Functional Programming Lecture Notes

Thomas Dinsdale-Young

1st September, 2017

Functional programming is a discipline of computer science where *computations* are represented by *functions*. Functions in this sense are not necessarily those mathematical objects studied in set theory or analysis: that is, relations between two sets where each value in the first set is mapped to a single value in the second. What they principally have in common is that we can *apply* functions to arguments.

1 SKI

In the simplest kind of functional programming language, we think of everything as being a function. That is, if we have any two terms in our language, say M and N, then we can write a new term that is the application of M to N: MN. For instance, we might have terms 'sin' and 'π', in which case we also have the term '$\sin \pi$' representing the application of 'sin' to 'π'. (In mathematics, it is common to write application as '$\sin(\pi)$'. In computer science, these parentheses are not required, as it is unambiguous that 'π' is the argument to which the function 'sin' is applied.) We haven't said what this *means* yet — though we might wish it to mean 0. Note that we can also write terms like '$\sin \sin$' or '$\pi \sin$' which do not have an obvious mathematical meaning.

Since applications are terms, we can also apply them, as in '$(MN)P$', or use them as arguments, as in $M(NP)$. Note that these are *not* the same thing. For instance, we can think of '+' as a function that takes a number and returns a function that adds this number to its argument. In this case '(+ 1) 2' makes sense. However, '1' doesn't have a meaning as a function, so '+ (1 2)' does not make sense. When we drop the parentheses and write MNP, we will always mean $(MN)P$.

Let us consider a language where we have only three basic terms: **S**, **K** and **I**. The syntax of the language is thus given by the grammar:

$$M, N ::= \mathbf{S} \mid \mathbf{K} \mid \mathbf{I} \mid MN$$

To give some meaning to the terms of our language, let's define some rules for rewriting them. We will define the judgement $M \to N$ between two terms, with the interpretation that "M rewrites to N" (or "M reduces to N"). We will give a number of rules that determine when the judgement holds. Each rule has a number of premisses (possibly zero) and a conclusion, with the premisses written above a line and the conclusion below. The interpretation is that if the premisses hold then the conclusion does also. The first two rules allow us to

rewrite subterms of applications:

$$\frac{X \to Y}{XZ \to YZ} \qquad \frac{X \to Y}{ZX \to ZY}$$

These rules are parametrised by variables X, Y, Z, which can be substituted (consistently) with concrete terms of the language to give particular instances of the rules, such as:

$$\frac{\mathbf{IS} \to \mathbf{S}}{\mathbf{ISK} \to \mathbf{SK}} \qquad \frac{\mathbf{IK} \to \mathbf{SS}}{\mathbf{K}(\mathbf{IK}) \to \mathbf{K}(\mathbf{SS})}$$

By themselves, these two rules don't allow us to deduce any judgements, since the conclusion only holds if the premiss holds. We therefore need some axioms to provide the base cases for our derivations. We give one axiom for each of the basic terms. Firstly, **I** should be a function that simply returns its argument:

$$\frac{}{\mathbf{I}X \to X}$$

The rule for **K** is that, given two arguments, it returns the first. (Put otherwise, it takes its argument to the constant function that returns this argument.)

$$\frac{}{\mathbf{K}XY \to X}$$

The rule for **S** is that, given three arguments, it applies the first to the third and the result of applying the second to the third:

$$\frac{}{\mathbf{S}XYZ \to XZ(YZ)}$$

Now we can say, for instance, that $\mathbf{ISK} \to \mathbf{SK}$ since it is justified by the derivation:

$$\frac{\dfrac{}{\mathbf{IS} \to \mathbf{S}}}{\mathbf{ISK} \to \mathbf{SK}}$$

Note that our rewriting rule is *non-deterministic* in that there are multiple possible ways of rewriting some terms:

$$\frac{\dfrac{}{\mathbf{I}(\mathbf{KK}) \to \mathbf{KK}}}{\mathbf{I}(\mathbf{KK})(\mathbf{I}(\mathbf{KI})) \to \mathbf{KK}(\mathbf{I}(\mathbf{KI}))} \qquad \frac{\dfrac{}{\mathbf{I}(\mathbf{KI}) \to \mathbf{KI}}}{\mathbf{I}(\mathbf{KK})(\mathbf{I}(\mathbf{KI})) \to \mathbf{I}(\mathbf{KK})(\mathbf{KI})}$$

Note also that there are some terms that cannot be rewritten, for instance '**I**', '**KI**', '**S**(**SK**)(**KS**)'. These terms are said to be *irreducible*, and we write $M \not\to$ to indicate that M is irreducible. We write $M \to^* N$ if there is a finite sequence of rewrites

$$M \to M_1 \to M_2 \to \ldots \to N$$

More formally, we can define this judgement by the following rules:

$$\frac{}{X \to^* X} \qquad \frac{X \to^* Y \qquad Y \to Z}{X \to^* Z}$$

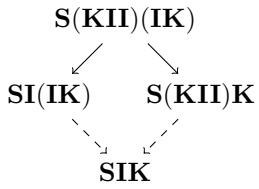
If $M \to^* N \not\to$, then we say that N is a normal form of M.

1.1 Confluence

An interesting and important property that term rewriting systems (such as $\rightarrow$) may have is *confluence*. A system is confluent if, for any x, y, z with $x \rightarrow^* y$ and $x \rightarrow^* z$, there exists some w such that $y \rightarrow^* w$ and $z \rightarrow^* w$. In a sense, this means that whatever strategy we choose to rewrite a term will give the same result. (I say "in a sense" because terms may still have both finite and infinite rewrite paths.)

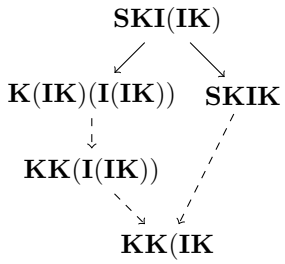
It turns out that the **SKI** combinator system is confluent. To see intuitively why this is the case, suppose we have a term x that rewrites in two different ways (to y and to z).

It could be that these rewrites involve different parts of the term, for example:



In general, this is the case where we have $x = C[x_1 x_2]$ for some context C and terms x_1, x_2 with $x_1 \rightarrow w_1$ and $x_2 \rightarrow w_2$ such that $y = C[w_1 x_2]$ and $z = C[x_1 w_2]$. Then we can simply pick $w = C[w_1 w_2]$.

The trickier case is when the rewrites involve overlapping parts of the term, for example:



In such cases there is some reducible subexpression (i.e. of the form $\mathbf{I}M$, $\mathbf{K}MN$ or $\mathbf{S}MNP$) such that the other reduction occurs in one of its arguments. Note that it is important that a reducible expression (or *redex* for short) can only be reduced in one way. For instance, if we had a combinator **C** with rules $\mathbf{C}MN \rightarrow M$ and $\mathbf{C}MN \rightarrow N$, we could not expect the system to be confluent. In the above example, reducing the outer redex **SKI**(**IK**) duplicated the inner redex (**IK**). It is also possible for an inner redex to be eliminated completely, as in $\mathbf{KI}(\mathbf{IK}) \rightarrow \mathbf{I}$. After reducing the outer redex, we can reduce all copies of the inner redex, though it may take a number of steps (0, 1 or 2). If we first reduce the inner redex, the outer redex will still exist, but with different arguments; we can reduce that to get to the same result.

While this should give some intuition for why the system is confluent, it is a bit trickier to prove it formally.

Note that confluence implies that if a term has a normal form then that normal form is unique. Suppose that $M \rightarrow^* N \not\rightarrow$ and $M \rightarrow^* N' \not\rightarrow$. By

confluence, there must be some N'' such that $N \to^* N''$ and $N' \to^* N''$. But since N and N' are irreducible, it must be that $N = N'' = N'$.

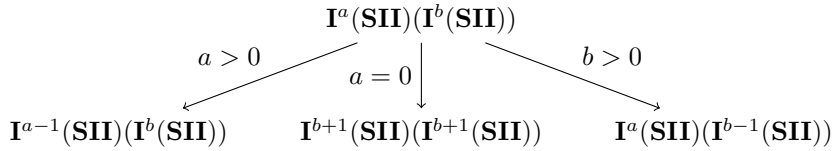
1.2 Normalisation

A term M is *normalising* if it is reducible to an irreducible term: $M \to^* N \not\to$. A term M is *strongly normalising* if it has no infinite reduction sequence; i.e. no infinite sequence of the form $M \to M_1 \to M_2 \to \dots$. (Naturally, a term that is strongly normalising will be normalising.)

For **SKI**, not every term is strongly normalising, or even normalising. For instance, consider the term $\mathbf{\Omega} = \mathbf{SII}(\mathbf{SII})$. This term has an infinite reduction sequence:

$$\mathbf{SII}(\mathbf{SII}) \longrightarrow \mathbf{I}(\mathbf{SII})(\mathbf{I}(\mathbf{SII})) \longrightarrow \mathbf{SII}(\mathbf{I}(\mathbf{SII}))$$

It is therefore not strongly normalising. Moreover, every term of the form $\mathbf{I}^a(\mathbf{SII})(\mathbf{I}^b(\mathbf{SII}))$ is reducible, and every term it reduces to is also of that form.



Therefore $\mathbf{\Omega}$ (which is the special case when $a = b = 0$) has no finite reduction sequence, and so is not normalising.

1.3 History

The **SKI**-calculus that we have seen above is an instance of combinatory logic, which was introduced by Moses Schönfinkel and Haskell Curry in the 1920s. While the **SKI** combinators are often treated as the canonical basis of combinatory logic, Curry's system had the following combinators:

$$\begin{aligned} \mathbf{B}XYZ &\to X(YZ) \\ \mathbf{C}XYZ &\to XZY \\ \mathbf{K}XY &\to X \\ \mathbf{W}XY &\to XYY \end{aligned}$$

It is possible to encode this system in **SKI** by defining these in terms for the **S**, **K** and **I** combinators. (The encoded versions may take a different number of reduction steps, however.)

It is remarkable that the **SKI**-calculus is Turing-complete: any computable function can be represented by an **SKI**-term. It is possible to represent a Turing machine as a term that takes a representation of the input tape and normalises to a term which is the output of the machine exactly when the Turing machine terminates.

1.4 Exercises

Exercise 1. Let $\mathbf{W} = \mathbf{SS}(\mathbf{SK})$. Show that $\mathbf{W}XY \to^* XYY$.

Exercise 2. Find an **SKI**-term that is normalising, but not strongly normalising. Justify your answer by showing both a finite reduction sequence to a normal form and an infinite reduction sequence.

Exercise 3 (Encoding datatypes with functions)**.** **SKI** doesn't have basic datatypes that we are used to in common programming languages, such as booleans and integers. However, it is possible to represent these as functions. Booleans can be represented as functions of two arguments: **true** returns the first argument and **false** returns the second. Find **SKI**-terms to represent booleans in this way. Find a term **if** that takes three arguments and returns the second when the first argument is **true** and the third when the first argument is **false**.

Exercise 4. Find an **SKI**-term **B** that composes two functions. That is $\mathbf{B}XYZ \to^* X(YZ)$.

2 Lambda Calculus

Constructing functions from the **S**, **K** and **I** combinators alone is quite challenging. To do so effectively, we typically want to build a collection of helper functions that we can use to define even more complicated functions. For example, it is convenient to have a function that flips the arguments of a function:

$$\mathbf{C}XYZ \to^* XZY$$

But even such a simple function is difficult to define in **SKI**:

$$\mathbf{C} = \mathbf{S}(\mathbf{S}(\mathbf{K}(\mathbf{S}(\mathbf{KS})\mathbf{K}))\mathbf{S})(\mathbf{KK})$$

The *lambda calculus* (or λ-calculus) makes it much easier to define these sorts of functions directly. Terms of the lambda calculus are defined by the following grammar:

$$M, N ::= x \mid MN \mid \lambda x.\, M$$

where $x \in \mathbf{Var}$ ranges over an infinite set of variable names.

As with the **SKI** calculus, we have application in the language (and the same notational conventions apply). We also have variables x, and *lambda abstraction* $\lambda x.\, M$. The intuition for $\lambda x.\, M$ is the function that takes x to M. For example, a function that doubles its input might be $\lambda x.\, x + x$. Staying within the syntax, we might define $\mathbf{I} = \lambda x.\, x$, $\mathbf{K} = \lambda x.\, \lambda y.\, x$ and $\mathbf{S} = \lambda x.\, \lambda y.\, \lambda z.\, xz(yz)$.

In a given term, an occurrence of a variable (i.e. a subterm x) may be either *free* or *bound*. A bound variable is one that refers to a particular lambda abstraction (of which it is a subterm), while a free variable does not refer to any lambda abstraction. A bound variable is always bound by the closest containing lambda abstraction for the given variable. For example, in $\lambda x.\, xy$, the variable x is bound by the lambda abstraction, while y occurs free. We consider terms to be equivalent if they differ only in the names of their bound variables (this is called α-equivalence). Thus $\lambda x.\, xy$ is equivalent to $\lambda z.\, zy$; however, $\lambda y.\, yy$ is

not equivalent to either of these, since y here is bound. Moreover, $\lambda x.\, \lambda y.\, y$ is equivalent to $\lambda y.\, \lambda y.\, y$, since the variable y is bound by the innermost λ; it is not equivalent to $\lambda y.\, \lambda x.\, y$.

The core rewriting judgement for λ-calculus is called β-reduction:

$$\frac{}{(\lambda x.\, M)N \to_\beta M[N/x]}$$

β-reduction is defined in terms of substitution $M[N/x]$ (M with N for x) which replaces occurrences of x in M by N. However, there are two important issues we want to avoid with substitution.

The first issue is that we only want to substitute *free* occurrences of a variable. This means that we want $(\lambda x.\, M)[N/x] = \lambda x.\, M$. The reason behind this is that occurrences of x in M refer to the variable that is bound by the λ.

In particular, supposing that y is not free in M, then we would have $\lambda y.\, (M[y/x])$ equivalent to $\lambda x.\, M$. If we substitute in equivalent terms, we want the results to be the equivalent. If substitution replaced bound occurrences of x, this would not be the case.

The second issue is that we wish to avoid variable capture: a variable that is free in N should also be free in $M[N/x]$. Specifically, if x is free in N, we would not want $(\lambda x.\, x)[N/x]$ to be $\lambda x.\, N$. In particular, this does not respect renaming of bound variables: $(\lambda z.\, z)[N/x] = \lambda z.\, z$, which is not equivalent.

We can thus define substitution as follows:

$$\begin{aligned}
x[N/x] &= N \\
y[N/x] &= y \\
(M_1 M_2)[N/x] &= (M_1[N/x])(M_2[N/x]) \\
(\lambda x.\, M)[N/x] &= \lambda x.\, M \\
(\lambda y.\, M)[N/x] &= \lambda z.\, (M[z/y][N/x]) && z \text{ not free in } N
\end{aligned}$$

(Note that, while there is not strictly a unique result for substitution, the result is unique up to α-equivalence.)

As with the **SKI**-calculus, we can reduce terms on either side of an application:

$$\frac{X \to_\beta Y}{XZ \to_\beta YZ} \qquad\qquad \frac{X \to_\beta Y}{ZX \to_\beta ZY}$$

It is also permissible to reduce under a binder:

$$\frac{X \to_\beta Y}{\lambda x.\, X \to_\beta \lambda x.\, Y}$$

Notation. By convention, the scope of a λ binding extends as far as possible. That is, in $\lambda x.\, xx$ both occurrences of x are bound; this is different to the term $(\lambda x.\, x)x$, where the right-most occurrence of x is free. When we have multiple lambda abstractions, we can combine them: $\lambda x, y, z.\, M$ is short-hand for $\lambda x.\, \lambda y.\, \lambda z.\, M$.

It is sometimes useful to consider not just β-reduction, but *β-equivalence* ($=_\beta$): the symmetric, reflexive, transitive closure of $\to_\beta$. Two terms are β-equivalent if there is a path of forward and backward β-reductions between them. If terms are β-equivalent, we may also say that they are *β-convertible*.

2.1 Evaluation Strategies

As we saw with the **SKI**-calculus, reduction can be nondeterministic. Beta-reduction in lambda calculus is confluent. (This was proved by Alonzo Church and J. Barkley Rosser in 1936, and is known as the Church-Rosser theorem.) Yet to actually implement the calculus, we want to have a deterministic strategy. The two simplest evaluation strategies are *call-by-value* and *call-by-name*.

In call-by-value evaluation, a redex $(\lambda x.\, M)N$ is only reduced once the term N has been reduced to a value. In call-by-name evaluation, a redex $(\lambda x.\, M)N$ is reduced before any evaluation of N takes place.

Of course, we have not said exactly what constitutes a value. The simplest choice is that values are the irreducible terms (i.e. β-normal forms). An alternative is to consider all lambda abstractions $\lambda x.\, M$ as values, regardless of whether M is in normal form. In the former, for call-by-value evaluation of $(\lambda x.\, M)N$, first M would be reduced to normal form M', then N reduced to normal form N', then the substitution $M'[N'/x]$ would be reduced to normal form to give the result. In the latter, N would be reduced to a value N' and then $M[N'/x]$ would be evaluated. (This strategy may give us a value in the end which is not a β-normal form.)

The trade off between call-by-value and call-by-name is that, in the former, function arguments are always evaluated exactly once, while in the latter, arguments are evaluated as often as they are used. If an argument is used many times, then this can be a significant penalty for call-by-name; whereas if an argument is not used at all (e.g. it is only used under certain conditions) then this can be a significant advantage for call-by-name. In particular, when arguments are non-terminating (i.e. they cannot be reduced to a normal form) then call-by-name can still reach a normal form (if it exists), while call-by-value will not.

Call-by-need or *lazy evaluation* is another strategy, where arguments are evaluated when they are first used; the result is memoised, so that other copies of an argument do not recompute the result. Like call-by-name, if an argument is not used, then it is not evaluated. This strategy is used by the Haskell programming language. (Although this may seem like the best of both worlds, implementations can still pay a performance cost from storing memoised computations. Another consequence is that performance is less predictable than with call-by-value.)

One concern that is relevant to the choice of evaluation strategy, which, however, we do not see with the pure lambda calculus, is the issue of *side effects*. Suppose that we have a function **print**, which prints out its argument to the screen when it is evaluated:

$$\frac{}{\mathbf{print}M \xrightarrow{\text{print } M} M}$$

Under different evaluation strategies, the program $(\lambda x.\, M)(\mathbf{print}\text{“hello”})$ can have quite different effects. In call-by-value, the side effect of the **print** happens first, before any side effects of the function. In call-by-name, when and how many times the side effect happens depends on the body of the function M. In call-by-need, the side effect will happen at most once, but it may happen interleaved with the side effects of the function. Languages that allow side effects therefore typically use a call-by-value strategy, since this has the most

predictable semantics. (In particular, we do not need to know what the function does to know that its side effects happen after the argument's side effects.) Languages with lazy evaluation (such as Haskell) use additional techniques (such as Monads) to enforce explicit ordering of side-effects.

2.2 Church Encoding of Inductive Datatypes

The lambda calculus does not include the basic datatypes that are common in most programming languages, such as booleans and integers. However, it turns out that we can represent such datatypes purely using functions.

An inductive datatype is defined by a (recursive) grammar. For example, the booleans can be defined as

$$b ::= \textbf{true} \mid \textbf{false}$$

The natural numbers can be defined as

$$n ::= \mathsf{0} \mid \mathsf{S}n$$

(That is, $\mathsf{0}$ is a natural number, and for every natural number n, its successor $\mathsf{S}n$ is a natural number.) We can view each case of the grammar as a *constructor* for the datatype, which is a function that takes some number (possibly zero) of appropriate arguments and gives a term of the datatype. The booleans have two nullary constructors, **true** and **false**. The naturals have one nullary constructor, $\mathsf{0}$, and one unary constructor S.

We could add such constructors to the language. This would allow us to directly represent values such as **true**, **false**, $\mathsf{0}$, $\mathsf{S0}$, $\mathsf{S}(\mathsf{S0})$, etc. However, we can encode these values by *abstracting over the constructors*. So our boolean value **true** becomes $\lambda \mathit{true}, \mathit{false}.\, \mathit{true}$, and **false** becomes $\lambda \mathit{true}, \mathit{false}.\, \mathit{false}$. Similarly for the naturals, $\mathsf{0}$ becomes $\lambda o, s.\, o$; $\mathsf{S0}$ becomes $\lambda o, s.\, so$; $\mathsf{S}(\mathsf{S0})$ becomes $\lambda o, s.\, s(so)$; etc. (Note that we have to be consistent about the order of the constructors, but the particular choice is not important.) This approach to encoding datatypes is called Church encoding (after Alonzo Church, who introduced the lambda calculus).

We can also define encodings for datatypes over lambda terms. For instance pairs:

$$p ::= \mathsf{pair}MN$$

We represent $\mathsf{pair}MN$ as $\lambda p.\, pMN$.

For lists:

$$l ::= \mathsf{nil} \mid \mathsf{cons}\ M\ l$$

We represent nil as $\lambda n, c.\, n$, $\mathsf{cons}\ M\ \mathsf{nil}$ as $\lambda nc.\, cMn$, etc.

One way of justifying these representations as good is that they can easily be converted into any other representation you desire by simply applying them to the constructors for that representation. Moreover, the Church encoding allows us to define functions by *primitive recursion*.

On the natural numbers, a primitive recursive function h is defined by giving cases f and g, so that

$$h(\mathsf{0}) = f$$
$$h(\mathsf{S}y) = g(hy)$$

Given the Church encoding, we can simply define

$$h = \lambda x.\, x f g$$

Many computable functions can be defined using primitive recursion. (A notable exception is Ackermann's function.) It is left as an exercise to define basic functions for Church encodings.

2.3 Fixed-point Combinators and Recursion

A *fixed point* of a function f is a value x such that $f(x) = x$. In the lambda calculus, a *fixed-point combinator* is a term that, for a given function, computes a fixed-point (up to β-equivalence). A fixed-point combinator is thus a term $\mathbf{Y}$ such that

$$\mathbf{Y}g =_\beta g(\mathbf{Y}g)$$

One such combinator, called the $\mathbf{Y}$ combinator, discovered by Curry, is:

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

A recursive function is one that is defined in terms of itself. For example, the factorial function might be defined along the following lines:

$$\mathit{fac} = \lambda x.\, \mathbf{if}(\mathbf{isZero}\ x)(1)(x \times \mathit{fac}(x-1))$$

We can think of this equation as showing us how to define a factorial function out of a factorial function. That is, we can define a function that takes a factorial function to a factorial function:

$$\lambda \mathit{fac}.\, \lambda x.\, \mathbf{if}(\mathbf{isZero}\ x)(1)(x \times \mathit{fac}(x-1))$$

A fixed point of this function will satisfy the definition of the factorial function. Therefore, we can define the factorial function as

$$\mathbf{fac} = \mathbf{Y}\lambda \mathit{fac}.\, \lambda x.\, \mathbf{if}(\mathbf{isZero}\ x)(1)(x \times \mathit{fac}(x-1))$$

2.4 Exercises

Exercise 5. Group the following lambda terms into equivalence classes with respect to α-equivalence (and notational conventions).

$$\lambda x, y.\, x(\lambda x.\, x)y \qquad \lambda y, x.\, x(\lambda x.\, x)y \qquad \lambda x, y.\, z(\lambda z.\, z)y \qquad \lambda y, x.\, y(\lambda z.\, z)x$$

$$\lambda x.\, (\lambda y.\, x(\lambda z.\, z))y \qquad \lambda x, y.\, x((\lambda z.\, z)y) \qquad \lambda x, z.\, x((\lambda x.\, x)z) \qquad \lambda y, z.\, yz$$

$$\lambda x, y.\, ((z(\lambda w.\, w))y)$$

Exercise 6. Compute the β-normal form of each of the following:

$$(\lambda x.\, x)(\lambda x, y.\, x)(\lambda x, y.\, y)(\lambda x, y.\, x)$$
$$(\lambda x, y, z.\, xy(yz))((\lambda x, y.\, x)(\lambda x, y, z.\, xz(yz)))(\lambda x, y.\, x)$$

Exercise 7. Define the functions **if**, **and**, **or** and **not** on the Church-encoded booleans, such that

if true $M\ N \to^* M$	**if false** $M\ N \to^* N$
and true true $\to^*$ **true**	**and true false** $\to^*$ **false**
and false true $\to^*$ **false**	**and false false** $\to^*$ **false**
or true true $\to^*$ **true**	**or true false** $\to^*$ **false**
or false true $\to^*$ **false**	**or false false** $\to^*$ **false**
not true $\to$ **false**	**not false** $\to$ **true**

where **true** $= \lambda \mathit{true}, \mathit{false}.\ \mathit{true}$ and **false** $= \lambda \mathit{true}, \mathit{false}.\ \mathit{false}$ are the Church booleans.

Exercise 8. Define arithmetic operators on the Church numerals (the encoding of natural numbers above) for: successor, addition, multiplication, predecessor (take predecessor of $\mathsf{0}$ to be $\mathsf{0}$) and subtraction (when $n < m$, $n - m$ should result in $\mathsf{0}$). (You may find it useful to use the former operators in the definition of the later ones, although some are straightforward without doing so. You may also find boolean operators helpful in some cases.) Using subtraction, then define an operator for less-than-or-equal (taking two numbers and returning a boolean). Using this, then define and equality operator.

Note that it is helpful to think of the Church numeral n as being a function that apply a given function n times to a given value. All of the above operations can (and should) be defined without recourse to a fixed-point combinator.

Exercise 9. Verify that the definition of the $\mathbf{Y}$ combinator indeed satisfies $\mathbf{Y}g = g(\mathbf{Y}g)$. Check that **fac** $3 \to^*_\beta 6$ (you need not unfold the encodings of numbers and booleans).