n_0, \ldots, n_{k-1})$ is defined, and $F \pi_0 \pi_1 \ldots \pi_{k-1}$ has no normal form otherwise.

A very simple example are the constant functions. The term $C_k \equiv \lambda x. k$ $\lambda$-defines the function $c_k: \mathbb{N} \rightarrow \mathbb{N}$ such that $c_k(n) = k$. For $C_k \pi \equiv (\lambda x. k) \pi \rightarrow k$ for any $n$. The identity function is $\lambda$-defined by $\lambda x. x$. More complex functions are of course harder to define, and often require a lot of ingenuity. So it is perhaps surprising that every computable function is $\lambda$-definable. The converse is also true: if a function is $\lambda$-definable, it is computable.

**rep.2 $\lambda$-Definable Arithmetical Functions**

**Proposition rep.3.** The successor function $\text{succ}$ is $\lambda$-definable.

**Proof.** A term that $\lambda$-defines the successor function is

$$\text{Succ} \equiv \lambda a. \lambda fx. f(ax).$$

Given our conventions, this is short for

$$\text{Succ} \equiv \lambda a. \lambda f. \lambda x. (f((af)x)).$$

Succ is a function that accepts as argument a number $a$, and evaluates to another function, $\lambda fx. f(ax)$. That function is not itself a Church numeral. However, if the argument $a$ is a Church numeral, it reduces to one. Consider:

$$(\lambda a. \lambda fx. f(ax)) \pi \rightarrow \lambda fx. f(\pi fx).$$

The embedded term $\pi fx$ is a redex, since $\pi$ is $\lambda fx. f^n x$. So $\pi fx \rightarrow f^n x$ and so, for the entire term we have

$$\text{Succ} \pi \rightarrow \lambda fx. f(f^n(x)),$$

i.e., $n + 1$. □
Example rep.4. Let’s look at what happens when we apply \( \text{Succ} \) to \( \overline{0} \), i.e., \( \lambda f. x \). We’ll spell the terms out in full:

\[
\text{Succ} \overline{0} \equiv (\lambda a. \lambda f. \lambda x. (f((af)x)))((\lambda f. \lambda x. x)) \\
\rightarrow \lambda f. \lambda x. (f(((\lambda f. \lambda x. x)f)x)) \\
\rightarrow \lambda f. \lambda x. (f((\lambda x. x)x)) \\
\rightarrow \lambda f. \lambda x. (fx) \equiv 1
\]

Problem rep.1. The term

\[ \text{Succ}' \equiv \lambda n. \lambda f. x. nf(fx) \]

\( \lambda \)-defines the successor function. Explain why.

Proposition rep.5. *The addition function* \( \text{add} \) *is \( \lambda \)-definable.*

Proof. Addition is \( \lambda \)-defined by the terms

\[ \text{Add} \equiv \lambda ab. \lambda f. x. af(bfx) \]

or, alternatively,

\[ \text{Add}' \equiv \lambda ab. a \text{ Succ } b. \]

The first addition works as follows: Add first accept two numbers \( a \) and \( b \). The result is a function that accepts \( f \) and \( x \) and returns \( af(bfx) \). If \( a \) and \( b \) are Church numerals \( \overline{n} \) and \( \overline{m} \), this reduces to \( f^{n+m}(x) \), which is identical to \( f^n(f^m(x)) \). Or, slowly:

\[
(\lambda ab. \lambda f. x. af(bfx))n m \rightarrow \lambda f. x. f^m f(x) \\
\rightarrow \lambda f. x. f^{n+m}(x) \\
\rightarrow \lambda f. x. f^n(f^m(x)) \equiv \overline{n+m}.
\]

The second representation of addition \( \text{Add}' \) works differently: Applied to two Church numerals \( \overline{n} \) and \( \overline{m} \),

\[ \text{Add}' \overline{n} \overline{m} \rightarrow \overline{n} \text{ Succ } \overline{m}. \]

But \( \overline{n} f. x \) always reduces to \( f^n(x) \). So,

\[ \overline{n} \text{ Succ } \overline{m} \rightarrow \text{ Succ }^n(\overline{m}) \]

And since \( \text{Succ} \) \( \lambda \)-defines the successor function, and the successor function applied \( n \) times to \( m \) gives \( n + m \), this in turn reduces to \( \overline{n+m} \). \( \square \)
**Proposition rep.6.** Multiplication is \(\lambda\)-definable by the term

\[
\text{Mult} \equiv \lambda ab. \lambda fx. a(bf)x
\]

*Proof.* To see how this works, suppose we apply \(\text{Mult}\) to Church numerals \(\text{n}\) and \(\text{m}\): \(\text{Mult}\; \text{n}\; \text{m}\) reduces to \(\lambda fx. \text{n}(\text{m}\; f)x\). The term \(\text{m}\; f\) defines a function which applies \(f\) to its argument \(m\) times. Consequently, \(\text{n}(\text{m}\; f)x\) applies the function “apply \(f\) \(m\) times” itself \(n\) times to \(x\). In other words, we apply \(f\) to \(x\), \(n \cdot m\) times. But the resulting normal term is just the Church numeral \(\text{nm}\).

We can actually simplify this term further by \(\eta\)-reduction:

\[
\text{Mult} \equiv \lambda ab. \lambda f. a(bf).
\]

But then we first have to explain \(\eta\)-reduction.

**Problem rep.2.** Multiplication can be \(\lambda\)-defined by the term

\[
\text{Mult}' \equiv \lambda ab. a(\text{Add } a)\text{0}.
\]

Explain why this works.

The definition of exponentiation as a \(\lambda\)-term is surprisingly simple:

\[
\text{Exp} \equiv \lambda be. eb.
\]

The first argument \(b\) is the base and the second \(e\) is the exponent. Intuitively, \(ef\) is \(f^e\) by our encoding of numbers. If you find it hard to understand, we can still define exponentiation also by iterated multiplication:

\[
\text{Exp}' \equiv \lambda be. e(\text{Mult } b)\text{1}.
\]

Predecessor and subtraction on Church numeral is not as simple as we might think: it requires encoding of pairs.
Given a pair, we also want to recover its elements. For this we need two access functions, which accept a pair as argument and return the first or second elements in it:

\[
\text{Fst} \equiv \lambda p. p (\lambda mn. m)
\]
\[
\text{Snd} \equiv \lambda p. p (\lambda mn. n)
\]

**Problem ldf.3.** Explain why the access functions Fst and Snd work.

Now with pairs we can \(\lambda\)-define the predecessor function:

\[
\text{Pred} \equiv \lambda n. \text{Fst}(n (\lambda p. (\text{Snd} p, \text{Succ}(\text{Snd} p))(\overline{0}, \overline{0})))
\]

Remember that \(\overline{n} f x\) reduces to \(f^n(x)\); in this case \(f\) is a function that accepts a pair \(p\) and returns a new pair containing the second component of \(p\) and the successor of the second component; \(x\) is the pair \(\langle 0, 0 \rangle\). Thus, the result is \(\langle 0, 0 \rangle\) for \(n = 0\), and \(\langle n - 1, \overline{n} \rangle\) otherwise. Pred then returns the first component of the result.

Subtraction can be defined as Pred applied to \(a, b\) times:

\[
\text{Sub} \equiv \lambda ab. b \text{Pred} a.
\]

### ldf.4 Truth Values and Relations

We can encode truth values in the pure lambda calculus as follows:

\[
\text{true} \equiv \lambda x. \lambda y. x
\]
\[
\text{false} \equiv \lambda x. \lambda y. y
\]

Truth values are represented as selectors, i.e., functions that accept two arguments and returning one of them. The truth value true selects its first argument, and false its second. For example, \(\text{true} M N\) always reduces to \(M\), while \(\text{false} M N\) always reduces to \(N\).

**Definition ldf.8.** We call a relation \(R \subseteq \mathbb{N}^n\) \(\lambda\)-definable if there is a term \(R\) such that

\[
R \overline{n_1} \ldots \overline{n_k} \rightarrow^* \text{true}
\]

whenever \(R(n_1, \ldots, n_k)\) and

\[
R \overline{n_1} \ldots \overline{n_k} \rightarrow^* \text{false}
\]

otherwise.
For instance, the relation \( \text{IsZero} = \{0\} \) which holds of 0 and 0 only, is \( \lambda \)-definable by
\[
\text{IsZero} \equiv \lambda n. n(\lambda x. \text{false}) \text{true}.
\]

How does it work? Since Church numerals are defined as iterators (functions which apply their first argument \( n \) times to the second), we set the initial value to be true, and for every step of iteration, we return false regardless of the result of the last iteration. This step will be applied to the initial value \( n \) times, and the result will be true if and only if the step is not applied at all, i.e., when \( n = 0 \).

On the basis of this representation of truth values, we can further define some truth functions. Here are two, the representations of negation and conjunction:
\[
\text{Not} \equiv \lambda x. x \text{false true}
\]
\[
\text{And} \equiv \lambda x. \lambda y. xy \text{false}
\]

The function “Not” accepts one argument, and returns true if the argument is false, and false if the argument is true. The function “And” accepts two truth values as arguments, and should return true iff both arguments are true. Truth values are represented as selectors (described above), so when \( x \) is a truth value and is applied to two arguments, the result will be the first argument if \( x \) is true and the second argument otherwise. Now And takes its two arguments \( x \) and \( y \), and in return passes \( y \) and false to its first argument \( x \). Assuming \( x \) is a truth value, the result will evaluate to \( y \) if \( x \) is true, and to false if \( x \) is false, which is just what is desired.

Note that we assume here that only truth values are used as arguments to And. If it is passed other terms, the result (i.e., the normal form, if it exists) may well not be a truth value.

**Problem ldf.4.** Define the functions Or and Xor representing the truth functions of inclusive and exclusive disjunction using the encoding of truth values as \( \lambda \)-terms.

**ldf.5 Primitive Recursive Functions are \( \lambda \)-Definable**

Recall that the primitive recursive functions are those that can be defined from the basic functions zero, succ, and \( P^n_i \) by composition and primitive recursion.

**Lemma ldf.9.** The basic primitive recursive functions zero, succ, and projections \( P^n_i \) are \( \lambda \)-definable.

**Proof.** They are \( \lambda \)-defined by the following terms:
\[
\text{Zero} \equiv \lambda a. \lambda f x. x
\]
\[
\text{Succ} \equiv \lambda a. \lambda f x. f(ax)
\]
\[
\text{Proj}_n^i \equiv \lambda x_0 \ldots x_{n-1}. x_i
\]
Lemma ldf.10. Suppose the \( k \)-ary function \( f \), and \( n \)-ary functions \( g_0, \ldots, g_{k-1} \), are \( \lambda \)-definable by terms \( F \), \( G_0 \), \ldots, \( G_k \), and \( h \) is defined from them by composition. Then \( H \) is \( \lambda \)-definable.

Proof. \( h \) can be \( \lambda \)-defined by the term

\[
H \equiv \lambda x_0 \ldots x_{n-1}. F(G_0 x_0 \ldots x_{n-1}) \ldots (G_{k-1} x_0 \ldots x_{n-1})
\]

We leave verification of this fact as an exercise.

Problem ldf.5. Complete the proof of Lemma ldf.10 by showing that \( H^{n_0, \ldots, n_{n-1}} \rightarrow h(n_0, \ldots, n_{n-1}) \).

Note that Lemma ldf.10 did not require that \( f \) and \( g_0, \ldots, g_{k-1} \) are primitive recursive; it is only required that they are total and \( \lambda \)-definable.

Lemma ldf.11. Suppose \( f \) is an \( n \)-ary function and \( g \) is an \( n+2 \)-ary function, they are \( \lambda \)-definable by terms \( F \) and \( G \), and the function \( h \) is defined from \( f \) and \( g \) by primitive recursion. Then \( h \) is also \( \lambda \)-definable.

Proof. Recall that \( h \) is defined by

\[
\begin{align*}
    h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\
    h(x_1, \ldots, x_n, y + 1) &= h(x_1, \ldots, x_n, y, h(x_1, \ldots, x_n, y)).
\end{align*}
\]

Informally speaking, the primitive recursive definition iterates the application of the function \( h \) \( y \) times and applies it to \( f(x_1, \ldots, x_n) \). This is reminiscent of the definition of Church numerals, which is also defined as a iterator.

For simplicity, we give the definition and proof for a single additional argument \( x \). The function \( h \) is \( \lambda \)-defined by:

\[
H \equiv \lambda x. \lambda y. \text{Snd}(D\langle 0, F x \rangle)
\]

where

\[
D \equiv \lambda p. \langle \text{Succ}(Fst \ p), (Gx(Fst \ p)(\text{Snd} \ p)) \rangle
\]

The iteration state we maintain is a pair, the first of which is the current \( y \) and the second is the corresponding value of \( h \). For every step of iteration we create a pair of new values of \( y \) and \( h \); after the iteration is done we return the second part of the pair and that’s the final \( h \) value. We now prove this is indeed a representation of primitive recursion.

We want to prove that for any \( n \) and \( m \), \( H^{\pi \text{mm}} \rightarrow h(n, m) \). To do this we first show that if \( D_n \equiv D[\pi/x] \), then \( D^n_n(\langle 0, F \pi \rangle) \rightarrow (\text{mm}, h(n, m)) \). We proceed by induction on \( m \).

If \( m = 0 \), we want \( D^n_n(\langle 0, F \pi \rangle) \rightarrow (\langle 0, h(n, 0) \rangle) \). But \( D^n_n(\langle 0, F \pi \rangle) \) just is \( (\langle 0, F \pi \rangle) \).

Since \( F \) \( \lambda \)-defines \( f \), this reduces to \( (\langle 0, f(n) \rangle) \), and since \( f(n) = h(n, 0) \), this is \( (\langle 0, h(n, 0) \rangle) \).
Now suppose that $D^m_m(\overline{0}, F \overline{n}) \to \langle \overline{m}, \overline{h}(n, m) \rangle$. We want to show that $D^{m+1}_m(\overline{0}, F \overline{n}) \to \langle \overline{m+1}, \overline{h}(n, m+1) \rangle$.

$D^{m+1}_m(\overline{0}, F \overline{n}) \equiv D_n(D^m_m(\overline{0}, F \overline{n}))$

$\to D_n(\overline{m}, \overline{h}(n, m))$ (by IH)

$\equiv (\lambda p. (\text{Succ}(\text{Fst}(p)), (\text{G}(\text{Fst}(p))(\text{Snd}(p)))(\overline{m}, \overline{h}(n, m)))$

$\to (\text{Succ}(\text{Fst}(\overline{m}, \overline{h}(n, m)))(\overline{m}, \overline{h}(n, m)))$

$\to \langle \text{Succ}\overline{m}, (\text{G}(\text{Fst}(\text{Snd}(\overline{m}, \overline{h}(n, m))))(\overline{m}, \overline{h}(n, m))) \rangle$

$\to \langle \overline{m+1}, \overline{g}(n, m, \overline{h}(n, m)) \rangle$

Since $g(n, m, \overline{h}(n, m)) = \overline{h}(n, m+1)$, we are done.

Finally, consider

$$H \overline{n} \overline{m} \equiv \lambda x. \lambda y. (\text{Snd}(y(\lambda p. (\text{Succ}(\text{Fst}(p)), (\text{G}x(\text{Fst}(p))(\text{Snd}(p)))))(\overline{m}, F \overline{x}))$$

$$\overline{n} \overline{m}$$

$\to \text{Snd}(\overline{m} (\lambda p. (\text{Succ}(\text{Fst}(p)), (\text{G}(\text{Fst}(p))(\text{Snd}(p)))))(\overline{m}, F \overline{n}))$

$\equiv \text{Snd}(\overline{m} D_n(\overline{0}, F \overline{n}))$

$\to \text{Snd}(D^m_n(\overline{0}, F \overline{n}))$

$\to \text{Snd}(\overline{m}, \overline{h}(n, m))$

$\to \overline{h}(n, m)$. \qed

**Proposition ldf.12.** Every primitive recursive function is $\lambda$-definable.

**Proof.** By Lemma ldf.9, all basic functions are $\lambda$-definable, and by Lemma ldf.10 and Lemma ldf.11, the $\lambda$-definable functions are closed under composition and primitive recursion. \qed

**ldf.6 Fixpoints**

Suppose we wanted to define the factorial function by recursion as a term $\text{Fac}$ with the following property:

$$\text{Fac} \equiv \lambda n. \text{IsZero } n \overline{1}(\text{Mult } n(\text{Fac}(\text{Pred } n)))$$

That is, the factorial of $n$ is 1 if $n = 0$, and $n$ times the factorial of $n - 1$ otherwise. Of course, we cannot define the term $\text{Fac}$ this way since $\text{Fac}$ itself occurs in the right-hand side. Such recursive definitions involving self-reference are not part of the lambda calculus. Defining a term, e.g., by $\text{Mult} \equiv \lambda ab. a(\text{Add } a)0$
only involves previously defined terms in the right-hand side, such as Add. We can always remove Add by replacing it with its defining term. This would give the term Mult as a pure lambda term; if Add itself involved defined terms (as, e.g., Add' does), we could continue this process and finally arrive at a pure lambda term.

However this is not true in the case of recursive definitions like the one of Fac above. If we replace the occurrence of Fac on the right-hand side with the definition of Fac itself, we get:

\[
\text{Fac} \equiv \lambda n. \text{IsZero} \; n \; \top \; \\
(Mult \; n((\lambda n. \text{IsZero} \; n \; \top \; (\text{Mult} \; n \; (\text{Fac} \; (\text{Pred} \; n)))))(\text{Pred} \; n))
\]

and we still haven’t gotten rid of Fac on the right-hand side. Clearly, if we repeat this process, the definition keeps growing longer and the process never results in a pure lambda term. Thus this way of defining factorial (or more generally recursive functions) is not feasible.

The recursive definition does tell us something, though: If \( f \) were a term representing the factorial function, then the term

\[
\text{Fac'} \equiv \lambda g. \lambda n. \text{IsZero} \; n \; \top \; (\text{Mult} \; n \; (g(\text{Pred} n)))
\]

applied to the term \( f \), i.e., \( \text{Fac'} \; f \), also represents the factorial function. That is, if we regard \( \text{Fac'} \) as a function accepting a function and returning a function, the value of \( \text{Fac'} \; f \) is just \( f \), provided \( f \) is the factorial. A function \( f \) with the property that \( \text{Fac'} \; f \; \beta \equiv f \) is called a fixpoint of \( \text{Fac'} \). So, the factorial is a fixpoint of \( \text{Fac'} \).

There are terms in the lambda calculus that compute the fixpoints of a given term, and these terms can then be used to turn a term like \( \text{Fac'} \) into the definition of the factorial.

**Definition ldf.13.** The \( \text{Y-combinator} \) is the term:

\[
Y \equiv (\lambda u x. x (u u x))(\lambda u x. x (u u x)).
\]

**Theorem ldf.14.** \( Y \) has the property that \( Y \; g \rightarrow g(Y \; g) \) for any term \( g \). Thus, \( Y \; g \) is always a fixpoint of \( g \).

*Proof. Let’s abbreviate \( (\lambda u x. x (u u x)) \) by \( U \), so that \( Y \equiv U U \). Then

\[
Y \; g \equiv (\lambda u x. x (u u x)) U \; g \\
\rightarrow (\lambda x. x (U U x)) g \\
\rightarrow g(U U g) \equiv g(Y \; g).
\]

Since \( g(Y \; g) \) and \( Y \; g \) both reduce to \( g(Y \; g) \), \( g(Y \; g) \; \beta \equiv Y \; g \), so \( Y \; g \) is a fixpoint of \( g \). \qed
Of course, since \( Yg \) is a redex, the reduction can continue indefinitely:

\[
Yg \rightarrow g(Yg) \\
\rightarrow g(g(Yg)) \\
\rightarrow g(g(g(Yg))) \\
\ldots
\]

So we can think of \( Yg \) as \( g \) applied to itself infinitely many times. If we apply \( g \) to it one additional time, we—so to speak—aren’t doing anything extra; \( g \) applied to \( g \) applied infinitely many times to \( Yg \) is still \( g \) applied to \( Yg \) infinitely many times.

Note that the above sequence of \( \beta \)-reduction steps starting with \( Yg \) is infinite. So if we apply \( Yg \) to some term, i.e., consider \((Yg)N\), that term will also reduce to infinitely many different terms, namely \((g(Yg))N, (g(g(Yg)))N, \ldots\).

It is nevertheless possible that some other sequence of reduction steps does terminate in a normal form.

Take the factorial for instance. Define \( \text{Fac} \) as \( Y\text{Fac}' \) (i.e., a fixpoint of \( \text{Fac}' \)). Then:

\[
\begin{align*}
\text{Fac} \ 3 & \rightarrow Y\text{Fac}' \ 3 \\
& \rightarrow \text{Fac}'(Y\text{Fac}') \ 3 \\
& \equiv (\lambda x. \lambda n. \text{IsZero } n \ I \ (\text{Mult } n \ (x(Pred \ n)))) \ \text{Fac} \ 3 \\
& \rightarrow \text{IsZero } 3 \ I \ (\text{Mult } 3 \ (\text{Fac}(\text{Pred } 3))) \\
& \rightarrow \text{Mult } 3 \ (\text{Fac } 2).
\end{align*}
\]

Similarly,

\[
\begin{align*}
\text{Fac} \ 2 & \rightarrow \text{Mult } 2 \ (\text{Fac } 1) \\
\text{Fac} \ 1 & \rightarrow \text{Mult } 1 \ (\text{Fac } 0)
\end{align*}
\]

but

\[
\begin{align*}
\text{Fac} \ 0 & \rightarrow \text{Fac}'(Y\text{Fac}') \ 0 \\
& \equiv (\lambda x. \lambda n. \text{IsZero } n \ I \ (\text{Mult } n \ (x(Pred \ n)))) \ \text{Fac} \ 0 \\
& \rightarrow \text{IsZero } 0 \ I \ (\text{Mult } 0 \ (\text{Fac}(\text{Pred } 0))) \\
& \rightarrow I.
\end{align*}
\]

So together

\[
\text{Fac} \ 3 \rightarrow \text{Mult } 3 \ (\text{Mult } 2 \ (\text{Mult } 1)).
\]

What goes for \( \text{Fac}' \) goes for any recursive definition. Suppose we have a recursive equation

\[
g \ x_1 \ldots x_n \xrightarrow{\beta} N
\]

\[\lambda\text{-definability rev. 788b9aa (2022-03-22) by OLP / CC–BY}\]
where $N$ may contain $g$ and $x_1, \ldots, x_n$. Then there is always a term $G \equiv (Y \lambda g. \lambda x_1 \ldots x_n. N)$ such that

$$G x_1 \ldots x_n \beta \equiv N[G/g].$$

For by the fixpoint theorem,

$$G \equiv (Y \lambda g. \lambda x_1 \ldots x_n. N) \Rightarrow \lambda g. \lambda x_1 \ldots x_n. N(Y \lambda g. \lambda x_1 \ldots x_n. N) \equiv (\lambda g. \lambda x_1 \ldots x_n. N) G$$

and consequently

$$G x_1 \ldots x_n \Rightarrow (\lambda g. \lambda x_1 \ldots x_n. N) G x_1 \ldots x_n$$
$$\Rightarrow (\lambda x_1 \ldots x_n. N[G/g]) x_1 \ldots x_n$$
$$\Rightarrow N[G/g].$$

The $Y$ combinator of Definition ldf.13 is due to Alan Turing. Alonzo Church had proposed a different version which we’ll call $Y_C$:

$$Y_C \equiv \lambda g. (\lambda x. g(xx))(\lambda x. g(xx)).$$

Church’s combinator is a bit weaker than Turing’s in that $Y g \beta \equiv g(Y g)$ but not $Y g \Rightarrow g(Y g)$. Let $V$ be the term $\lambda x. g(xx)$, so that $Y_C \equiv \lambda g. VV$. Then

$$VV \equiv (\lambda x. g(xx))V \Rightarrow g(VV) \text{ and thus}$$
$$Y_C g \equiv (\lambda g. VV)g \Rightarrow VV \Rightarrow g(VV), \text{ but also}$$
$$g(Y_C g) \equiv g((\lambda g. VV)g) \Rightarrow g(VV).$$

In other words, $Y_C g$ and $g(Y_C g)$ reduce to a common term $g(VV)$; so $Y_C g \beta \equiv g(Y_C g)$. This is often enough for applications.

**ldf.7 Minimization**

The general recursive functions are those that can be obtained from the basic functions zero, succ, $P^n$ by composition, primitive recursion, and regular minimization. To show that all general recursive functions are $\lambda$-definable we have to show that any function defined by regular minimization from a $\lambda$-definable function is itself $\lambda$-definable.

**Lemma ldf.15.** If $f(x_1, \ldots, x_k, y)$ is regular and $\lambda$-definable, then $g$ defined by

$$g(x_1, \ldots, x_k) = \mu y \ f(x_1, \ldots, x_k, y) = 0$$

is also $\lambda$-definable.
Proof. Suppose the lambda term $F$ $\lambda$-defines the regular function $f(x, y)$. To $\lambda$-define $h$ we use a search function and a fixpoint combinator:

$$\text{Search} \equiv \lambda g. \lambda f x y. \text{IsZero}(f x y) \, g x (\text{Succ} y)$$

$$H \equiv \lambda x. (\text{Search}) F x \bar{u},$$

where $Y$ is any fixpoint combinator. Informally speaking, Search is a self-referencing function: starting with $y$, test whether $f x y$ is zero: if so, return $y$, otherwise call itself with $\text{Succ} y$. Thus $(Y \text{Search}) F n_1 \ldots n_k \bar{u}$ returns the least $m$ for which $f(n_1, \ldots, n_k, m) = 0$.

Specifically, observe that

$$(Y \text{Search}) F n_1 \ldots n_k \bar{u} \rightarrow m$$

if $f(n_1, \ldots, n_k, m) = 0$, or

$$\rightarrow (Y \text{Search}) F n_1 \ldots n_k \bar{u} \rightarrow m + 1$$

otherwise. Since $f$ is regular, $f(n_1, \ldots, n_k, y) = 0$ for some $y$, and so

$$(Y \text{Search}) F n_1 \ldots n_k \bar{u} \rightarrow h(n_1, \ldots, n_k). \quad \square$$

**Proposition ldf.16.** Every general recursive function is $\lambda$-definable.

Proof. By Lemma ldf.9, all basic functions are $\lambda$-definable, and by Lemma ldf.10, Lemma ldf.11, and Lemma ldf.15, the $\lambda$-definable functions are closed under composition, primitive recursion, and regular minimization. \hfill $\square$

### ldf.8 Partial Recursive Functions are $\lambda$-Definable

Partial recursive functions are those obtained from the basic functions by composition, primitive recursion, and unbounded minimization. They differ from general recursive function in that the functions used in unbounded search are not required to be regular. Not requiring regularity means that functions defined by minimization may sometimes not be defined.

At first glance it might seem that the same methods used to show that the (total) general recursive functions are all $\lambda$-definable can be used to prove that all partial recursive functions are $\lambda$-definable. For instance, the composition of $f$ with $g$ is $\lambda$-defined by $\lambda x. F(Gx)$ if $f$ and $g$ are $\lambda$-defined by terms $F$ and $G$, respectively. However, when the functions are partial, this is problematic. When $g(x)$ is undefined, meaning $Gx$ has no normal form. In most cases this means that $F(Gx)$ has no normal forms either, which is what we want. But consider when $F$ is $\lambda x. \lambda y. y$, in which case $F(Gx)$ does have a normal form ($\lambda y. y$).

This problem is not insurmountable, and there are ways to $\lambda$-define all partial recursive functions in such a way that undefined values are represented...
by terms without a normal form. These ways are, however, somewhat more
complicated and less intuitive than the approach we have taken for general
recursive functions. We record the theorem here without proof:

**Theorem ldf.17.** All partial recursive functions are \(\lambda\)-definable.

\[ \text{thm:lambda-computable} \]

\[ \text{lam:dfl:ldr} \]

\[ \text{dfl.9 \ } \lambda\text{-Definable Functions are Recursive} \]

Not only are all partial recursive functions \(\lambda\)-definable, the converse is true,
too. That is, all \(\lambda\)-definable functions are partial recursive.

**Theorem dfl.18.** If a partial function \(f\) is \(\lambda\)-definable, it is partial recursive.

**Proof.** We only sketch the proof. First, we arithmetize \(\lambda\)-terms, i.e., systema-
tically assign Gödel numbers to \(\lambda\)-terms, using the usual power-of-primes coding
of sequences. Then we define a partial recursive function \(\text{normalize}(t)\) operating
on the Gödel number \(t\) of a lambda term as argument, and which returns
the Gödel number of the normal form if it has one, or is undefined otherwise.
Then define two partial recursive functions \(\text{toChurch}\) and \(\text{fromChurch}\) that
maps natural numbers to and from the Gödel numbers of the corresponding
Church numeral.

Using these recursive functions, we can define the function \(f\) as a par-
tial recursive function. There is a \(\lambda\)-term \(F\) that \(\lambda\)-defines \(f\). To compute
\(f(n_1, \ldots, n_k)\), first obtain the Gödel numbers of the corresponding Church
numerals using \(\text{toChurch}(n_i)\), append these to \(\#F\) to obtain the Gödel number of
the term \(F\bar{n}_1 \cdots \bar{n}_k\). Now use \(\text{normalize}\) on this Gödel number. If \(f(n_1, \ldots, n_k)\)
is defined, \(F\bar{n}_1 \cdots \bar{n}_k\) has a normal form (which must be a Church numeral),
and otherwise it has no normal form (and so

\[ \text{normalize}(\#F\bar{n}_1 \cdots \bar{n}_k) \]

is undefined). Finally, use \(\text{fromChurch}\) on the Gödel number of the normalized
term.

\[ \square \]

**Photo Credits**
Bibliography