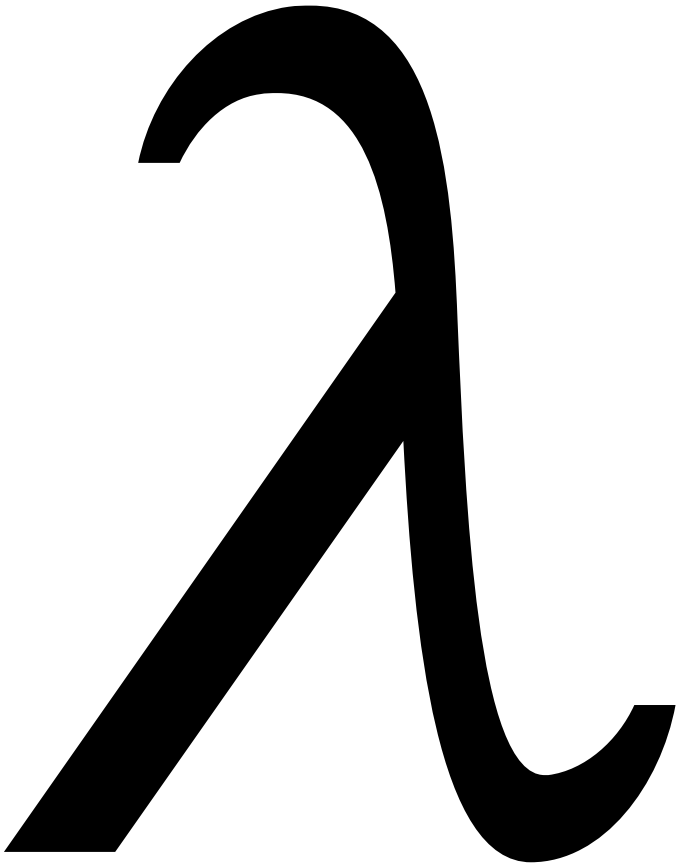
Types and λ-Calculus

Lecture Notes



Steven Ramsay

Revised 10/10/2018.

Contents

1. Introduction

The λ-calculus is a tiny language lying at the intersection of programming languages, the theory of computation and the foundations of mathematics. It earns this place by codifying the notion of *function* in the purest sense. In this first lecture, our aim is to get some informal intuitions about the λ-calculus and its type systems.

The function is a fundamental building block used throughout mathematics and computer science. Indeed, you have met many different kinds of functions in your studies already. In the first year you saw the truth-valued functions (predicates) of logic, the linear transformations of algebra, the isomorphisms of group theory, the continuous and differentiable functions of analysis, the Haskell-definable functions and so on.

Each area of computer science and mathematics has its own special language and notations, but the development of functions prescribed by the λ-calculus makes sense in all of them. The reason for this great generality is that the development presupposes only one thing — that abstract quantities are written using variables.

1.1 A syntax for functions

To the notion of *variable*, the λ-calculus adds another two syntactic forms: the *abstraction*, for introducing new functions, and the *application*, for eliminating old ones.

The abstraction. Suppose M is an expression denoting some object of interest. It doesn't matter what language this expression belongs to but, for the purposes of illustration, let's assume that M contains a variable x. Then, we can use this expression to describe a function of x, namely: the function that takes as input x and delivers as output M. In the λ-calculus, this function is written:

$$\lambda x.M \qquad \text{(ABSTRACTION)}$$

As we shall see, this way of describing a function — by abstracting over a variable in a given expression — is so basic that we can understand the behaviour of the function $\lambda x.M$ without understanding the behaviour of M. For now, though, let's consider some familiar examples:

$\lambda x.x^2 - 1$	The function that takes an integer input x, squares it and subtracts 1. Number theorists may write this function as $x \mapsto x^2 - 1$ or refer to it indirectly by $y = x^2 - 1$ or $f(x) = x^2 - 1$.
$\lambda x.\mathsf{reverse}\ x \mathbin{+\!\!+} x$	The function that takes a string input x and appends it to its own reversal. Haskell programmers may write this function as $\backslash x$ -> reverse x ++ x in a notation deliberately intended to resemble the λ-calculus.
$\lambda x.\emptyset$	The function that ignores its input and always returns the empty set. You may have seen this written as $x \mapsto \emptyset$ or referred to it indirectly by $f(x) = \emptyset$.

You can see from that last example that the abstraction $\lambda x.M$ is sensible even when M does not contain the variable x.

The application. Suppose M is an expression denoting some object of interest. We have seen that it makes sense to consider the function $(\lambda x.M)$ obtained from M by abstracting over x. Now suppose N is an expression denoting another object, of the same kind as x. Then, irrespective of the language that these expressions are written in, it surely makes sense to *apply* the function $\lambda x.M$ to the input N. In the λ-calculus, we describe the action of providing $\lambda x.M$ with an input N by writing:

$$(\lambda x.M)N \qquad \text{(APPLICATION)}$$

Here are some examples:

$(\lambda x.x^2 - 1)\,3$	The action of supplying $\lambda x.x^2 - 1$ with the input 3. A number theorist may write this as $f(3)$ if it is understood that f refers to the function defined by $f(x) = x^2 - 1$.
$(\lambda x.\mathsf{reverse}\ x \mathbin{+\!\!+} x)$ "d or even"	Supplying "d or even" as an input to the function $\lambda x.\mathsf{reverse}\ x \mathbin{+\!\!+} x$.
$(\lambda x.\emptyset)\ (y \cap z)$	The action of supplying $\lambda x.\emptyset$ with the input $y \cap z$.

Conversion. Suppose M and N are expressions denoting objects of interest. We have seen that it makes sense to consider the application $(\lambda x.M)N$ of the function $\lambda x.M$, obtained from M by abstracting over x, to the input N. Moreover, we already know the output of this function when applied to this input. Since $\lambda x.M$ is the function with input x that returns M, it follows that we can express the output by taking M and replacing in it every occurrence of x by N. Let's write the expression that results from this replacement as $M[N/x]$. Then, irrespective

of the language in which M and N are written, the output of $(\lambda x.M)N$ can be described by the expression $M[N/x]$. In the λ-calculus, we write this fact as:

$$(\lambda x.M)N =_\beta M[N/x] \qquad \text{(CONVERSION)}$$

The little β subscript on the equals reminds us that the expression on the left and the expression on the right are not identical, one has to do some computation, namely the replacement, to obtain the latter from the former. Returning to our examples, we can write:

$$\begin{aligned} (\lambda x.x^2-1)\,3 &=_\beta 3^2-1 \\ (\lambda x.\text{reverse } x +\!\!+ x)\text{“d or even”} &=_\beta \text{reverse “d or even” } +\!\!+ \text{“d or even”} \\ (\lambda x.\emptyset)(y \cap z) &=_\beta \emptyset \end{aligned}$$

Allow me briefly to revisit the claim I made earlier — that we can understand the behaviour of the function $\lambda x.M$ without necessarily understanding M. Let's say you invent a new branch of mathematics whose purpose is to study some kind of interesting objects called widgets. Your theory of widgets is truly revolutionary, so much so that its nuances can only be properly expressed through your own syntax, which you invent specifically for the purpose. Your syntax is quite weird, but it follows centuries of accepted wisdom in using variables to describe widgets abstractly. Now, I don't have the first idea what a widget is and I don't know what half the symbols in your language are supposed to mean (I find the occurrences of  and  especially confusing). I can, nevertheless, describe to you the theory of functions on widgets — all I need from you is to know how to recognise a well-formed widget expression when I see one, for example, by suppling me with a grammar. Whenever M denotes a widget, I tell you (knowingly), $\lambda x.M$ denotes the function on widgets that sends every widget x to the widget M. Moreover, $(\lambda x.M)N$ denotes the action of supplying this function with a particular widget N as input. Then the theory of functions on widgets is just the set of all equations $(\lambda x.M)N =_\beta M[N/x]$.

1.2 Unit outline

The unit runs every week for the whole of teaching block 1, except weeks 8 (which is a reading week) and, unless necessary due to overspill, week 12. There are essentially 8 blocks of material.

- Weeks 1, 2 and 3: the basic definitions associated with the untyped λ-calculus.
- Week 4: the fundamental theorem of confluence of β-reduction.
- Week 5: the fundamental theorem of Turing-completeness.

- Week 6: the basic definitions associated with simply typed λ-calculi.
- Week 7: a particular example of a simply typed calculus, second-order propositional logic.
- Week 9: the basic definitions associated with the polymorphic λ-calculus, known as System F.
- Week 10: the fundamental theorem of strong normalisation.
- Week 11: the fundamental theorem of Curry-Howard correspondence.

This splits the unit roughly evenly, with half the time spent on the untyped λ-calculus and half spent on two kinds of typed λ-calculi. Let me guide you through some of the ideas that we will make precise during the next 12 weeks.

Untyped λ-calculus

The whole of the first half of this unit will concern the study of the *pure, untyped λ-calculus*. This version of the calculus arises as the formalisation of functions over functions. The language consists only of variables, applications and abstractions; there are no numbers, no strings, in fact no data of any kind except functions. Variables stand for functions, functions take functions as input and return functions as output. The whole language can be presented by the following grammar:

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

In other words, the language consists of those expressions, generically written M and N, that are either variables x (we assume an infinity of them y, z, x' and so on) or formed by the application of some expression M to another N or by the abstraction $\lambda x.M$ over a variable x in an existing expression M. In the pure, untyped λ-calculus, we allow a more general kind of application than the one we discussed above: we may apply any well-formed expression M to an input expression N. This makes sense because there are only functions in this calculus: every well-formed expression denotes a function (including the variables), so it makes sense that we should be able to use any expression M as an operator.

Allow me to emphasize the minimality of this language: every program (expression) is built out of variables, application, abstraction and nothing else — there are no secret primitives hiding around the corner. However, even without primitives, there is a rich variety of programs that one can write in this language, for example:

$\lambda x.x$	The identity function.
$\lambda y.x$	The function that is constantly x (whatever x is).
$\lambda x.(\lambda y.x)$	The (curried) function that returns its first argument.
$\lambda x.(\lambda y.x(xy))$	The function that applies its first argument to its second, twice.

In fact, although the above examples may not convince you of it, *every computable function is definable in the pure, untyped λ-calculus*. Hence, this tiny language is as powerful as any programming language ever invented and, assuming the Church-Turing thesis, that will ever be invented. This is a result that we will work towards in the first half of the course.

In the sense that it is a syntax for describing computable functions, we nowadays recognise the λ-calculus as a kind of programming language. For this purpose, the notion of making a computational "step" is more important that the notion of equality given by conversion. In the λ-calculus, an expression M makes a single step of computation by replacing a single occurrence of $(\lambda x.M)N$ by $M[N/x]$, *no matter what N is* and *no matter where inside the expression it occurs*. This notion of computation is called β-reduction.

Although it seems like quite a non-deterministic mode of computation, we will show that β-reduction is *confluent*. This means that there is a sense in which it doesn't really matter which occurrence of $(\lambda x.M)N$ that we choose at each step. This may not sound like a very exciting result but, because of the intimate connection between reduction and conversion, it will turn out to be one of the most useful tools that we have in order to reason about the latter.

Incidentally, several well-known programming languages have been designed as direct extensions of the untyped λ-calculus. Perhaps the best example is Scheme, which is a variant of LISP invented by Guy Steele and Gerald Sussman at MIT in the mid 1970s. Through a highly influential series of papers, collectively called "Lambda: The Ultimate", Steele and Sussman showed how all of the great innovations of programming language design of the 60s and 70s were actually just instances of λ-abstraction. Their project turned into a very well-regarded textbook on programming languages: "The Structure and Interpretation of Computer Programs", colloquially known as SICP or "the wizard book", that was used by most of the top universities in the USA and Europe through the 80s, 90s and early 00s.

I learned programming from this textbook in my first year as an undergraduate, and I highly recommend it. Its use, however, has gradually receded in favour of teaching undergraduates "useful" programming languages like Java. As it turns out, a generation of programmers irritated by having to program in Java have started to turn back to Scheme: it's latest incarnation Clojure is a LISP that runs on the Java Virtual Machine. OVO Energy, who are based here in Bristol, are regularly advertising for Clojure programmers.

Typed λ-calculi

Another reason for the demise of SICP was the rise of types in programming, as exemplified by the standardisation of ML (a grandparent of Haskell) that occurred during the 1990s. We shall study two kinds of *typed λ-calculi*, simply-typed λ-calculus and the polymorphic λ-calculus (also known as System F).

There are many ways that one can usefully think about types. One of the most basic ways is to think of types as a syntax for classifying program expressions according to some property they share.

In this way of thinking, the simply typed λ-calculus arises as a way to classify functions according to what kind of inputs they will accept and what kind of outputs they will produce. For example, the *simple type* $\mathsf{int} \to \mathsf{string}$ classifies those expressions that are certain to deliver an string output (if they terminate) having been given an integer input. The simple type $(\mathsf{int} \to \mathsf{int}) \to \mathsf{int}$ classifies those expressions that are certain to deliver an integer output (if they terminate) having been given a function as input which has the property that it is certain to deliver an integer output (if terminating) given an integer input.

The most important part of any typed language is the *type system*, which is the collection of rules by which types are assigned to program expressions (thereby classifying them). The simple type system is the least expressive of the systems that we shall study, and many untyped expressions cannot be assigned types by this system. However, it has a rather useful property which are not enjoyed by the others: if an untyped expression has a type assignment, then it has a best type assignment — called *principal* — which is representative of all the others.

One specific drawback of the simple type system is that there is no *polymorphism*. A particular expression is assigned a particular type and that type fixes its interface rigidly. The following expression has no type assignment in a simple type system:

$$\mathsf{let\ id} = \lambda x.x \ \mathsf{in}\ (\mathsf{id}\ \text{“hello”})\ !\ (\mathsf{id}\ 2)$$

because we are forced to choose whether id gets the type $\mathsf{string} \to \mathsf{string}$, making it suitable to be applied to “hello”, *or* whether it gets the type $\mathsf{int} \to \mathsf{int}$ making it suitable to be applied to 2. Whichever we choose makes id unsuitable to be applied to the other and hence the expression as a whole is untypable.

In the System F, the language of types is enriched so that we can express the fact that a single expression can be used at more than one simple type — polymorphism. In System F, we will see that the polymorphic identity can be assigned the type $\Pi a.\, a \to a$. In System F, we have a type system that is closer to the one used by typed functional programming languages like Haskell. In Haskell, the type $\Pi a.\, a \to a$ can be written $\mathsf{forall}\ a.\, a \to a$, but is usually just abbreviated as $a \to a$.

Polymorphism allows us to express more programs than we could with simple types, but there are still fundamental limitations to the power of this typed λ-calculus. We will show that System F is *strongly normalising*, this means that every program that we can write in this language is guaranteed to always terminate.

The study of type systems and the study of logics are intimately connected. In programming we are used to types preventing us from constructing stupid programs, but they were already being used as far back as 1902, as a means to stop logicians from constructing stupid formulas. Logic was the original motivation for typed λ-calculus and we will get a taste of it when we look at second-order

propositional logic presented as a simply typed λ-calculus. This is a presentation of logic in which the logical operators are taken to be simple truth-valued *functions*. Types are key to the presentation, we will see that the consequences of not having types for formulas is disastrous: inconsistency.

However, the connection goes further than simply a presentation of formulas. In the conclusion of course we will look at a quite remarkable correspondence between formulas and types, proofs and programs. According to the correspondence, writing programs in a typed λ-calculus is *exactly the same thing* as writing a proof in a corresponding logic. One side of the correspondence allows us to prove mathematical statements by writing programs. Indeed, large tracts of mathematics have already been formalised this way in *interactive proof assistants* such as Coq. The other side of the correspondence allows us to extract the computational content of mathematical proofs. Under the correspondence, a (constructive) proof of a statement $\forall x : \mathsf{int}.\, \exists y : \mathsf{int}.\, P(x, y)$ *is* a function that sends each integer input x to an integer output y, guaranteed to satisfy $P(x, y)$. We will study the correspondence at the level of polymorphic λ-calculus and second-order propositional logic.

1.3 Some advice

A large part of the unit consists of taking very simple, intuitive concepts, making them absolutely precise and then deducing the consequences logically. That is: we will make mathematical definitions and write mathematical proofs. I won't assume that you have any particular aptitude for writing proofs, but I will assume that you are comfortable in verifying whether a proof is valid or not (e.g. by checking, in your head, that each step in the proof corresponds to a valid inference).

The problem sheets will give you plenty of practice at writing proofs. It is ***absolutely essential*** that you block out some time each week to work on that week's problem sheet. If you do not attempt the problem sheets during the week in which they are posted (and preferably before the Monday lecture of the following week) then it is very likely that you will get lost. At all times there will be two parallel tracks in your head: one consists of ideas and the other consists of definitions. Ideas are fascinating but too wooly to actually work with. To make progress, to understand the consequences of an idea, the concept first has to made concrete with a definition and working with definitions requires *practice*.

Here's an example from programming languages to try to illustrate the importance of definitions. Suppose you have a very basic programming language, the language only includes the ability to define recursive functions and tuples over Booleans, but the language evaluates lazily. I claim that, even in such a simple, restricted programming language, it is possible to implement an infinite datatype: the type of queues of Booleans. How could you convince yourself of this claim? Where would you start?

The only way to make any kind of progress is to *make some concrete definitions* that *pin down* the concepts involved. Let me define "Boolean Haskell" as Haskell restricted to expressions that are either of type Bool or are tuples or functions of Bools. This will serve as a fine definition of the concept of programming recursive functions over Bools. Let me define a "Boolean Queue" to be any type T with the following operations: $\mathsf{enqueue} : \mathsf{Bool} \times T \to T$, $\mathsf{dequeue} : T \to T \times \mathsf{Bool}$ (dequeue is undefined if the queue is empty) and $\mathsf{empty} : T$ having their usual meanings (I should specify exactly their meanings too, but it's just an illustration). Now the problem is much clearer. To be convinced of the claim I just have to find a type T in Boolean Haskell and program three functions that behave in the right way and have the correct types.

It is still a fiendish problem[1] to find this type T and implement the functions, but at least it is a concrete one and, if you presented your implementation, we could all agree on whether or not it constituted a solution. However, notice that if you don't know how to actually program in (Boolean) Haskell, i.e. you have not had any *practice working with the definitions*, it is an order of magnitude more difficult still.

In this unit you won't be required to invent definitions yourself (and it is my responsibility to convince you that the definitions do a good job of making the concepts precise) but you will need to work with them, so it is essential to practice by doing the problem sheets!

[1] An answer can be derived from the work of Jones and Muchnik, in the article titled "The Complexity of Finite Memory Programs With Recursion" in JACM 25(2) 1978 (but they do not frame the problem using Haskell).

2. Terms and their Structure

We build up this great theory starting from humble origins: the *language* of the λ-calculus is just a certain collection of strings.

2.1 Terms

Variables. Like any reasonable programming language, there are *variables*. It doesn't matter what they are called, only that we can tell them apart. Let's agree to refer to them, generically, by variations on x, y, z and other lower case letters towards the end of the alphabet. Also, we don't want to run out, so let's assume we have a countably infinite set of them: $\mathbb{V}$.

Then our language is a certain set of strings built from an alphabet containing every variable, the period, parentheses and, of course, λ. We call the strings in this language *terms*.

Definition 2.1 (Terms)**.** *The set of* ***terms****, written* Λ*, is the subset of strings over the alphabet* $\mathbb{V} + \{\lambda, ., (,)\}$ *that is defined inductively by the rules:*

$$x \in \mathbb{V} \ \frac{}{x \in \Lambda} \ (Var) \qquad \frac{M \in \Lambda \quad N \in \Lambda}{(MN) \in \Lambda} \ (App) \qquad x \in \mathbb{V} \ \frac{M \in \Lambda}{(\lambda x.M) \in \Lambda} \ (Abs)$$

We will use M*,* N*,* U*,* V*,* W *and other variations on upper case letters in the second half of the alphabet to stand for terms, generically.*

We will see what precisely is meant by the phrase "defined inductively by the rules" in the following chapter. For now you should think of Λ as the set containing all and only those strings that can be formed according to the given rules. Each rule should be read as:

> "if the statement(s) above the line are true of any particular terms M and N and any condition to the left is satisfied, then the statement below the line is also true"

For example, $((\lambda x.x)y)$ is a term because we can justify its membership in Λ according to the given rules, as follows:

- By (Var), since x is a variable, also $x \in \Lambda$.

- It follows by (Abs), with M being the particular term x, that $(\lambda x.x)$ is also in Λ.
- By (Var), since y is a variable, also y is a term in Λ.
- It follows from (App), with M being $(\lambda x.x)$ and N being y, that $((\lambda x.x)y)$ is also a term in Λ.

Here are some other examples of terms, I leave you to justify their membership in Λ with reference to the rules:

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

Some non-examples are: $(\lambda\lambda x.y))$, since a variable always follows a lambda; $(\lambda(x).y)$, since no parentheses are allowed between lambda and the following dot; λx, since a dot and a term must follow.

2.2 Subterms

λ-terms are just strings, but our intution is that there is more to them than just a flat sequence of characters. To start with, the way they are built using the formation rules gives them a kind of hierachical structure — new terms are built by adding onto old ones. Intuitively, we can easily recognise the old terms inside the new ones, we make this precise with the notion of subterm.

Definition 2.2. *The **subterm** relation $\sqsubseteq$ is the subset of $\Lambda \times \Lambda$ defined inductively by the rules:*

$$\frac{}{M \sqsubseteq M}\ (SubRefl) \qquad \frac{P \sqsubseteq M}{P \sqsubseteq (\lambda x.M)}\ (SubAbs)$$

$$\frac{P \sqsubseteq M}{P \sqsubseteq (MN)}\ (SubAppL) \qquad \frac{P \sqsubseteq N}{P \sqsubseteq (MN)}\ (SubAppR)$$

For example, (yz) is a subterm of $((\lambda x.x)(yz))$; x is a subterm of x and x is a subterm of (xy). The former can be justified using the rules as follows:

- By (SubRefl), with M being the particular term (yz): $(yz) \sqsubseteq (yz)$.
- Therefore, by (SubAppR), with P being (yz) and M being $(\lambda x.x)$ and N being (yz): $(yz) \sqsubseteq ((\lambda x.x)(yz))$.

The immediate subterms of a term often are referred to with special names:

- The subterm M of (MN) is called the *operator*.
- The subterm N of (MN) is called the *operand*.
- The subterm M of $(\lambda x.M)$ is called the *body*.

Syntactical conventions

The parentheses are an essential part of the inductive definition of terms because they ensure that the structure of terms (e.g. the notion of subterm) is unambiguous. However, there are an awful lot of them and they are very tedious to write. So, from this point onwards we will start to omit *some* of them. It is possible to do this without introducing ambiguities as long as we agree some conventions. From now on:

- *We will omit the outermost parentheses.* For example, whenever we write MN to mean a term, the term that we mean is (MN).
- *In any subterm, we will assume that application associates to the left.* For example, whenever we write MNP, the term that we mean is $((MN)P)$.
- *In any subterm, we will assume that the body of an abstraction extends as far to the right as possible.* For example, when we write $\lambda x.MN$, the term that we mean is $(\lambda x.(MN))$.
- *In any subterm, iterated abstractions can be grouped.* For example, when we write $\lambda xy.M$, the term that we mean is $(\lambda x.(\lambda y.M))$

These conventions allow our example terms to be rendered unambiguously as:

$$xy \qquad \lambda x.yzx \qquad (\lambda x.x)(yz) \qquad z \qquad \lambda xy.x$$

2.3 The scope of a variable

Consider the term $\lambda x.yx$. We started with $\lambda x.yx$ being a completely flat string and now we have given it a tiny bit of structure through the subterm relation: we can say that yx is a fully formed part of $\lambda x.yx$ through the statement $yx \sqsubseteq \lambda x.yx$. But, our intuition is that there is more structure here than just subterms: there is a sense in which the occurrence of x before the dot and the x after the dot are linked — the first x names the input and the second x is how it gets used. So far, there is nothing explicit in our definitions that distinguishes these two occurrences of x. We are now going to say that the latter x is *bound* by the former in λx. Conversely, we say that the y, which is not bound by any lambda, is *free*.

Definition 2.3. *The set of **free variables** of a term M is defined by recursion on the structure of M:*

$$\begin{aligned} \mathsf{FV}(x) &= \{x\} \\ \mathsf{FV}(PQ) &= \mathsf{FV}(P) \cup \mathsf{FV}(Q) \\ \mathsf{FV}(\lambda x.N) &= \mathsf{FV}(N) \setminus \{x\} \end{aligned}$$

*A term M without free variables is said to be **closed** or a **combinator**. The set of all closed terms is written Λ^0.*

By contrast, a term M for which $\mathsf{FV}(M) \neq \emptyset$ is said to be *open*. The term $\lambda xy.yx$ is closed, because:

$$\begin{array}{rcl} \mathsf{FV}(\lambda xy.yx) & = & \mathsf{FV}(\lambda y.yx) \setminus \{x\} \\ & = & \mathsf{FV}(yx) \setminus \{x, y\} \\ & = & (\mathsf{FV}(x) \cup \mathsf{FV}(y)) \setminus \{x, y\} \\ & = & (\{x\} \cup \{y\}) \setminus \{x, y\} \\ & = & \emptyset \end{array}$$

We can think of an open term as being incomplete. Contrast x with $\lambda x.x$. We think of $\lambda x.x$ as being a function waiting for an input which, when supplied, will return it. Without more information we have no idea what the term x is supposed to do.

Variable binding occurs in all sorts of places in mathematics and computer science. Sometimes the binding is implicit, such as when we define a function $f\ x = x + x$ in Haskell, naming the input x. Sometimes the binding is explicit, such as in a formula $\forall x.\, x > z$ or an integral $\int x^2\ dx$. What they all have in common is to delimit the scope of x: if you see another x elsewhere in your Haskell program, you know it does *not* refer to the x inside this function f. Now, since you know the whole scope of the variable x, you can feel free to change its name to y or z as long as you change it everywhere within that scope. In other words, the particular name of a bound variable doesn't matter. we don't usually care to distinguish between the formulas $\forall x.x > z$ and $\forall y.y > z$. Nor do we see any difference between an integral as a function in x integrated with respect to x: $\int x^2\ dx$, or the same function expressed in y and integrated with respect to y: $\int y^2\ dy$.

We want to be able to say the same thing about λ-terms that differ only in the choice of a bound variable name. For example, intuitively, the following strings are all essentially describing the same term:

$$\lambda x.x \qquad \lambda y.y \qquad \lambda z.z$$

However, according to our definition, terms are just certain strings and these three are different strings (assuming x, y and z are different variables). So, we have to make a new definition, something that makes precise the idea that these three strings, although different, are essentially the same. For historical reasons[1], we say they are α-convertible.

Definition 2.4. *We say that two terms M and N are **α-convertible** just if M and N are the same up-to renaming of bound variables. More explicitly, we define the*

[1] This is a way of saying that your lecturer believes it is a bad choice.

relation of α-convertibility, written $M =_\alpha N$, inductively by the following rules:

$$\frac{}{x =_\alpha x}\ (\alpha Var)$$

$$\frac{M =_\alpha P \quad N =_\alpha Q}{MN =_\alpha PQ}\ (\alpha App) \qquad z \text{ fresh } \frac{M[z/x] =_\alpha N[z/y]}{\lambda x.M =_\alpha \lambda y.N}\ (\alpha Abs)$$

The meaning of "z fresh" is that z is a variable not in use anywhere else within our general consideration. The term $M[z/x]$ is the term M but with every occurrence of variable x renamed to z. The term $N[z/y]$ is the term N but with every occurrence of variable y renameed to z.

Intuitively, the terms $\lambda x.xy$ and $\lambda z.zy$ are α-convertible because, if I pick a fresh variable y' to use instead of x in the body of the first term and instead of z in the second term, $y'y$ and $y'y$ are trivially α-convertible (you can check this by writing out the proof tree).

Most of the time it will be obvious when two terms are α-convertible, such as the case with $\lambda x.x$ and $\lambda y.y$, or $\lambda xy.x$ and $\lambda yx.y$. However, it is absolutely crucial that we only allow renaming of *bound* variables in the definition. The term $\lambda x.xy$ is *not* α-convertible with $\lambda x.xz$ (assuming $y \neq z$). You should think of free variables as being part of the explicit interface to the term. For example, consider the closed term $\lambda yz.(\lambda x.xy)(zy)$. The body of this abstraction consists of two subterms $\lambda x.xy$ and zy, with the former applied to the latter. We can think of these two subterms interacting via the variable y, which is free in both of them (but bound in the term of which they are both a part). On the other hand, a variable that is bound in one of these subterms, like x in the first, cannot possibly interact with the other because its whole lifetime exists within the abstraction, which is completely enclosed by that subterm.

2.4 Assumption: α-convertible terms are identical

From this point on, we are going to build α-convertibility into our definition of what it is to be a term. We are going to consider the expressions $\lambda x.yx$ and $\lambda z.yz$ no longer as two terms that are *essentially the same*, but rather that they are *literally the same*.

Definition 2.5. *The set of λ-terms is the set Λ with all α-convertible terms identified.*

From now on you should think of terms not simply as strings but as an abstract data type whose notion of equality does not distinguish between terms that are α-equivalent.

The only subtle part of the unit

We have now given two separate definitions of λ-terms. For the purpose of the following discussion, let us refer to the set of strings that we defined in Definition 2.1 as the *raw terms* and the definition above as *λ-terms*.

A question: how should we refer to the unique λ-term that lies behind the raw terms $\lambda x.yx$ and $\lambda z.yz$? Experience has taught us that the best thing to do is just to refer to this λ-term by writing down whichever of the infinitely many raw terms $\lambda x_1.yx_1$, $\lambda x_2.yx_2$, ... that we find most convenient.

The situation is analogous to that for ratios and fractions. When we want to express a ratio (a rational number), we do so by writing down a fraction, which is a pair of integers. It happens that we write this pair of integers down in a stylised way, one on top of the other with a bar through the middle, but the essential content is the (ordered) pair of integers. However, a ratio is not *literally* a pair of integers. We know this because, for example, $\frac{3}{4}$ and $\frac{12}{16}$ are obviously two different pairs of integers, *yet represent one and the same ratio*. Pairs of integers are a *convenient* way to write down ratios, so long as we are mindful that $\frac{3}{4}$ and $\frac{12}{16}$ are two ways of writing the same thing. Just like in our definition of the set of λ-terms, the rational numbers can be recovered from the set of all pairs of integers by identifying[2] certain elements. The set of rational numbers is the set $\mathbb{Z} \times (\mathbb{Z} \setminus \{0\})$ with all pairs (m, n) and (p, q) identified whenever $mq - np = 0$.

From now on we will <u>write</u> raw terms, but <u>think</u> λ-terms.

This method of working has advantages and disadvantages. A disadvantage is that, because we are writing raw terms, we may write something that, although sensible for raw terms, is nonsensical for λ-terms. A silly example is that we could write $\lambda x.x \neq \lambda y.y$, which is true about these two strings, but false if we are thinking of them as λ-terms. However, you will have to trust me that this only ever becomes something we need to think about when we make new definitions regarding λ-terms (and even then, only in rare cases). Since, in this unit, you will not be asked to write any new definitions over λ-terms, you need not worry about this disadvantage.

An advantage is that, like ratios, we are free to choose the representation that is most convenient to us at any given time. With ratios, typically the most convenient representation is when the ratio is given as a fraction in lowest terms (i.e. when the numerator and denominator are coprime). For λ-terms, the most convenient representation is to choose a raw term whose bound variable names are different from any other bound variable names or any free variable names within our general consideration. For example, $(\lambda z.z)((\lambda x.x)y)$ is typically a better choice than $(\lambda y.y)((\lambda y.y)y)$, even though both strings denote the same λ-term. You will find it particularly useful when writing proofs that, if you have an abstraction term M, then you are always able to assume that it has shape $\lambda x.P$

[2] Identifying is a fancy way of saying "asserting to be equal"

with x different from any free variable name that might currently be in scope. It is so useful that it gets its own name:

> **The Variable Convention.** If $M_1, \ldots, M_k$ occur within the same scope, then in these terms you may assume that all bound variables are chosen to be different from the free variables.

Implementing λ-terms

You may ask, what really *is* a λ-term if it is not literally a string? The official answer is that, like other abstract data types, we don't really care how λ-terms are implemented. But, since you ask, there are various ways that we can achieve the desired effect of having Λ be a set containing exactly one element for each term and having α-convertible terms be the same element. One possibility is to implement the set of λ-terms as the set of equivalence classes of raw terms with respect to α-convertibility. Another possibility is to implement λ-terms as a variation on raw terms in which bound variable names are eliminated in favour of indexing the inputs from left to right.[3]

[3] This approach, called the *locally nameless representation*, would have the α-convertible raw terms $\lambda xy.yxz$ and $\lambda yx.xyz$ represented by the same string $\lambda\lambda.21z$, with 1 meaning "first input" and 2 meaning "second input".

3. Reasoning about inductive sets

In the course we will see many examples of "inductive" definitions, such as that of the set Λ of λ-terms. They are a bit strange to the uninitiated because they seem to define a new concept (e.g. the set Λ), without simply saying what it is equal to. Consequently, it is not completely straightforward to reason about inductive sets. Luckily however, inductive definitions (no matter of what set) always give us two reasoning principles that we can use and they are very helpful. One allows to show that some object is a member of the inductive set, the other allows us to show that all elements of an inductive set satisfy a property.

3.1 Membership in Λ

Membership in Λ is governed by the following principle:

Principle 3.1 (Λ-membership)**.** *A string M is a member of Λ iff there is a* proof tree *built from the rules (Var), (App) and (Abs) with conclusion $M \in \Lambda$.*

A *proof tree* for Λ is tree whose nodes are labelled by statements of the form $M \in \Lambda$ in such a way that, if a node is labelled $M \in \Lambda$ and its children are labelled $M_1 \in \Lambda \ldots M_k \in \Lambda$, then it must be that:

$$\frac{M_1 \in \Lambda \cdots M_k \in \Lambda}{M \in \Lambda}$$

is an instance of one of the given rules: (Var), (App) or (Abs). An *instance* of a rule is just a copy of the rule but with the generic M, N, x and so on replaced consistently by particular terms like $\lambda z.z$ or y.

For example, an instance of the rule (Abs) is:

$$\frac{z \in \Lambda}{\lambda z.z \in \Lambda}\ (\mathsf{Abs})$$

with $M := z$ and $x := z$ (as usual we will omit parentheses according to our conventions). A proof tree justifying the fact that $(\lambda x.x)y \in \Lambda$ is as follows:

$$\dfrac{\dfrac{\dfrac{}{x \in \Lambda}\ (\mathsf{Var})}{(\lambda x.x) \in \Lambda}\ (\mathsf{Abs}) \quad \dfrac{}{y \in \Lambda}\ (\mathsf{Var})}{((\lambda x.x)y) \in \Lambda}\ (\mathsf{App})$$

Compare this proof tree to the argument we gave for justifying the same fact in the previous chapter — the proof tree is just a very clear and compact way to present the argument. It can be useful to label all the instances of rules by the name of the rule, as we have done in the proof tree above, but typically we will be too lazy to do it. In almost all cases which rule is being used can be inferred from the terms involved.

Since the principle is an iff, we can use it to argue that some string is not a λ-term (i.e. is not a member of Λ). For example, the empty string, ϵ, is not a λ-term. It is not possible to build a proof tree for ϵ by the following reasoning. First note that any such proof tree must be rooted at the conclusion $\epsilon \in \Lambda$. The root of such a proof tree must be an instance of one of the rules (Var), (App) or (Abs). However, no matter which choice of M, N and x, the subject of every conclusion to those rules contains a non-empty string: the subject of (Var) requires that x be a variable, which is a single letter string; and both of the subjects of (App) and (Abs) contain at least one pair of parentheses.

3.2 Induction on Λ

The set of lambda terms, Λ, supports the following induction principle:

Principle 3.2 (Λ-induction)**.** *Let Φ be some property of λ-terms. If the following conditions are all met:*

(LI1) For all variables x, Φ holds of x.

(LI2) For all terms M and N, if Φ holds of M and N then Φ holds of MN.

(LI3) For all terms M and variables x, if Φ holds of M then Φ holds of $\lambda x.M$.

Then it follows that Φ holds of all λ-terms.

The conditions "Φ holds of M and N" in the antecedent of (LI2) and "Φ holds of M" in the antecedent of (LI3) are often referred to as the *induction hypotheses*. This principle gives a very useful tool for proving statements of the form "for all λ-terms M, Φ holds of M". Rather than having to prove such a statement by assuming some arbitrary term M and then showing Φ directly, we can instead split into three cases and attempt to prove (LI1), (LI2) and (LI3).

Lemma 3.1. *In the official syntax of terms, every term contains balanced parentheses (an equal number of left and right parentheses).*

Proof. We show that all terms satisfy the property of containing balanced parentheses by induction.

- To show (LI1), observe that every variable x contains exactly 0 left and 0 right parentheses.

- To show (LI2), let M and N be arbitrary terms. Let the number of left parentheses in M be l_M and the number of right parentheses be r_M. Let the number of left parentheses in N be l_N and the number of right parentheses in N be r_N. Assume the induction hypothesis, i.e. that $l_M = r_M$ and $l_N = r_N$. Then it follows that the number of left parentheses in (MN) is $1 + l_M + l_N$ and the number of right parentheses in (MN) is $r_M + r_N + 1$. Therefore (MN) also has balanced parentheses.

- Finally, to show (LI3), let x be an arbitrary variable and let M be an arbitrary term. Then assume the induction hypothesis, i.e. that M has balanced parentheses. It follows immediately that, therefore, $(\lambda x.M)$ also has balanced parentheses.

Since (LI1), (LI2) and (LI3) have been shown to be true for this property, the result follows from the principle of induction on λ-terms. □

Note that, since it is a first example, this proof has been given in as much explicit detail as possible. In practice, many shortcuts are routinely taken which you will become familiar with as the course progresses.

If you choose to read the next chapter, which is optional, you will see how to justify formally why this principle is valid (never allows us to conclude something that is false). For now, consider the following intuition. Say I have proofs of (LI1), (LI2) and (LI3) and then I claim that as a consequence Φ holds of all λ-terms. You are skeptical, so you aim to refute my claim by producing a λ-term for which Φ does not hold. Let's say you pick some particular term M and you claim that this is a term for which Φ does not hold. However, because M is made of finitely many applications of the term formation rules, no matter which M you picked I can always copy and paste together particular instances my proofs of (LI1), (LI2) and (LI3) finitely many times in order to get a (longhand) proof that Φ holds of that particular term M.

3.3 Membership in $\sqsubseteq$

Membership in $\sqsubseteq$ is governed by the following principle:

Principle 3.3 ($\sqsubseteq$-membership)**.** *A term M is a subterm of a term N iff there is a proof tree built from the rules (SubRefl), (SubAppL), (SubAppR) and (SubAbs), with conclusion $M \sqsubseteq N$.*

For example, here is a proof tree witnessing the fact that $xy \sqsubseteq \lambda x.xy$, and

two distinct proofs of $\lambda x.x \sqsubseteq (\lambda x.x)(\lambda x.x)$:

$$\dfrac{\dfrac{\dfrac{}{xy \sqsubseteq xy}\,(\mathsf{SubRefl})}{xy \sqsubseteq \lambda x.xy}\,(\mathsf{SubAbs})}{\lambda x.x \sqsubseteq (\lambda x.x)(\lambda x.x)}$$

$$\dfrac{\dfrac{\dfrac{}{\lambda x.x \sqsubseteq \lambda x.x}\,(\mathsf{SubRefl})}{\lambda x.x \sqsubseteq (\lambda x.x)(\lambda x.x)}\,(\mathsf{SubAppL}) \qquad \dfrac{\dfrac{}{\lambda x.x \sqsubseteq \lambda x.x}\,(\mathsf{SubRefl})}{\lambda x.x \sqsubseteq (\lambda x.x)(\lambda x.x)}\,(\mathsf{SubAppR})}{}$$

Generally we will not be at all interested in whether there is more than one way in which to justify a particular statement of membership, which is lucky because we will not typically label the proof trees with the rule names.

3.4 Induction on $\sqsubseteq$

The subterm relation supports the following induction principle:

Principle 3.4 ($\sqsubseteq$-induction)**.** *Let Φ be some property of pairs of terms. If the following conditions are all met:*

(SI1) For all terms M, Φ holds of (M, M).

(SI2) For all terms M, N and P, if Φ holds of (P, M) then Φ holds of (P, MN).

(SI3) For all terms M, N and P, if Φ holds of (P, N) then Φ holds of (P, MN).

(SI4) For all terms M and P, if Φ holds of (P, M) then Φ holds of $(P, \lambda x.M)$.

Then it follows that, for all pairs of terms (M,N) such that $M \sqsubseteq N$, Φ holds of (M, N).

As in the case for Λ-induction, the antecedents "Φ holds of (P, M)", "Φ holds of (P, N)" and "Φ holds of (P, M)" of conditions (SI2), (SI3) and (SI4) respectively are known as the *induction hypotheses*. As before, this principle gives us a very helpful tool for proving universal statements, but this time about the set of pairs of terms that are in the subterm relation. Instead of proving "for all terms M and N such that $M \sqsubseteq N$, Φ holds of (M, N)" directly, we can split into three cases and attempt to prove (SI1), (SI2), (SI3) and (SI4) and then appeal to the principle.

By way of an example, let's consider a commonly used alternative approach to defining the notion of subterm of a λ-term.

Definition 3.1. *We define the set of subterms $\mathsf{sub}(M)$ of a term M recursively:*

$$\begin{aligned} \mathsf{sub}(x) &= \{x\} \\ \mathsf{sub}(MN) &= \{MN\} \cup \mathsf{sub}(M) \cup \mathsf{sub}(N) \\ \mathsf{sub}(\lambda x.M) &= \{\lambda x.M\} \cup \mathsf{sub}(M) \end{aligned}$$

Obviously the definitions of the subterm relation and the set of subterms are closely related. In one direction we have the following.

Lemma 3.2. *If $M \sqsubseteq N$ then $M \in \mathsf{sub}(N)$.*

Proof. We prove that, for all terms M and N such that $M \sqsubseteq N$, it is also true that $M \in \mathsf{sub}(N)$. We use the principle of $\sqsubseteq$-induction with $\Phi(M,N)$ being the property $M \in \mathsf{sub}(N)$.

- For (SI1), let M be an arbitray term. There are three possibilities for the shape of M, either it is a variable x, an application MN or an abstraction $\lambda x.M$. By the definition of $\mathsf{sub}(M)$, no matter which is the case for M, always $M \in \mathsf{sub}(M)$.
- For (SI2), let M, N and P be arbitrary terms. Assume the induction hypothesis, i.e. $P \in \mathsf{sub}(M)$. Then, since $\mathsf{sub}(MN) = \{MN\} \cup \mathsf{sub}(M) \cup \mathsf{sub}(N)$, it follows that $P \in \mathsf{sub}(MN)$.
- For (SI3), let M, N and P be arbitrary terms. Assume the induction hypothesis, i.e. $P \in \mathsf{sub}(N)$. Then, since $\mathsf{sub}(MN) = \{MN\} \cup \mathsf{sub}(M) \cup \mathsf{sub}(N)$, it follows that $P \in \mathsf{sub}(MN)$.
- For (SI4), let M and P be arbitrary terms. Assume the induction hypothesis, i.e. $P \in \mathsf{sub}(M)$. Since, by definition, $\mathsf{sub}(\lambda x.M) = \{\lambda x.M\} \cup \mathsf{sub}(M)$, it follows that $P \in \mathsf{sub}(M)$.

□

3.5 Writing more natural proofs

In practice, using induction these induction principles so explicitly is not a very intuitive way to construct a proof. Take the case of induction on the natural numbers.

Principle 3.5. *Let Φ be a property of natural numbers. If all of the following conditions are met:*

(NI1) Φ holds of 0

(NI2) For all $n \in \mathbb{N}$, if Φ holds of n then Φ holds of $n+1$.

Then it follows that, for all natural numbers $n \in \mathbb{N}$, Φ holds of n.

If we want to prove that "for all $n \in \mathbb{N}$, $\Phi(n)$", we will usually introduce an arbitrary natural number n and say "by induction on n" then phrase the proof in terms of the two cases for the shape n: "when n is 0 ... therefore $\Phi(0)$" and "when n is of the form $m+1$, assume $\Phi(m)$ ... therefore $\Phi(m+1)$". This is just a kind of stylised use of the induction principle. If we have shown that, when an arbitrary

natural number n is 0, that $\Phi(0)$ holds then we have shown (NI1). If we have shown that, when an arbitrary natural number n is of shape $m+1$ and $\Phi(m)$, it follows that $\Phi(m+1)$, then we have shown (NI2).

I will generally write induction proofs in this more stylised way because I find it helps me to think about the subject of the proof more clearly, but you are free to write it whichever way you feel most comfortable with. For example, I would more usually write the proof of Lemma 3.2 as follows.

Proof. I will show that $P \sqsubseteq Q$ implies $P \in \mathsf{sub}(Q)$ by induction on $P \sqsubseteq Q$.

- In case $(\mathsf{SubRefl})$, $P \sqsubseteq Q$ is of shape $P \sqsubseteq P$ (i.e. $Q \equiv P$ in this case). It follows immediately from the definition of $\mathsf{sub}(P)$ that $P \in \mathsf{sub}(P)$.

- In case $(\mathsf{SubAppL})$, $P \sqsubseteq Q$ is of shape $P \sqsubseteq MN$ for some terms M and N. By definition, $\mathsf{sub}(M) \subseteq \mathsf{sub}(MN)$ and it follows from the induction hypothesis that $P \in \mathsf{sub}(M)$. Therefore $P \in \mathsf{sub}(MN)$.

- In case $(\mathsf{SubAppR})$, $P \sqsubseteq Q$ is of shape $P \sqsubseteq MN$ for some terms M and N. By definition, $\mathsf{sub}(N) \subseteq \mathsf{sub}(MN)$ and it follows from the induction hypothesis that $P \in \mathsf{sub}(N)$. Therefore $P \in \mathsf{sub}(MN)$.

- In case (SubAbs), $P \sqsubseteq Q$ is of shape $P \sqsubseteq \lambda x.M$ for some term M. By definition, $\mathsf{sub}(M) \subseteq \mathsf{sub}(\lambda x.M)$ and it follows from the induction hypothesis that $P \in \mathsf{sub}(M)$. Therefore $P \in \mathsf{sub}(\lambda x.M)$.

□

Note that I have refered to the names of the rules in the definition of the subterm relation, rather than the names of the conditions in the corresponding induction principle. That is simply because we will be introducing a lot of inductive definitions in the course and I will just assume that you can imagine the corresponding induction principle without actually writing it out. You should convince yourself that the proof given above can be justified using the principle of $\sqsubseteq$-induction, i.e. that you could, in principle, extract the original proof of Lemma 3.2 from this one.

3.6 Other inductive definitions

Many of the main definitions that we will consider in this unit are sets that are defined inductively using rules, much like Λ, $\sqsubseteq$ and $=_\alpha$. Each one will have an associated principle for deducing membership and an associated induction principle. Both can be derived mechanically from the defining rules. I hope you have a sense of how this might work based on these two examples but, when in doubt, use the following procedure.

Principle 3.6 (Induction meta-principles)**.** *Suppose we have an inductive definition of a set S using rules $R_1, \ldots, R_k$. Then membership in S is governed by the principle that $s \in S$ just if there is a proof tree built from $R_1, \ldots, R_k$ rooted at the conclusion $s \in S$. The induction principle for proving $\forall s \in S.\, \Phi(s)$, has k clauses, one for each of the rules. If rule R_i has m premises and a side condition ψ:*

$$\psi \;\; \frac{s_1 \in S \quad \cdots \quad s_m \in S}{s \in S} \;\; (R_i)$$

then the corresponding clause in the induction principle requires showing:

$$\textit{if } \Phi(s_1) \textit{ and } \cdots \textit{ and } \Phi(s_m) \textit{ then } \Phi(s)$$

Make sure that you can see how the principle of induction for Λ, $\sqsubseteq$ and $\mathbb{N}$ could be derived by this procedure.

You are well within your rights to ask how do I know that by following this procedure you end up with a principle that allows for concluding $\forall s \in S.\, \Phi(s)$? Indeed, it does not simply follow from anything we have discussed so far. To understand how these principles come about, you simply need to know precisely what is meant by "inductive definition". This is the subject of the next chapter. The material of the next chapter is not formally part of the unit, there will not be a lecture on it and it is not examinable. However, this is not because it is any more difficult than any other part of the unit, rather this is because it is not strictly to do with λ-calculus and, assuming you are happy to derive the induction principle from an inductive definition, it is not necessary for anything that follows. If, at some point during the unit, you feel curious about how inductive definitions "work" then I recommend coming back and reading it.

Inductively Defined Sets

In Chapter 2, we have already met three inductive definitions: the set Λ of all λ-terms, the subterm relation $\sqsubseteq$ and the relation of α-convertibility. We have now seen that each of these has its own induction principle, which is very helpful for proving statements of the form "for all elements ...". We will meet many more inductive definitions as the course progresses and, from now on, I will leave it up to you to derive the new induction principle from the definition of the set. Having seen these examples, and from your experience with natural number induction, I hope you can infer what the induction principle is corresponding to a given inductive definition. If you are unsure, or are interested in where these principles come from (why they are valid), then you may like to read on.

This chapter is non-examinable.
Read it only to satisfy your own curiosity.

3.7 Anatomy of an inductive definition

The idea of an inductive definition is that we want to describe an infinite set using a finite description (because we only have a little bit of room on an a4 page), by exploiting the fact that the set we have in mind has some repeated structure.

In order to keep things simple, we'll always describe inductive definitions in a uniform way. There are two components:

The universe. Whenever we define a set inductively, it will always be a subset of some other set U, sometimes rather ostentatiously called *the universe*, that we already understand. There is no restriction on what U can be, as long as it is a set.

The rules. Whenever we define a set inductively, it will always be with respect to a certain set of rules. They will always have a certain shape, which is important

to making inductive definitions work. The general shape of a rule (here specifying conditions about some set A) is as follows:

$$\phi(a_1, \ldots, a_n, a) \ \frac{a_1 \in A \quad \cdots \quad a_n \in A}{a \in A} \ (\mathsf{Name})$$

The membership statements $a_i \in A$ above the horizontal bar are traditionally called the *premises* of the rule and the statement below $a \in A$ is the *conclusion*. Any condition ϕ not involving the given set A is called a *side condition* and is set off to one side. Sometimes the rule has a name which is set off to the other side (here in parentheses).

In case you were wondering, the important point about the shape of the rules is that, when they talk about membership in the set A, they only talk about it positively (they never mention $\notin A$). All the premises and the conclusion talk about $\in A$ directly, and the side condition is forbidden from mentioning A at all.

3.8 Satisfaction of rules

We think of A as a parameter of the rule. For some values of A (some particular sets), the rule is satisfied and for others the rule is not satisfied.

Definition 3.2. *We say that a rule R is **satisfied** by a set S if, whenever the premises of R are true of S and the side condition holds, then the conclusion is true of S.*

For example, the set of all natural numbers, $\mathbb{N}$, satisfies both of the rules (in parameter A):

$$\frac{}{0 \in A} \qquad \frac{n \in A}{n+1 \in A}$$

This is because replacing the parameter A by $\mathbb{N}$ makes each of these rules true: $0 \in \mathbb{N}$ and, if n is a number such that $n \in \mathbb{N}$ then it follows that $n+1 \in \mathbb{N}$ also. Another set that satisfies both of these rules is the set of all integers $\mathbb{Z}$, because $0 \in \mathbb{Z}$ and, if $n \in \mathbb{Z}$ then, indeed, it follows that $n+1 \in \mathbb{Z}$. However, the set of even numbers does not satify the second rule, because 2 is an even number, yet $2+1$ is not.

The set Λ of all λ-terms satisfies the rules (in parameter X):

$$x \in \mathbb{V} \ \frac{}{x \in X} \qquad \frac{M \in X \quad N \in X}{(uv) \in X} \qquad x \in \mathbb{V} \ \frac{M \in X}{(\lambda x.M) \in X}$$

Let us call this collection of rules R_λ for future reference. Another set that satisfies R_λ is Σ_λ^*, the set of all finite strings over the alphabet Σ_λ. This is because: (i) every variable is a finite string over that alphabet, (ii) if M is a finite string over Σ_λ and N is a finite string over Σ_λ, then also (MN) is a finite string over Σ_λ, since '(' and ')' are letters in that alphabet, and (iii) if M is a finite string over that alphabet then also $(\lambda x.M)$ is a string over that alphabet since also $\lambda \in \Sigma_\lambda$. A set which does not satisfy all the rules in R_λ is $\{\lambda x.M \mid M \in \Lambda\}$, the set of all abstraction terms (although it does satisfy the third rule).

3.9 Meaning of the definition

All the sets that satisfy a given collection of rules (by satisfying each rule in the collection) exhibit the repeated structure that we are interested in. All that remains is to say which one of these is the set that we really mean.

Definition 3.3. *Let U be a set and let R be a collection of rules. We say that $S \subseteq U$ is **inductively defined** by the rules in R just if S is the least subset of U that satisfies all the rules in R.*

When we say *least* here, we mean smallest with respect to the subset ordering (not just in terms of cardinality). That is, a set is the *least* in some family of sets if it is a subset of every other member of the family.

Inductive definitions are another place where we have an "official" syntax but we will take shortcuts in practice. Indeed, we already have. The official definition of the set Λ of λ-terms should read as follows.

> The set $\Lambda \subseteq \Sigma_\lambda^*$ is inductively defined by the rules (in parameter X): R_λ.

However, in practice, because the particular name chosen for the parameter of the rules doesn't matter (in R_λ I happened to choose X), so we usually just replace it with the name of the set being defined. The "official" definition of the subterm relation would read as follows:

> The subterm relation $\sqsubseteq$, a subset of $\Lambda \times \Lambda$, is defined inductively by the rules (in X):
>
> $$\frac{}{(M, M) \in X}\ (\mathsf{SubRefl}) \qquad \frac{(P, M) \in X}{(P, (\lambda x.M)) \in X}\ (\mathsf{SubAbs})$$
>
> $$\frac{(P, M) \in X}{(P, (MN)) \in X}\ (\mathsf{SubAppL}) \qquad \frac{(P, N) \in X}{(P, (MN)) \in X}\ (\mathsf{SubAppR})$$

Hopefully this makes it clear why we don't insist on this strict, official syntax in practice.

3.10 The associated induction principle

It follows immediately from definition 3.3 that a set $S \subseteq U$ that is *inductively defined* by some rules R has two important properties. The fact that an inductive definition has an associated principle of induction is an easy consequence of these properties.

Theorem 3.1. *Let $S \subseteq U$ be inductively defined by some rules R. Then:*

(I1) S satisfies all the rules in R.

(I2) If T is a subset of U that satisfies all the rules in R, then $S \subseteq T$.

Proof. Follows from Definition 3.3 and the meaning of *least*. □

In fact, this property (I2), which follows immedaitely from the inductive definition, is exactly the induction principle. That might not be obvious at first glance, so allow me to explain.

An induction principle is a way of showing that some property Φ is true of all elements of an inductively defined set. We can always describe such a property Φ as a set — the subset of the universe that contains exactly those elements for which the property is true. For example, Lemma 3.2 proved that the property $M \in \mathsf{sub}(N)$ is true of all pairs of terms (M, N) in the subterm relation. This property is just the subset $\{(M, N) \mid M \in \mathsf{sub}(N)\} \subseteq \Lambda \times \Lambda$.

When we view a property Φ as a set in this way, proving that all elements of some set S satisfy the property amounts to proving that $S \subseteq \Phi$. This is because, by definition, $S \subseteq \Phi$ means exactly that $\forall s \in S. s \in \Phi$ and $s \in \Phi$ is just a way of saying that Φ holds of s. In the case that S is an inductively defined set, we have a short cut for showing that $S \subseteq \Phi$, namely property (I2). Property (I2) allows us to conclude that the inductively defined set S is a subset of another set, we just need to instantiate it with T being Φ.

For example, let's unpack (I2) in the case of the inductively defined set Λ, suggestively using the symbol Φ in place of T. For Λ, (I2) reads:

> If Φ is a subset of Σ_λ^* that satisfies all the rules (in parameter X) of R_λ, then $\Lambda \subseteq \Phi$.

Next, let's unpack our shorthand R_λ.

> If Φ is a subset of Σ_λ^* that satisfies all the rules (in parameter X):
>
> $$x \in \mathbb{V} \ \frac{}{x \in X} \qquad \frac{M \in X \quad N \in X}{(uv) \in X} \qquad x \in \mathbb{V} \ \frac{M \in X}{(\lambda x.M) \in X}$$
>
> Then $\Lambda \subseteq \Phi$.

Next, let's unpack what we mean by *satisfy* (Definition 3.2):

> If Φ is a subseteq of Σ_λ^* for which the following conditions hold:
>
> - For all variables x, $x \in \Phi$.
> - For all strings M and N, if $M \in \Phi$ and $N \in \Phi$, then $(MN) \in \Phi$.
> - For all variables x and strings M, if $M \in \Phi$ then $(\lambda x.M) \in \Phi$.
>
> then $\Lambda \subseteq \Phi$.

Finally, we just note that $M \in \Phi$ is just another way of saying "Φ holds of M" and that $\Lambda \subseteq \Phi$ is just another way of saying "Φ holds of all λ-terms", and what we end up with is Principle 3.2. So, what we introduced as an unjustified principle, Principle 3.2, is actually just a straightforward consequence of what we meant when we said that Λ is inductively defined by a certain collection of rules.

4. Substitution and β

In this chapter, we will give some life to λ-terms, by saying precisely *how* they compute. For example, we will see that $(\lambda x.x)y$ can make a computation step to become y and $(\lambda z.y)x$ can also make a computation step to become y.

4.1 Substitution

Substitution is the fundamental operation for composing terms. The idea of substitution is just to make a new term $M[N/x]$ that is the same as M but with any free occurrences of a variable x replaced by another term N. So, you may think of it as a generalisation of the idea of renaming a variable that we saw in the definition of α-equivalence and, indeed, we will use the same notation.

The idea is fine, but it leaves a question mark over certain edge cases to do with binding: does $(\lambda x.yx)[x/y]$ (we want to replace y with x) mean $\lambda x.xx$?

It turns out that it will be most useful to us for substitution to avoid disturbing the binding structure in the term M. So for these dangerous edge cases, in which the binding might be perturbed, we will simply leave substitution undefined: if a case like this would arise during substitution then you simply can't do the substitution at all.

Definition 4.1. *We define* ***capture-avoiding substitution*** *of term N for variable x in term M, written $M[N/x]$, recursively on the structure of M:*

$$\begin{array}{rcll} y[N/x] & = & y & \text{if } x \neq y \\ y[N/x] & = & N & \text{if } x = y \\ (PQ)[N/x] & = & P[N/x]Q[N/x] & \\ (\lambda y.P)[N/x] & = & \lambda y.P & \text{if } y = x \\ (\lambda y.P)[N/x] & = & \lambda y.P[N/x] & \text{if } y \neq x \text{ and } y \notin \mathsf{FV}(N) \end{array}$$

Notice the last case is only applicable under the condition that $y \notin \mathsf{FV}(N)$. This condition excludes the edge case above. The general problem with $(\lambda x.yx)[x/y]$ is that the term that we are introducing in place of y, namely x, contains a free variable that is already bound in $\lambda x.yx$. If we were to make the replacement, then this x becomes a bound variable in the new term. In such a situation, the x is said to be "captured" hence the name "capture-avoiding substitution".

There is a potential for confusion due to the interaction between the notation for substitution $M[N/x]$ and our conventions for omitting parentheses when writing terms. So let's avoid it by agreeing that $MN[P/x]$ means M applied to $N[P/x]$ and $\lambda y.M[N/x]$ means an abstraction of y with body $M[N/x]$. Here are some examples of substitutions, make sure you follow the notation!

$$\begin{aligned}(\lambda x.yx)[(\lambda z.z)/y] &= \lambda x.(yx)[\lambda z.z/y] \\ &= \lambda x.y[\lambda z.z/y]x[\lambda z.z/y] \\ &= \lambda x.(\lambda z.z)x\end{aligned}$$

$$\begin{aligned}((\lambda x.yx)x)[y/x] &= (\lambda x.yx)[y/x]x[y/x] \\ &= (\lambda x.yx)y\end{aligned}$$

$$\begin{aligned}(x(\lambda z.z)x)[\lambda z.z/x] &= x[\lambda z.z/x](\lambda z.z)[\lambda z.z/x]x[\lambda z.z/x] \\ &= (\lambda z.z)(\lambda z.z[\lambda z.z/x])(\lambda z.z) \\ &= (\lambda z.z)(\lambda z.z)(\lambda z.z)\end{aligned}$$

$$\begin{aligned}(xx)[\lambda z.z/y] &= x[\lambda z.z/y]x[\lambda z.z/y] \\ &= xx\end{aligned}$$

On the other hand $(\lambda x.yx)[\lambda z.xz/y]$ and $(y(\lambda x.x))[zx/y]$ are both undefined (even though, intuitively, the latter is not problematic).

Substitution is one of the best illustrations of us *writing* terms as certain strings, but *thinking* of λ-terms. To define substitution I have written down a certain operation on strings, which prescribes how certain strings get replaced by certain other strings. It's my duty, since I made the definition, to ensure that it makes sense when we think of the strings as λ-terms and the conditions $y \neq x$ and $y \notin \mathsf{FV}(N)$ on the last clause are what I had to do to ensure that.

You reap the benefit: since I promised you that the definition (like all definitions that we will use in this unit) makes sense for λ-terms, you get to use it with whichever choice of bound variables makes you happiest. In this case, this has the consequence that you will *never* be prevented from making a substitution, despite the condition on the last clause. If you ever want to perform a substitution $(\lambda y.P)[N/x]$ where either $y = x$ or $y \in \mathsf{FV}(N)$, then you can just choose a different representation of $(\lambda y.P)$ which is α-convertible with the original. I recommend choosing a bound variable z which doesn't occur anywhere within your sight. This way, the conditions will be satisfied and you can continue.

4.2 One-step β-reduction

The standard notion of computation associated with λ-terms is called β-reduction. If $\lambda x.M$ is meant to represent a function with formal parameter x and $(\lambda x.M)N$ is meant to represent the act of applying that function to an actual parameter N, then the output should be $M[N/x]$.

Definition 4.2. *A term of the form $(\lambda x.M)N$ is called a β-**redex** and we say that $M[N/x]$ is the **contraction** of the redex.*

Beta reduction belongs to a larger family of computation called *rewriting*, in which a state of a computation is just some syntactic expression and the computation proceeds by manipulating or rewriting this expression into a new one. The name "redex" hails from this lineage; it is a portmanteau of "reducible expression".

In β-reduction of a λ term, computation proceeds by contracting redexes. A term M makes step of β-reduction resulting in a term N just if N is the result of contracting a single β-redex anywhere inside M. In such a case, we write $M \to_\beta N$. We will give the formal definition of $\to_\beta$ in a moment, but the idea is that all of the following are instances of this relation:

$$\begin{aligned}
(\lambda x.x)(\lambda y.y) &\to_\beta \lambda y.y \\
(\lambda z.(\lambda x.yx)z) &\to_\beta \lambda z.yz \\
(\lambda x.(\lambda z.x)yy)(\lambda x.xy) &\to_\beta (\lambda x.xy)(\lambda x.xy) \\
(\lambda x.(\lambda z.x)yy)(\lambda x.xy) &\to_\beta (\lambda z.(\lambda x.xy))yy
\end{aligned}$$

On the other hand, the following are not an instances of this relation. In the first case, no redexes are contracted and in the second two redexes are contracted one after the other.

$$\begin{aligned}
\lambda x.(\lambda z.x)y((\lambda z.z)x) &\not\to_\beta \lambda x.(\lambda z.x)y((\lambda z.z)x) \\
\lambda x.(\lambda z.x)y((\lambda z.z)x) &\not\to_\beta \lambda x.xx
\end{aligned}$$

It is important to make sure you are clear on the bracketing within complicated terms! There are *no* redexes in the term $\lambda x.x(\lambda z.z)y$. If we put the implicit parentheses in that we have omitted by convention, this term would read: $(\lambda x.((x(\lambda z.z))y)$. This term is a function that takes an input a function x and first applies that function to $(\lambda z.z)$ and then second applies the result of that to y. There is no subterm of shape $(\lambda x.M)N$ – a redex. On the other hand, there is a redex in the term $\lambda x.x((\lambda z.z)y)$, namely $(\lambda z.z)y$.

In order to define β-reduction formally, we need to formalise the (admittedly easily understood) phrase "contracting a single β-redex anywhere inside M".

Definition 4.3. *The **one-step β-reduction** relation, written infix as $\to_\beta$, is inducti-*

vely defined by the following rules:

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

$$\frac{M \to_\beta M'}{MN \to_\beta M'N}\ (AppL) \qquad \frac{N \to_\beta N'}{MN \to_\beta MN'}\ (AppR) \qquad \frac{M \to_\beta N}{\lambda x.M \to_\beta \lambda x.N}\ (Abs)$$

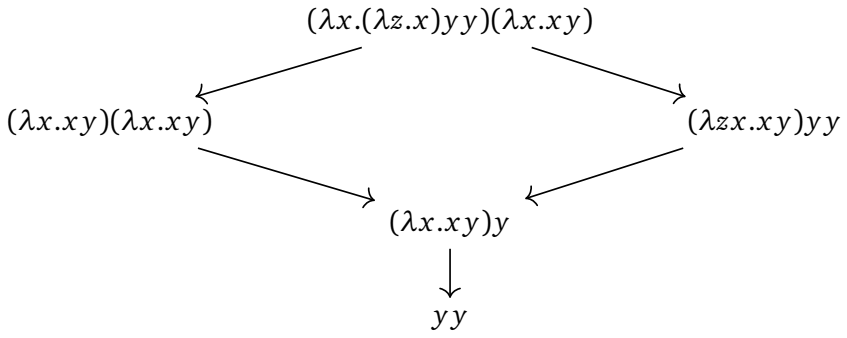
If $M \to_\beta N$ then we say that N is a ***reduct*** *of M.*

The three inference rules with premises are known as the rules of compatible closure. It is these rules that help make precise the phrase "anywhere in M" because they allow us to say that some relationship between a subterm of M and a subterm of N can be lifted all the way up to become a relationship between M and N.

Since we know how to reason about membership in inductive sets, we are now in a position to justify the one-step reductions listed at the start of this subsection. You may wish to do this for practice, but I will not ask you to do it for marks because I think the idea of contracting a redex "anywhere in M" is straightforward enough that we can all recognise it when we see it. What is more useful is the fact that we can use the principle of induction in order to justify properties shared by all possible one-step reductions.

4.3 Reduction Graph

There is a sense in which one-step β-reduction is non-deterministic: as we saw in the earlier examples, for certain terms there is more than one possible reduct. This can be pictured very nicely if we draw the reduction graph of a term. The reduction graph of a term P is just a rooted directed graph. The vertices of the graph are P and all of the terms that can be reached by making one-step reductions from P, one-step reductions from those terms and so on. There is an edge between two of these vertices M and N just if $M \to_\beta N$. For example, the reduction graph of $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$:



4.4 β-Normal Form

If we think of making a β-reduction step as performing one step of computation, then we can think of a term that cannot make a step as having terminated. In the rewriting parlance, we say that it is in *normal form*.

Definition 4.4 (β-normal form)**.** *A term M is in β-normal form just if there is no term N for which $M \to_\beta N$.*

It is a very simple notion to be worthy of its own definition, but it is an important one. Certainly, being halted or terminated or, as we say in this unit, being in *normal form* is a special kind of state to be in. In the λ-calculus especially, where there is no explicit separation between functions and the inputs and outputs that they operate on, the normal forms can be taken to be something analogous to data — we shall see an example of this next lecture.

Terms in normal form are typically easier to deal with than arbitrary terms. For example, their reduction graph is quite quick to draw. Moreover, they have a limited number of fixed shapes and if we take the set of terms in β-normal form as a subset of all λ-terms, it has an inductive structure. This is because to make a β-reduction step is to contract a redex anywhere, so to be in β-normal form is to say that there are no redexes anywhere to contract.

Lemma 4.1. *M is in β-normal form iff M has no β-redexes.*

Proof. This is quite clear if you are happy with the idea that making a β-reduction step means contracting a redex *anywhere* in the term. Otherwise it will require an induction in both directions. □

5. Reduction

Reduction gives rise to a notion of equality between terms by saying (roughly) that two terms are equal if you can perform some computation steps on them until they become syntactically identical. For example, $(\lambda x.x)y$ and $(\lambda z.y)x$ are equal in the sense that they both reduce to the same term y.

5.1 Many-step β-reduction

In general, when we say that a term M "β-reduces" to a term N, we mean that it does so in a sequence of zero or more steps. We will write this as $M \twoheadrightarrow_\beta N$. For example, we will say that $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$ β-reduces to yy, because:

$$(\lambda x.(\lambda z.x)yy)(\lambda x.xy) \to_\beta (\lambda x.xy)(\lambda x.xy) \to_\beta (\lambda x.xy)y \to_\beta yy$$

Perhaps the most obvious way to make this precise is to introduce the concept of reduction sequences, e.g. to say that $M \twoheadrightarrow_\beta N$ just if there is a finite sequence of terms $M_1, \ldots, M_k$ (for some $k \geq 1$) and $M_1 = M$ and $M_k = N$ and, for all $i \in [1..k-1]$, $M_i \to_\beta M_{i+1}$. This would work fine, and it is an excellent way to think about β-reduction in general. However, it turns out that there is an equivalent definition that is often simpler to work with.

Definition 5.1. *The β-**reduction** relation, written infix as $\twoheadrightarrow_\beta$ is defined inductively by the following rules:*

$$\frac{M \to_\beta N}{M \twoheadrightarrow_\beta N}\ (Step)$$

$$\frac{}{M \twoheadrightarrow_\beta M}\ (Refl) \qquad \frac{M \twoheadrightarrow_\beta P \quad P \twoheadrightarrow_\beta N}{M \twoheadrightarrow_\beta N}\ (Trans)$$

The addition of rules of shape (Refl) and (Trans) to an inductive definition of a binary relation is sometimes called taking the reflexive transitive closure of that relation. The rule (Refl) allows us to reduce by making zero steps, (Trans) allows us to chain together sequences of steps. Of course, it would also make sense to disallow zero step reductions by making another inductive definition but this time omitting the reflexivity rule. This is occasionally useful and the relation so defined is usually written $\to_\beta^+$.

Now that we know how to compute with λ-terms, let's look at some interesting programs and their names. On the left the combinators and how we shall abbreviate them. On the right an illustrative example of how they behave.

$$\begin{array}{rcll} \mathbf{I} & \coloneqq & \lambda x.x & \mathbf{I}M \twoheadrightarrow_\beta M \\ \mathbf{K} & \coloneqq & \lambda xy.x & \mathbf{K}MN \twoheadrightarrow_\beta M \\ \mathbf{S} & \coloneqq & \lambda xyz.xz(yz) & \mathbf{S}MNP \twoheadrightarrow_\beta MP(NP) \\ \omega & \coloneqq & \lambda x.xx & \omega M \twoheadrightarrow_\beta MM \\ \mathbf{\Omega} & \coloneqq & \omega\omega & \mathbf{\Omega} \twoheadrightarrow_\beta \mathbf{\Omega} \\ \mathbf{\Theta} & \coloneqq & (\lambda xy.y(xxy))(\lambda xy.y(xxy)) & \mathbf{\Theta}M \twoheadrightarrow_\beta M(\mathbf{\Theta}M) \end{array}$$

The term $\mathbf{I}$ is sometimes called the *identity* combinator, for obvious reasons. The term $\mathbf{\Theta}$ is sometimes called *Turing's combinator* after it's inventor, Alan Turing.

5.2 Conversion

For computational purposes, reduction is the primary notion. For the purposes of logic and reasoning, it is more useful to work with equations.

Definition 5.2. *The relation of β-**conversion**, written infix as $=_\beta$, is defined inductively using the following rules:*

$$\frac{M \to_\beta N}{M =_\beta N}\ (\mathit{Step}) \qquad \frac{}{M =_\beta M}\ (\mathit{Refl}) \qquad \frac{M =_\beta P \quad P =_\beta N}{M =_\beta N}\ (\mathit{Trans}) \qquad \frac{N =_\beta M}{M =_\beta N}\ (\mathit{Symm})$$

*We say that two terms M and N are β-**convertible** (or sometimes just β-equal) just if $M =_\beta N$.*

Since $=_\beta$ is defined by adding rules expressing reflexivity, transitivity and symmetry to a rule incorporating the relation $\to_\beta$, we can refer to it as the reflexive, transitive, symmetric closure of $\to_\beta$.

Most of the time we will not justify an equation $M =_\beta N$ using the rules *explicitly*. Since the definition contains all of the rules of the definition of $\twoheadrightarrow_\beta$, just with the symbol $\twoheadrightarrow_\beta$ swapped for $=_\beta$, any proof tree witnessing $M \twoheadrightarrow_\beta N$ can be easily turned into a proof tree for $M =_\beta N$. So if we know that $M \twoheadrightarrow_\beta N$, we already know that $M =_\beta N$ too.

Another common situation is as follows, suppose we know that $M \twoheadrightarrow_\beta N$ and also that $P \twoheadrightarrow_\beta N$. In such a case, we say that N is a *common reduct* of M and P because they both reduce to this same term N, i.e. $M \twoheadrightarrow_\beta N \twoheadleftarrow_\beta P$. Whenever two terms have a common reduct we can conclude that $M =_\beta P$. The reason is

that, as we stated above, $M \twoheadrightarrow_\beta N$ implies $M =_\beta N$, and $P \twoheadrightarrow_\beta N$ implies $P =_\beta N$. By symmetry, also $N =_\beta P$. From this and $M =_\beta N$ we can conclude $M =_\beta P$ using the transitivity rule.

It is possible to generalise this argument to whole sequences of β-reductions of alternating directions.

Lemma 5.1. *Let M and N be terms. If there is a sequence of terms $M_1, \ldots, M_k$ such that all of the following hold:*

(i) $M_1 = M$ *and* $M_k = N$

(ii) for all $i \in [1..k-1]$: $M_i \twoheadrightarrow_\beta M_{i+1}$ *or* $M_{i+1} \twoheadrightarrow_\beta M_i$.

Then it follows that $M =_\beta N$

We will not give a proof here, because the result follows from a more abstract theorem that we will prove next week. Here are some examples of conversions, justified using the above lemma:

$$\begin{array}{rll} \mathbf{K\Omega I} =_\beta \mathbf{\Omega} & \text{because} & \mathbf{K\Omega I} \twoheadrightarrow_\beta \mathbf{\Omega} \\ \mathbf{I\Omega} =_\beta \mathbf{K\Omega I} & \text{because} & \mathbf{I\Omega} \twoheadrightarrow_\beta \mathbf{\Omega} \twoheadleftarrow_\beta \mathbf{K\Omega I} \\ \mathbf{K\Omega I} =_\beta \mathbf{KK\Omega} & \text{because} & \mathbf{K\Omega I} \twoheadrightarrow_\beta \mathbf{\Omega} \twoheadleftarrow_\beta \mathbf{I\Omega} \twoheadleftarrow_\beta \mathbf{SKK\Omega} \end{array}$$

We will have a very useful alternative characterisation of when $M =_\beta N$ by the end of next week.

β-conversion allows us to describe the main characteristic of another well known combinator, Haskell Curry's *paradoxical combinator*, more often just called the Y combinator.

$$\mathbf{Y} \;:=\; \lambda f.(\lambda x.f(xx))(\lambda x.f(xx)) \qquad\qquad \mathbf{Y}M =_\beta M(\mathbf{Y}M)$$

This equation regarding the Y combinator is an excellent example of an instance where $M =_\beta N$ but $M \not\twoheadrightarrow_\beta N$.

Conversion allows us to transfer some of the concepts we are used to being associated with functions in mathematics, which are typically defined in terms of equality, to the functions in the λ-calculus.

5.3 Church's Numeral System

I described the combinators as being "interesting programs", but I will forgive you if you are thinking that you and I have very different conceptions of what is interesting. Probably you have in mind that interesting programs should at least compute with data. In computers as we know them, all data is stored in binary — that is, data is ultimately represented as a number, which happens to be presented using sequences of zeros and ones.

Binary is a kind of *numeral system*. A numeral is just a way of representing a number. In the binary numeral system, the numerals are sequences of zeros

and ones. In the decimal numeral system, the numerals are sequences of decimal digits. In the Roman numeral system, the numerals are sequences of certain letters 'I','V','X','C','D','M'. In ancient Mesopotamia, so Wikipedia tells me, clay tokens were used as numerals. In Church's numeral system, the numerals are certain λ-terms.

Definition 5.3. *The **Church numeral** for the number n, abbreviated $\ulcorner n \urcorner$, is:*

$$\lambda f\, x.\, \underbrace{f(\cdots(f}_{\text{n-times}}\, x)\cdots)$$

In other words, the Church numeral for n is the λ-term that takes a function f and an argument x and iterates f n-times on x. Let's list the first natural numbers, using Church numerals:

$$\begin{array}{rcl}
\ulcorner 0 \urcorner & := & \lambda f\, x.x \\
\ulcorner 1 \urcorner & := & \lambda f\, x.f\, x \\
\ulcorner 2 \urcorner & := & \lambda f\, x.f\,(f\, x) \\
\ulcorner 3 \urcorner & := & \lambda f\, x.f\,(f\,(f\, x)) \\
\ulcorner 4 \urcorner & := & \lambda f\, x.f\,(f\,(f\,(f\, x))) \\
 & \vdots &
\end{array}$$

You might ask whether this is a good way to represent numbers in the λ-calculus or even how we know it is a representation at all. Surprisingly there is not a standard answer to this question. It certainly has some merits. For example, it seems like a good idea that all the numerals are normal forms, because you don't typically expect your data to spontaneously start computing. I think you'll agree that is a point in its favour that every number can be represented this way and that we haven't used the same numeral for two different numbers. It might seem strange that each number is represented by a function, so you can form strange looking terms like $\ulcorner 1 \urcorner\, M\, N$, but since everything in the λ-calculus is a function, we were never going to be able to avoid that. Then, since numerals are unavoidably going to be a kind of function, there is something aesthetically pleasing about having the numeral for n be an n-fold iterator.

In reality, there are many different ways we could have used λ-terms to represent numbers. The central question is, does the system allow us to define the operations that we naturally associate with the number system?

I think it should be quite clear how we, as humans, could compute with this numeral system. For example, if I want to add two numbers $\ulcorner m \urcorner$ and $\ulcorner n \urcorner$, then I just look at how many occurrences of the symbol f (more precisely: the first bound variable) there are in each numeral, which will be $m + n$ in this case, and the answer is the numeral that has that number of applications. Similarly, if I want to compute subtraction, multiplication or exponentiation. What is especially useful about Church's numerals (although not unique to them), is that all of these standard numeric functions are actually computable *within the λ-calculus*.

The λ-term $\lambda yz.\lambda fx.yf(zfx)$ actually computes the addition of two church numerals given as input. More precisely, this term has the property:

$$(\lambda yz.\lambda fx.yf(zfx))\ulcorner m\urcorner\ulcorner n\urcorner \twoheadrightarrow_\beta \ulcorner m+n\urcorner$$

which holds for all natural numbers m and n. Let's add it to our list of useful combinators.

$$\mathbf{Add} \coloneqq \lambda yz.\lambda fx.yf(zfx) \qquad \mathbf{Add}\ulcorner m\urcorner\ulcorner n\urcorner \twoheadrightarrow_\beta \ulcorner m+n\urcorner$$

The idea is as follows. The term **Add** is meant to take two church numerals y and z as input and deliver another as output, so this explains its general shape $\lambda yz.\lambda fx.M$, where I deliberately separated the first and second from the third and fourth arguments for emphasis. Let's say we apply this term to two numerals $\ulcorner m\urcorner$ and $\ulcorner n\urcorner$. Then the result, here $M[\ulcorner m\urcorner/y][\ulcorner n\urcorner/z]$, must reduce to an $m+n$-fold application of f to x. Notice that this is the same thing as an m-fold application of f to (an n-fold application of f to x). An n-fold application of f to x can be arranged by the term $\ulcorner n\urcorner fx$, and an m-fold application of f to this can be arranged as the term $\ulcorner m\urcorner f(\ulcorner n\urcorner fx)$. Hence, this justifies the term making up the body: $yf(zfx)$.

Once you have gone away and thought about it a bit, I hope you will agree with me that addition is not too tricky. It really exploits the fact that the Church numeral for n *is* an n-fold iterator. Implementing λ-terms that compute multiplication and exponention are also quite reasonable, since multiplication can be thought of as iterated addition and exponentiation as iterated multiplication.

Implementing subtraction in terms of iterators, on the other hand, is much less obvious. In fact, even implementing subtraction by 1, the predecessor function, is already difficult. However, it can be done: $\lambda z.\lambda fx.z(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$ is a fine implementation of predecessor.

$$\mathbf{Pred} \coloneqq \lambda z.\lambda fx.z(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u)$$

$$\mathbf{Pred}\ulcorner 0\urcorner \twoheadrightarrow_\beta \ulcorner 0\urcorner$$
$$\mathbf{Pred}\ulcorner n+1\urcorner \twoheadrightarrow_\beta \ulcorner n\urcorner$$

It is a typical convention when in the natural numbers that the predecessor of zero is zero. It's not, however, very easy to understand why it works. In this weeks problems you will derive another implementation of predecessor which, although a larger term, has a clearer idea behind its inception. However, the bigger message is simply one of existence: there *exists* a term **Pred** with the property that $\mathbf{Pred}\ulcorner 0\urcorner \twoheadrightarrow_\beta \ulcorner 0\urcorner$ and $\mathbf{Pred}\ulcorner n+1\urcorner \twoheadrightarrow_\beta \ulcorner n\urcorner$. The fact that the definition of this term looks a bit of a mess is neither here nor there — whenever you want to use the term as part of a definition of another term or reason about it you will always just think in terms of its two properties.

There is another combinator that I want to introduce to you, which, although it may not occur to you as an operation that you typically associate with natural numbers, is nevertheless extremely useful. The *test-for-zero* combinator is the

term $(\lambda x\,y\,z.x(\mathbf{K}\,y)z)$. You should think of it as a conditional if $x = 0$ then y else z because it is easily verified that it satisfies the following properties.

$$\textbf{ifzero} \coloneqq (\lambda x\,y\,z.x(\mathbf{K}\,y)z) \qquad \begin{array}{l}\textbf{ifzero}\ulcorner 0\urcorner\ulcorner p\urcorner\ulcorner q\urcorner \twoheadrightarrow_\beta \ulcorner p\urcorner \\ \textbf{ifzero}\ulcorner n+1\urcorner\ulcorner p\urcorner\ulcorner q\urcorner \twoheadrightarrow_\beta \ulcorner q\urcorner\end{array}$$

This combinator gives us a way of building conditional branching into our number-computing programs.

6. Recursion

We usually use the term recursive function for those whose definition is given in terms of itself. The classic example is the factorial function on natural numbers:

$$\begin{array}{rcl} \mathsf{fact} & : & \mathbb{N} \to \mathbb{N} \\ \mathsf{fact}(n) & = & \text{if } n = 0 \text{ then } 1 \text{ else } n \cdot \mathsf{fact}(n-1) \end{array}$$

The presence of the function's name, fact, on both sides of the $=$ is the tell-tale sign that this is a recursive function. So, it seems that names are important for giving recursive function definitions. In λ-calculus there is no formal support for attaching names to functions — no syntax for naming that is built into the notion of reduction or conversion. So a question arises: can we even define recursive functions in the λ-calculus and, if not, how can we perform repetitive computations?

6.1 Recursive definition

To answer this question we have to understand more fully what we mean when we say that the above equation is a definition the factorial function. As computer scientists, and especially as functional programmers, we are a bit spoilt by the very liberal notion of function definition given in languages like Haskell. In Haskell every equation of the form $f\,x = e$ (for some Haskell expression e), assuming it type-checks, is a valid function definition. Consequently, we may be led to believe that, in general, if we write a name f followed by a formal parameter x and then $=$ followed by some expression e, we have defined a function.

However, this is not the case. Consider the following "definition" of a function yuck over the integers:

$$\begin{array}{rcl} \mathsf{yuck} & : & \mathbb{Z} \to \mathbb{Z} \\ \mathsf{yuck}(n) & = & \mathsf{yuck}(n) + 1 \end{array}$$

Is this a valid function definition? The answer is no: we can prove it. Assume, for the purposes of obtaining a contradiction, that there is a function yuck on integers with the property that the following integer equation is true: $\mathsf{yuck}(n) = \mathsf{yuck}(n) + 1$. Then we can subtract $\mathsf{yuck}(n)$ from both sides, to obtain the true equation $0 = 1$, which is absurd.

To understand this, we have to be clear on what we mean by the word function. Intuitively, we think of a function f as some black box which associates to each input x an output $f(x)$. This intuition is made precise through the *definition* of function in Zermelo Fraenkel (ZF) set theory, and this is usually the meaning of the word function in mathematics more generally. In ZF, a function f from a set A to a set B is a subset of $A \times B$ in which each element of A is associated with exactly one element of B.

According to this definition, a function is just a tabulation of its inputs and outputs, one can say that the notion of function and the notion of *graph of a function* are identified. So, strictly speaking, the factorial function is a certain set of pairs, starting off like:

$$\begin{array}{ccc} 0 & \mapsto & 1 \\ 1 & \mapsto & 1 \\ 2 & \mapsto & 2 \\ 3 & \mapsto & 6 \\ & \vdots & \end{array}$$

Since there is exactly one output for each input, the sentence "the output of fact corresponding to input 3" unambiguously refers to exactly one natural number, 6. However, it is a bit long-winded to say, so in 1734 Leonard Euler invented a more concise notation: $\mathsf{fact}(3)$.

Unfortunately, the department's budget does not stretch far enough to allow me to purchase enough paper to give a complete listing of the factorial function. So we have to look for some finite means by which we can *specify* this infinite set, unambiguously. This is purpose of the equation. The equation constitutes a *definition* of factorial in the following sense.

Definition 6.1. *We define* fact *as the unique function on natural numbers (i.e. subset of* $\mathbb{N} \times \mathbb{N}$ *with the property that exactly one output number is associated with each input number) that satisfies the equation (in parameter F):*

$$F(n) = \mathsf{if}\ n = 0\ \mathsf{then}\ 1\ \mathsf{else}\ n \cdot F(n-1)$$

So, in a sense, Definition 6.1 is the "official" definition of the factorial function, but, in practice we omit the parameter F and instead write the equation directly using fact, as we did at the start of this chapter. It should remind you of the way that we make inductive definitions of sets: remember that, officially, a set S inductively defined by some rules (given in terms of some parameter X) is the least set that satisfies the rules; but in practice omit X and just write the rules in terms of S.

Of course, if we want to make such a definition then there is some obligation on us to be sure that the equation has a unique solution. However, we know a few syntactic tricks that make it often straightforward to tell. For example, if we do write the equation in the form $f(x) = e$ *and* there is a well-founded order on

the domain *and* we only make recursive calls in e on strictly smaller values, then we can be sure that this equation specifies a genuine function.

We can now see what is the key mistake in our "definition" of yuck: there are *no* functions over integers that satisfy the equation (in parameter F):

$$F(n) = F(n) + 1$$

No wonder that by assuming that one exists, named yuck, we are led inevitably to a contradiction!

Before we leave this excursion into the world of recursive definitions outside of the λ-calculus, let me return briefly to Haskell. The equation:

$$\begin{array}{rcl} \mathsf{yuck} & :: & \mathsf{Int} \to \mathsf{Int} \\ \mathsf{yuck}\ n & = & \mathsf{yuck}\ n + 1 \end{array}$$

is certainly a meaningful definition of a Haskell function. Making this statement precise is the purpose of the subject called *semantics of programming languages*. Denotational semantics allows us to view our programs as defining genuine mathematical functions. In this case, the denotational view corresponds to the following theorem: *there is exactly one continuous*[1] *function F from the set $\mathbb{Z} \cup \{\bot\}$ to the set $\mathbb{Z} \cup \{\bot\}$ that satisfies the equation $F(n) = F(n) + 1$.* In fact, this function is:

$$\begin{array}{rcl} \bot & \mapsto & \bot \\ 0 & \mapsto & \bot \\ -1 & \mapsto & \bot \\ 1 & \mapsto & \bot \\ -2 & \mapsto & \bot \\ 2 & \mapsto & \bot \\ & \vdots & \end{array}$$

So, we can think of the Haskell equations as definitions of genuine mathematical functions, as long as we remember that they are specifying *continuous* functions over sets with bottoms.

6.2 Recursive functions in λ-calculus

Conversion allows us to specify recursive equations in the λ-calculus. It makes sense, for example, to ask: *is there a λ-term M satisfying:*

$$M\,x =_\beta x(M\,x)$$

I hope you agree that, if there is such a term M then, we can think of it as a recursive function. In fact, there is a term satisfying this equation, the term

[1] Without going into the definition, we can think of *continuity* as a certain property of functions which is something like computability.

$(\lambda yz.z(yyz))(\lambda yz.z(yyz))$. It's easy to check! For brevity, let me write the term $(\lambda yz.z(yyz))$ as N. Then:

$$NNx \to_\beta (\lambda z.z(NNz))x \to_\beta x(NNx) \qquad \text{therefore, also:} \qquad NNx =_\beta x(NNx)$$

so, indeed, NN is an example of a term M that satisfies the equation — remember that NNx is our abbreviation for $((NN)x)$. So, NN, or more fully

$$(\lambda yz.z(yyz))(\lambda yz.z(yyz))$$

is a recursive function, and it is defined just using variables, abstraction and application.

Not all recursive equations have solutions (terms that satisfy them) in the λ-calculus. For example, the following equation has no solution for M:

$$(\lambda x.\lambda y.y)M =_\beta (\lambda x.\lambda yz.y)M$$

How can you argue it? Something to do with normal forms. For now you'll just have to trust me, but by the end of the next chapter you will be able to prove it for yourself. I expect you think it's cheating to ask about this recursive equation because it hardly looks like the definition of a recursive function. Well, it is still a meaningful question to ask whether this recursive equation has a solution, but I agree that it is not a natural way to ask about recursive function definitions.

It might be more reasonable to restrict our attention to equations of the form:

$$M\,x_1, \ldots, x_n =_\beta N$$

for some fixed term N (which may, of course, contain occurrences of M). The equation that we started with, $Mx =_\beta x(Mx)$ has this shape, with N being $x(Mx)$. We will see that, in the pure, untyped λ-calculus, all such recursive equations have a solution for M.

6.3 Fixpoints

The fact that we can always find solutions to such equations follows from one important property enjoyed by all our λ-terms. Although very simple, some people would say that it is *the* fundamental property of untyped λ-calculus.

We start by transferring the concept of fixed point (or fixpoint) of a function to λ-terms. On mathematical (set-theoretic) functions, the concept is defined as follows. Let A be a set and f be a function from A to A. Then $a \in A$ is said to be a *fixed point* of f just if $f(a) = a$. For example, 0 is a fixed point of the "doubling" function $z \mapsto 2 \cdot z$ on natural numbers, because $2 \cdot 0 = 0$. Some functions on naturals do not posess any fixed points, for example, there is no natural z that is equal to its own successor $z+1$, so the successor function has no fixed points. On the other hand, every natural is a fixed point of the identity function on naturals. Let's transfer this idea to λ-calculus.

Definition 6.2. *A λ-term N is said to be a **fixed point** of another λ-term M just if $MN =_\beta N$.*

According to this definition there are analogues to the examples that we discussed above, but it's not a perfect correspondence. For example, every natural number is a fixed point of the identity:

$$\mathbf{I}\ulcorner n\urcorner =_\beta \ulcorner n\urcorner$$

It's also the case that no natural is the fixed point of the successor function:

$$(\lambda x.\mathbf{Add}\, x\, 1)\ulcorner n\urcorner \neq_\beta \ulcorner n\urcorner$$

although it's actually a bit difficult for us to prove this until next chapter. However, this implementation of the successor function *does* posess a fixed point! I can tell you that without even looking at its definition, because of the following theorem.

Theorem 6.1 (First Recursion Theorem)**.** *Every λ-term posesses a fixed point.*

Proof. Let M be an arbitrary λ-term. We claim that $\mathbf{Y}M$ is a fixed point of M. We calculate:

$$\mathbf{Y}M \twoheadrightarrow_\beta (\lambda x.M(xx))(\lambda x.M(xx)) \twoheadrightarrow_\beta M((\lambda x.M(xx))(\lambda x.M(xx))) \twoheadleftarrow_\beta M(\mathbf{Y}M)$$

Therefore $M(\mathbf{Y}M) =_\beta \mathbf{Y}M$, making $\mathbf{Y}M$ a fixed point of M. □

The proof of the First Recursion Theorem tells us more, not only does every λ-term possess a fixed point, but the fixed point can be computed within the λ-calculus. The term $\mathbf{Y}$ is an example of a *fixed point combinator*, so called because it computes the fixed point of another λ-term.

6.4 Fixpoints and recursive functions

We can use fixed point combinators to solve certain kinds of recursive equations, including those of shape:

$$Mx_1 \cdots x_n =_\beta N$$

where, of course, N may contain some occurrences of M and x_1,...,x_n. There is a certain recipe that we can follow in order to find M.

First, we know from the rules of $=_\beta$, it would be sufficient to find an M that satisfies the following equation:

$$M =_\beta \lambda x_1, \ldots, x_n.N$$

This is because we can go from the bottom equation to the top one: we just apply $x_1, \ldots, x_n$ in order to both sides which, you will show in this weeks problems, preserves $=_\beta$.

Next, we can "abstract out M", we just look at N and replace all the occurrences of M in there by the same fresh variable. Let's say that variable f does not occur anywhere else, and N' is the result of replacing all the occurrences of M in N by f. Then it is sufficient to find an M that satisfies:

$$M =_\beta (\lambda f.\lambda x_1, \ldots, x_n.N')M$$

We know about this shape. The First Recursion Theorem guarantees that there is a solution and fixed point combinators allow us to describe it directly. For example, $\mathbf{Y}(\lambda f.\lambda x_1, \ldots, x_n.N')$ is an M that satisfies this equation and hence also satisfies the very first equation.

7. Call-by-value

We have seen one kind of reduction relation $\to_\beta$ on λ-terms, and this can rightfully be called the "standard" or, at least, "most basic" notion of reduction associated with the untyped λ-calculus. However, there are many others that each have a part to play in different aspects of the theory. Some notable examples are head-reduction and leftmost reduction, but we will not have time to look at these in our unit. One that has been particularly influential in the world of programming languages is *call-by-value reduction*.

The idea of call-by-value is that a redex $(\lambda x.M)N$ can only be contracted if N has already be evaluated, which is to say that N is a value. What is a value, you may well ask. The answer differs from programming language to programming language, but for pure untyped λ-calculus we usually say that a term M is a value just if it is *not* an application.

Definition 7.1 (Call-by-value)**.** *We say that a term V is a **value** just if it is a variable or an abstraction.*

*The (weak, right-to-left) **one-step call-by-value** reduction relation is the relation $\to_v$ defined inductively by the following rules, in which V is required to be a value.*

$$\frac{}{(\lambda x.M)V \to_v M[V/x]}\ (Redex)$$

$$\frac{M \to_v M}{MV \to_v M'V}\ (AppL) \qquad \frac{N \to_v N'}{MN \to_v MN'}\ (AppR)$$

*Then, (weak, right-to-left) **call-by-value** reduction is the relation $\twoheadrightarrow_v$ defined inductively as follows:*

$$M \to_v N\ \frac{}{M \twoheadrightarrow_v N}\ (Step)$$

$$\frac{}{M \twoheadrightarrow_v M}\ (Refl) \qquad \frac{M \twoheadrightarrow_v P \quad P \twoheadrightarrow_v N}{M \twoheadrightarrow_v N}\ (Trans)$$

You'll notice I added a couple of extra qualifiers to the name of the definition: weak and right-to-left. That's because there are many variations of call-by-value that are appropriate to different situations.

This version is *weak* because it does not allow for reduction under a λ. If you recall the definition of β-reduction, there is an additional rule which allows $\lambda x.M$ to make a step to $\lambda x.M'$ whenever M can make a step to M'. By omitting a rule of that shape we disallow computation from happening in the body of an abstraction, and reduction relations that do this are usually referred to as weak. You may have heard of "weak head normal forms" in relation to Haskell, these are those Haskell expressions that cannot make a step (are in normal form) under "weak head reduction". I won't define weak head reduction here, but you can infer from the name that it does not allow reduction in the body of abstractions, which sounds about right for Haskell.

The other qualifier, *right-to-left* means we are enforcing an order on the evaluation of compound expressions (applications). The argument N of an application MN must be evaluated before the operator M. This is ensured by requiring, in the rule (AppL), that a compound term MN can only make a step via M when N is already a value. In many cases functional programming languages do not specify a particular order on the evaluation of applications (although particular compilers will choose one) but, if my memory serves me correctly, the call-by-value reduction employed by O'Caml is weak and right-to-left.

Notice that CBV reduction is essentially a more restricted version of β-reduction, in the sense that if $M \twoheadrightarrow_v N$, then also $M \twoheadrightarrow_\beta N$, but not necessarily conversely. That is because the rules defining $\rightarrow_v$ are all restrictions of rules of the same shape that define $\rightarrow_\beta$.

We can see the difference directly in the behaviour of the Y combinator. Consider the application $Y(\lambda xy.y)z = (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))(\lambda xy.y)z$. There are infinite β-reduction sequences starting from this term, but there are also β-reduction sequences that terminate:

$$\begin{array}{lll} \mathbf{Y}\,(\lambda xy.y)\,z & \rightarrow_\beta & (\lambda x.(\lambda xy.y)(xx))(\lambda x.(\lambda xy.y)(xx))\,z \\ & \rightarrow_\beta & (\lambda xy.y)(\lambda x.((\lambda xy.y)(xx))(\lambda x.(\lambda xy.y)(xx)))\,z \\ & \rightarrow_\beta & (\lambda y.y)\,z \\ & \rightarrow_\beta & z \end{array}$$

Under CBV, however, an infinite reduction sequence is forced. The subterm $(\lambda xy.y)$ never plays an active part in the computation:

$$\begin{array}{lll} \mathbf{Y}\,(\lambda xy.y)\,z & \rightarrow_v & (\lambda x.(\lambda xy.y)(xx))(\lambda x.(\lambda xy.y)(xx))\,z \\ & \rightarrow_v & (\lambda xy.y)(\lambda x.((\lambda xy.y)(xx))(\lambda x.(\lambda xy.y)(xx)))\,z \\ & \rightarrow_v & (\lambda xy.y)((\lambda xy.y)(\lambda x.((\lambda xy.y)(xx))(\lambda x.(\lambda xy.y)(xx))))\,z \\ & \twoheadrightarrow_v & \cdots \end{array}$$

A consequence of this is that, under CBV-reduction, we cannot use **Y** as a fixed point combinator in order to implement recursion. However, there are variations on **Y** that will work happily. A well-known call-by-value fixed point combinator is **Z**:

$$\lambda f.(\lambda x.f(\lambda z.xxz))(\lambda x.f(\lambda z.xxz))$$

There's nothing I like better than opening up the Clojure read-eval-print-loop and using it to implement my favourite recursive function **Z K**:

```
user=> (def K (fn [x] (fn [y] (println "Yum! I love to eat a" (type y)) x)))
#'user/K

user=> (def Z (fn [f] ((fn [x] (f (fn [z] ((x x) z))))(fn [x] (f (fn [z] ((x x) z))))) ) )
#'user/Z
```

By the way, Clojure (and Scheme before it) are λ-calculi extended with extra commands for convenience, like **println**. The Clojure expression (**fn** [y] (**println** "Yum! I love to eat a" (**type** y)) x) is the function that takes y as input, prints a string disclosing the Java type of y and then returns x.

```
user=> (((((((Z K) 1) 2) "food") K) java.io.File) java.lang.System)
Yum! I love to eat a java.lang.Long
Yum! I love to eat a java.lang.Long
Yum! I love to eat a java.lang.String
Yum! I love to eat a user$K
Yum! I love to eat a java.lang.Class
Yum! I love to eat a java.lang.Class
```

Haha! Take that, Java framework classes!

8. Confluence

It may surprise you to hear, but at no point in the course so far have we ever given any proof that two terms M and N are *not* convertible. It is actually quite difficult to do without a better understanding of the structure of β-convertibility. For example, in the previous chapter I claimed that $(\lambda x.\mathbf{Add}\, x\, 1)\ulcorner n\urcorner \neq_\beta \ulcorner n\urcorner$, but how can we prove this? Surely a good start is to recognise that $(\lambda x.\mathbf{Add}\, x\, 1)\ulcorner n\urcorner =_\beta \ulcorner n+1\urcorner$. Since convertibility is transitive, it is enough to show that $\ulcorner n+1\urcorner \neq_\beta \ulcorner n\urcorner$.

We can show that an object is a member of an inductive set very easily, by giving a proof tree. However, inductive sets do not give us easy tools that can be used to show *non-membership* in the set. A complication in this case is that sometimes convertibility of two terms is justified somehow indirectly — e.g. because there is some common term that reduces to them both. How do we know that there is not some term P such that $\ulcorner n\urcorner \twoheadleftarrow_\beta P \twoheadrightarrow_\beta \ulcorner n+1\urcorner$? If there was such a P, then we could justify $\ulcorner n\urcorner =_\beta \ulcorner n+1\urcorner$.

Thankfully (for our sanity), there is no such term P and it really is the case that $\ulcorner n\urcorner \neq_\beta \ulcorner n+1\urcorner$. In the rest of this chapter, we will develop a theorem that helps us to prove this and many other non-convertibility results very easily. This tool will be a seminal theorem of the pure, untyped λ-calculus.

Theorem 8.1 (Church-Rosser)**.** *For all terms M, N and P: if $M \twoheadrightarrow_\beta N$ and $M \twoheadrightarrow_\beta P$ then there exists a term Q such that $N \twoheadrightarrow_\beta Q$ and $P \twoheadrightarrow_\beta Q$.*

We can read it as saying that, no matter which reduction path you choose to take, there is always some common term Q that joins them.

The rest of this chapter is devoted to proving this result. It will be our first taste of the depth hidden behind our quite simple definitions.

8.1 Abstract rewriting on λ-terms

There is a general theory of reduction relations called *abstract rewriting*. Abstract rewriting seeks to prove results about reduction relations in general, rather than about a specific reduction relation like β-reduction. We have seen β-reduction and CBV-reduction and, in proving the Church-Rosser theorem, we will need to introduce a third kind of reduction. Therefore, it is helpful to develop a tiny bit of the theory of abstract rewriting, so far as it applies to reduction relations over

λ-terms. In the following I will use the long right arrow $\longrightarrow$ to refer to a reduction relation generically.

Based on what we have seen, a first observation that we can make is that given any one-step reduction relation, it is possible to obtain the many-step version in a standard way, namely by taking the *reflexive transitive closure*.

Definition 8.1. *Let $\longrightarrow \subseteq \Lambda \times \Lambda$ be a reduction relation on terms. We write $\longrightarrow\!\!\!\!\!\rightarrow$ for the **reflexive, transitive closure** of $\longrightarrow$, which is the relation defined inductively by:*

$$\frac{M \longrightarrow N}{M \longrightarrow\!\!\!\!\!\rightarrow N}\ (Step) \qquad \frac{}{M \longrightarrow\!\!\!\!\!\rightarrow M}\ (Refl) \qquad \frac{M \longrightarrow\!\!\!\!\!\rightarrow P \quad P \longrightarrow\!\!\!\!\!\rightarrow N}{M \longrightarrow\!\!\!\!\!\rightarrow N}\ (Trans)$$

*We say that a term M is in **normal form** with respect to $\longrightarrow$ just if there is no term N such that $M \longrightarrow N$. We say that a term M is **normalisable** with respect to $\longrightarrow$ if there is some term N in normal form with respect to $\longrightarrow$ and $M \longrightarrow\!\!\!\!\!\rightarrow N$.*

As an aside, *closure* is a operation that applies to any relation R and it means constructing the least relation that contains R and also satisfies certain rules that define the kind of closure. So the reflexive transitive closure of the relation $\longrightarrow$ is the least relation that contains $\longrightarrow$ and is closed under the reflexivity and transitivity rules. This is exactly the set defined inductively by those rules and a rule, here (Step), whose purpose is to include the original relation into the new one.

This definition of relexive transitive closure gives us a notion of reduction that is, in a sense, based on trees because to justify a pair M, N being related by $\longrightarrow\!\!\!\!\!\rightarrow$ we would give a proof tree using the rules. However, due to the shape of the rules, it can also be convenient to think of $\longrightarrow\!\!\!\!\!\rightarrow$ in terms of sequences of reductions.

Lemma 8.1 (Sequentialisation of reduction)**.** *The relation $P \longrightarrow\!\!\!\!\!\rightarrow Q$ holds if and only if there exists a natural $n \in \mathbb{N}$ and a sequence of terms $M_0, \ldots, M_n$ such that:*

- $P = M_0$,
- $Q = M_n$
- *and, for all $i \in [0..n-1]$, $M_i \longrightarrow M_{i+1}$*

Proof. We prove the iff in two parts, but the right-to-left direction is clear enough.

In the left-to-right direction, the proof is by induction on $P \longrightarrow\!\!\!\!\!\rightarrow Q$.

(Step) In this case, $P \longrightarrow Q$ and so we take $n = 1$ and the sequence is P, Q.

(Refl) In this case, $P = Q$ and so we take $n = 0$ and the sequence is just P.

(Trans) In this case, let M be arbitrary. We assume the induction hypothesis:

(IH1) There is some k_1 and sequence $M_0 \longrightarrow \cdots \longrightarrow M_{k_1}$ with $M_0 = P$ and $M_{k_1} = M$.

(IH2) There is some k_2 and sequence $N_0 \longrightarrow \cdots \longrightarrow N_{k_2}$ with $N_0 = M$ and $N_{k_2} = Q$.

Therefore, we take $n = k_1 + k_2$ and sequence $M_0, \ldots, M_{k_1}, N_1, \ldots, N_{k_2}$.

□

Confluence and the diamond property

We note that the Church-Rosser theorem above is an instance of the more abstract property of rewriting called *confluence*.

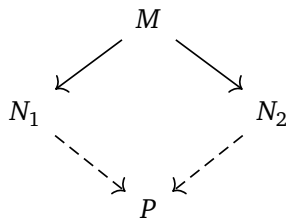
Definition 8.2 (Confluence)**.** *Let $\longrightarrow \subseteq \Lambda \times \Lambda$ be a reduction relation on terms. We say that $\longrightarrow$ is **confluent** just if, for all $M, N_1, N_2 \in \Lambda$, $M \longrightarrow\!\!\!\!\!\rightarrow N_1$ and $M \longrightarrow\!\!\!\!\!\rightarrow^* N_2$ implies the existence of some P such that $N_1 \longrightarrow\!\!\!\!\!\rightarrow^* P$ and $N_2 \longrightarrow\!\!\!\!\!\rightarrow^* P$.*

Notice that confluence is a property of reduction relations on terms, in general. Since we ultimately want to prove the Church-Rosser theorem, we are interested in the confluence of the particular reduction relation $\rightarrow_\beta$ but we could equally well ask about the confluence of $\rightarrow_v$. Also notice the slightly unusual (in my opinion) terminology, confluence is a property of $\longrightarrow$ but it is defined entirely in terms of $\longrightarrow\!\!\!\!\!\rightarrow$.

Confluence is typically very difficult to prove directly. You could try to do some kind of double induction on $M \longrightarrow\!\!\!\!\!\rightarrow N_1$ and $M \longrightarrow\!\!\!\!\!\rightarrow N_2$, in order to show the existence of the term P, but I assure you that you won't get far. However, it is fairly straightforward to show that confluence of some abstract reduction relation $\longrightarrow$ follows from another, simpler property of $\longrightarrow$ called the diamond property.

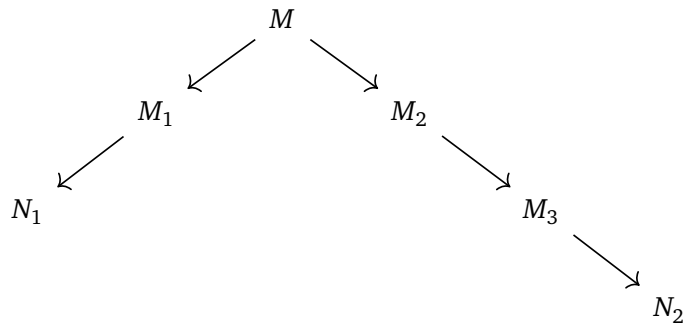
Definition 8.3. *Let $\rightarrow \subseteq \Lambda \times \Lambda$ be a reduction relation on terms. We say that $\longrightarrow$ satisfies the **Diamond Property** just if, for all $M, N_1, N_2 \in \Lambda$, $M \longrightarrow N_1$ and $M \longrightarrow N_2$ implies the existence of some P such that $N_1 \longrightarrow P$ and $N_2 \longrightarrow P$.*

So the diamond property is just like confluence, but it only concerns one step of reduction. In fact, we could define confluence of $\longrightarrow$ means exactly that $\longrightarrow\!\!\!\!\!\rightarrow$ satisfies the diamond property. The diamond property is so called because it turns out that it is helpful to visualise the property in the following way:

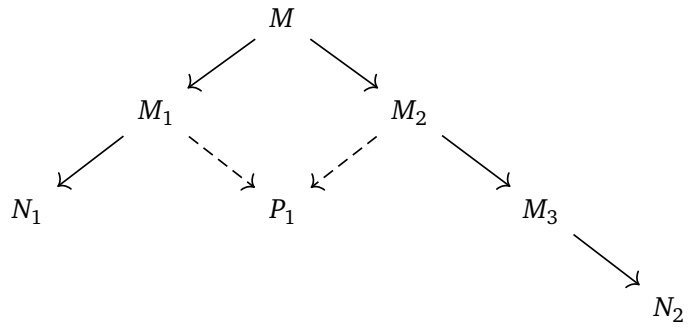


It's obviously not a perfect representation of the logical content of the definition, but the solid lines are meant to represent one-step reductions quantified by "for all" and the dashed lines those quantified by "exists".

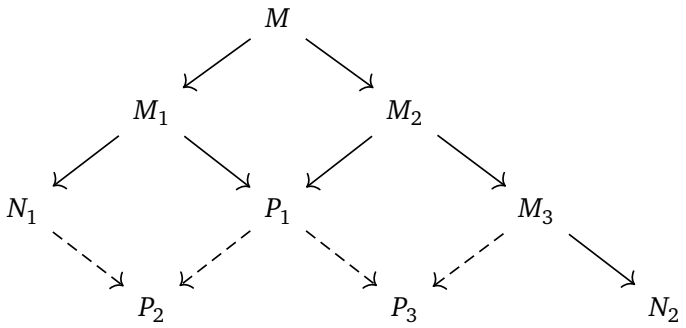
Proving that some reduction relation $\longrightarrow$ satisfies the diamond property is a very convenient intermediate step towards proving that $\longrightarrow$ is confluent. Let me demonstrate by an (abstract) example. For confluence we need to show that $M \longrightarrow\!\!\!\!\!\rightarrow N_1$ and $M \longrightarrow\!\!\!\!\!\rightarrow N_2$ guarantees us the existence of some P which joins the two reductions so that $N_1 \longrightarrow\!\!\!\!\!\rightarrow P$ and $N_2 \longrightarrow\!\!\!\!\!\rightarrow P$. We know that, whenever we have a reduction sequence $M \longrightarrow\!\!\!\!\!\rightarrow N$, it is comprised of a specific number, n, of one-step reductions (which may be zero). So let me depict the hypotheses of confluence by the following diagram, for the specific example in which $M \longrightarrow N_1$ is comprised of two one-step reductions and $M \longrightarrow N_2$ is comprised of three one-step reductions.



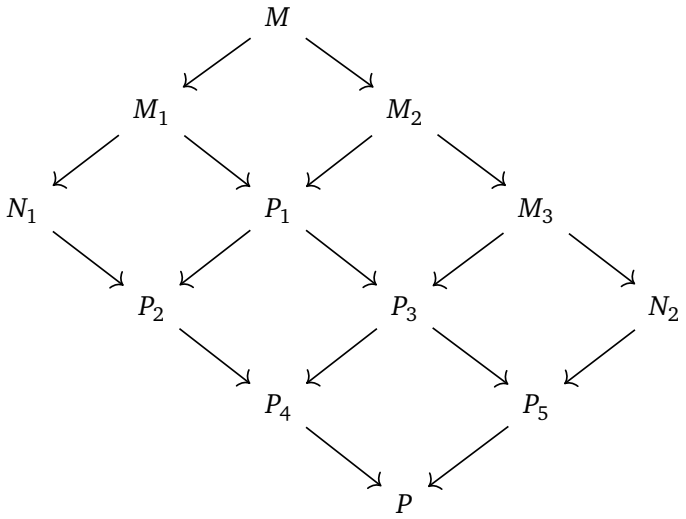
Recall that confluence requires us to find some joining term P. I claim that, if $\longrightarrow$ satisfies the diamond property, then it is straightforward to find this P. So let's assume that $\longrightarrow$ satisfies diamond. Is there an opportunity to exploit that property here? Indeed, we have the situation described by the hypotheses to diamond at the top of the diagram: there is some term M that makes two one-step reductions $M \longrightarrow M_1$ and $M \longrightarrow M_2$. Therefore, since we assumed that $\longrightarrow$ satisfies the diamond property, it follows that there is some term, let's call it P_1, such that $M_1 \longrightarrow P_1$ and $M_2 \longrightarrow P_1$. Diagrammatically, we now know that the situation is actually like:



The sharp eyed amongst you will notice that this re-rendering of the same situation has turned up more opportunities to deduce the existence of terms using the Diamond property. By closing the diamonds with apex M_1 and M_2 we have:



We can continue in this way until we have closed all the diamonds. The result is this rather pretty grid:



Therefore, it follows from the Diamond Property that, if I have a two-step reduction $M \longrightarrow^* N_1$ and a three step reduction $M \longrightarrow^* N_2$, there is a joining term P such that $N_1 \longrightarrow^* P$ and $N_2 \longrightarrow^* P$. However, I hope that this proof for the 2,3-step case convinces you that I can perform the same reasoning, drawing similar diagrams, for any pair of m- and n-step reduction sequences. Since most people are convinced by this style of argument, and because similar arguments are useful in other areas of abstract rewriting, they have earned themselves a name: *diagram chasing*.

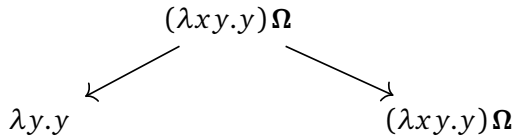
Lemma 8.2 (Diamond implies Confluence)**.** *Let $\longrightarrow$ be a reduction relation on terms. If $\longrightarrow$ satisfies the diamond property then $\longrightarrow$ is confluent.*

Proof. By diagram chase. □

It's also possible to prove the result by induction, but it is quite boring.

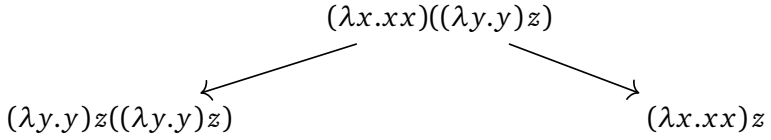
All that remains, then is to show that $\rightarrow_\beta$ satisfies the diamond property. Unfortunately, it doesn't. It actually comes very close, in the sense that there are many *specific* pairs of one-step reductions $M \rightarrow_\beta N_1$, $M \rightarrow_\beta N_2$ which can be joined by some P with $N_1 \rightarrow_\beta P$ and $N_2 \rightarrow_\beta P$; but not all of them. There are essentially two kinds of reduction that are problematic.

One problematic kind of reduction is when M makes a step by contracting a redex that results in another redex being erased and the other reduction from M contracts the erased redex. An example is as follows, recall that $\mathbf{\Omega} = (\lambda x.xx)(\lambda x.xx)$ reduces to itself in one step:



Then these two terms cannot be joined in exactly one step. There is a common term that they both reduce to $\lambda y.y$, but the diamond property requires that they both reach the common term after *exactly one step*, which is impossible for the term on the left.

The other kind of problematic reduction is when M makes a step that duplicates a redex and the second reduction from contracts that redex. An example is:



In this case, there is a joining term zz, and the right-hand term can reach zz in one step, but it requires two steps for the left hand term to reach it. Since the diamond property requires that the joining term must be reached in exactly one step, this is a violation.

8.2 Parallel-β Reduction

In summary, $\rightarrow_\beta$ does not satisfy the diamond property because, intuitively:

- Sometimes a reduction step erases a redex which can cause joining to require 0 redexes to be contracted on one side or the other.
- Sometimes a reduction step duplicates a redex which can cause joining to require contracting more than a single redex.

This motivates a new variant of β-reduction, in which rather than making a step by contracting *exactly one redex*, instead a step consistes of contracting *any subset* of the redexes in the term, including the empty subset. Afterwards we will come to why this is useful for showing that $\rightarrow_\beta$ is confluent.

Definition 8.4 (Parallel-β)**.** *The **one-step parallel-β** reduction relation, written $\rightarrow_p$ infix, is defined inductively by the following rules:*

$$\frac{M \rightarrow_p M' \quad N \rightarrow_p N'}{(\lambda x.M)N \rightarrow_p M'[N'/x]}\ (Redex)$$

$$\frac{}{x \rightarrow_p x}\ (Var) \qquad \frac{M \rightarrow_p M' \quad N \rightarrow_\beta N'}{MN \rightarrow_p M'N'}\ (App) \qquad \frac{M \rightarrow_p N}{\lambda x.M \rightarrow_p \lambda x.N}\ (Abs)$$

It should be clear by studying the rules that any justification for $\rightarrow_\beta$ can be replayed as a justification for $\rightarrow_p$, so that $M \rightarrow_\beta N$ implies $M \rightarrow_p N$. However, the rules allow for strictly more one-step reductions that those of β. The (Var) rule, in combination with the others, allows for a step to be made by contracting no redexes. The rules (App) and (Redex) allow for contracting redexes in different subterms simultaneously. The result is that $M \rightarrow_p N$ can be achieved by contracting any subset of the redexes occurring in the term.

The idea of parallel-β is that it overcomes the shortcomings of β with respect to the diamond property. Indeed, we will now show that parallel-β satisfies diamond. We can save some time in the proof by making one further definition. This last definition exploits the fact that, given two one-step using parallel-β reductions from M, we can always choose the joining term P to be M with all redexes contracted.

Definition 8.5. *Given a term M, we define its **maximal parallel contraction**, written M^*, as the function on terms defined by structural recursion:*

$$\begin{array}{rcll} x^* & = & x & \\ (PQ)^* & = & N^*[Q^*/x] & \text{if } P \text{ is an abstraction } \lambda x.N \\ (PQ)^* & = & P^*Q^* & \text{otherwise} \\ (\lambda x.P)^* & = & \lambda x.P^* & \end{array}$$

I hope that it is clear from this definition that, given any term M, $M \rightarrow_p M^*$. What is not so clear, but which is nevertheless true, is that the notion of maximal contraction plays well with substitution. This is the subject of the following lemma, whose proof is one of the exercises of this week.

Lemma 8.3. *If $M \rightarrow_p M'$ and $N \rightarrow_p N'$ then $M[N/x] \rightarrow_p M'[N'/x]$.*

With this and the idea of maximal contraction, it is fairly straightforward to be convinced that parallel-β satisfies the diamond property.

Lemma 8.4 (Parallel-β is diamond)**.** *The reduction relation $\rightarrow_p$ satisfies the Diamond Property.*

Proof. To show this, we first show an intermediate result, called the triangle property: if $M \rightarrow_p N$ then $N \rightarrow_p M^*$. The proof is by induction on $M \rightarrow_p N$.

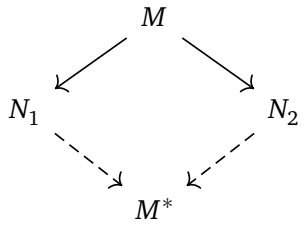
(Var) In this case $M = N = x = M^*$ so the result follows from rule (Var).

(Redex) In this case, M has shape $(\lambda x.P)Q$ and N has shape $P'[Q'/x]$. We assume the induction hypothesis: $P' \rightarrow_p P^*$ and $Q' \rightarrow_p Q^*$. Therefore it follows from Lemma 8.3 that $P'[Q'/x] \rightarrow_p P^*[Q^*/x]$.

(App) In this case, M has shape PQ and N has shape $P'Q'$. We assume the induction hypothesis: $P' \rightarrow_p P^*$ and $Q' \rightarrow_p Q^*$. Therefore, by (App), $P'Q' \rightarrow_p P^*Q^*$ and this is equal to $(PQ)^*$ by definition.

(Abs) In this case, M has shape $\lambda x.P$ and N has shape $\lambda x.P'$. We assume the induction hypothesis that $P' \rightarrow_p P^*$. Therefore, by (Abs), $\lambda x.P' \rightarrow_p \lambda x.P^*$. By definition $\lambda x.P^* = (\lambda x.P)^*$.

We can now obtain the diamond from two triangles, taking M^* as witness:



□

Corollary 8.1. *Parallel-β reduction $\rightarrow_p$ is confluent.*

Proof. Follows immediately from Lemmas 8.2 and 8.4. □

8.3 Proof of Church-Rosser

So, we have defined a kind of sibling to β-reduction, $\rightarrow_p$, that is confluent, but how does that help us to show that $\rightarrow_\beta$ itself is confluent? The reason is that taking the reflexive transitive closure of $\rightarrow_\beta$ and taking the reflexive transitive closure of $\rightarrow_p$ gives the same relation.

Lemma 8.5. *$M \twoheadrightarrow_\beta N$ iff $M \twoheadrightarrow_p N$*

Proof. We could prove both directions by induction, but it is convincing enough to reason as follows:

- In the forward direction, if I have some β-reduction sequence $M_1 \to_\beta M_2 \to_\beta \cdots \to_\beta M_k$ then, since each β-reduction step is also a $\to_p$-reduction step (in which I contract a single subset of redexes), the same sequence of terms is a valid $\to_p$-reduction sequence $M_1 \to_p M_2 \to_p \cdots \to_p M_k$.
- In the backward direction, if I have some $\to_p$-reduction sequence $M_1 \to_p M_2 \to_p \cdots \to_p M_k$ then I can always construct a new (typically longer) $\to_\beta$-reduction sequence $N_1 \to_\beta N_2 \to_\beta \cdots \to_\beta N_m$ with $M_1 = N_1$ and $M_k = N_m$. I can do this by translating each individual $\to_p$-step into a sequence of β-steps by simply sequentially contracting the same subset of redexes.

□

Then, we can finally prove the Church-Rosser Theorem.

Proof. We are required to show that $\to_\beta$ is confluent, which is to say that $\twoheadrightarrow_\beta$ has the diamond property. By Lemma 8.5, this is the same as showing that $\twoheadrightarrow_p$ has the diamond property. However, we know that $\twoheadrightarrow_p$ has the diamond property because in Corollary 8.1 we established that $\to_p$ is confluent. □

8.4 Confluence and Conversion

Let me now take you back to our original motivation for proving Church-Rosser. I told you that Church-Rosser, which we now understand to mean the confluence of $\to_\beta$, is a very useful tool that we can use in order to argue that certain pairs of terms are not convertible. To see how that works, we need another result from the theory of abstract rewriting (but here we just prove it for $=_\beta$).

Lemma 8.6 (Sequentialisation of Conversion)**.** *$P =_\beta Q$ if and only if there exists $k \in \mathbb{N}$ and some sequence of terms $M_0, N_0, \ldots, N_k, M_{k+1}$ such that $M_0 = P$ and $M_{k+1} = Q$ and, for all $i \in [0..k]$, $N_i \twoheadrightarrow_\beta M_i$ and $N_i \twoheadrightarrow_\beta M_{i+1}$.*

Proof. The backward direction is quite clear. In the foward direction, the proof is by induction on $P =_\beta Q$.

(Refl) In this case, P and Q are the same term, we take $M_0 = N_0 = M_1$ to be P.

(Step) In this case, $P \to_\beta Q$ and we take $n = 0$ and $M_0 = M_1 = Q$ and $N_0 = Q$.

(Sym) In this case we assume the induction hypothesis: there is some m and sequence $M'_0, N'_0, \ldots, N'_m, M'_{m+1}$ with $M'_0 = Q$ and $M'_{m+1} = P$ and the required relationship between the terms. We can just reverse this sequence to obtain the required sequence for $P =_\beta Q$.

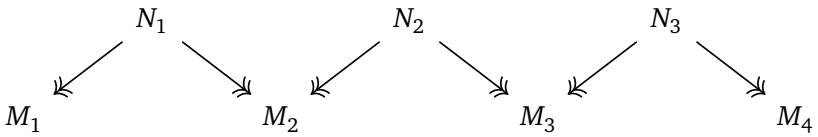
(Trans) In this case, let R be arbitary. We assume the induction hypothesis that there are sequences $M'_0, N'_0, \ldots, N'_m, M'_{m+1}$ and $M''_0, N''_0, \ldots, N''_n, M''_{n+1}$ with

$M'_0 = P$, $M'_{m+1} = R = M''_0$, $M_{n+1} = Q$ and the correct relationship between terms. We obtain the desired sequence for $P =_\beta Q$ as:

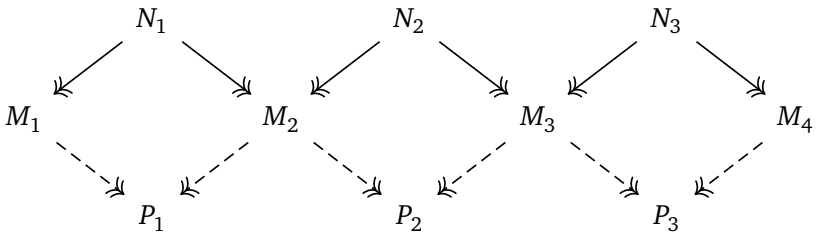
$$M'_0, N'_0, \ldots, N'_m, M''_0, N''_0, \ldots, N_n, '' M''_{n+1}$$

□

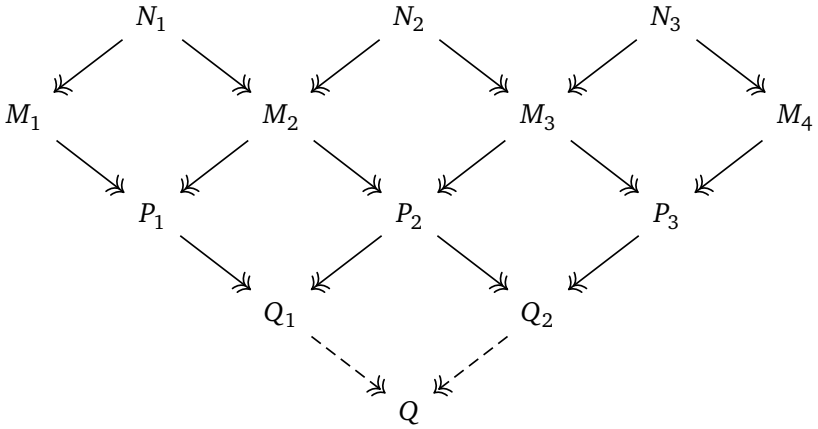
The sequentialisation of conversion allows us to write a statement of convertibility such as $M =_\beta N$ as a sequence of peaks. Let's suppose the lemma gives us $k = 3$:



Then, we can apply the Church-Rosser theorem 3 times to obtain the existence of three terms P_1, P_2 and P_3 as in the following diagram:



If we continue, in this way we eventually find that:



I hope you are convinced that, if I have some sequence of terms:

$$M_1 \twoheadleftarrow_\beta N_1 \twoheadrightarrow_\beta M_2 \cdots M_k \twoheadleftarrow_\beta N_k \twoheadrightarrow_\beta M_{k+1}$$

with forward and backward reduction sequences alternating in this way, then I can draw them as a sequence of peaks and repeat this exercise in order to obtain some term Q which all of them reduce to. Hence, if $M =_\beta N$, it follows that I can find a sequence of the correct shape, apply the Church-Rosser theorem some finite number of times and ultimately conclude that there is some term Q such that $M \twoheadrightarrow_\beta Q$ and $N \twoheadrightarrow_\beta Q$.

Corollary 8.2. *$M =_\beta N$ iff there is some term Q such that $M \twoheadrightarrow_\beta Q$ and $N \twoheadrightarrow_\beta Q$.*

Proof. The forward direction follows from the Church-Rosser theorem by diagram chase. The backward direction follows from the definition. □

Lemma 8.7. *For all $n \in \mathbb{N}$, $\ulcorner n \urcorner \neq_\beta \ulcorner n+1 \urcorner$.*

Proof. Let n be a natural number. Both of $\ulcorner n \urcorner$ and $\ulcorner n+1 \urcorner$ are in β-normal form and are distinct, so there is no term Q that satisfies *both* $\ulcorner n \urcorner \twoheadrightarrow_\beta Q$ and $\ulcorner n+1 \urcorner \twoheadrightarrow_\beta Q$ (the only term Q that satisfies the first is $\ulcorner n \urcorner$ and the only Q that satisfies the second is $\ulcorner n+1 \urcorner$ and these are not literally the same term). Hence, the result follows from the Church-Rosser (via Corollory 8.2). □

Of course the same proof would work for any two normal forms. So we can also conclude that, e.g. $\mathbf{K} \neq_\beta \mathbf{I}$. This, in turn allows us to prove more general non-existence results.

Corollary 8.3. *There is no term F such that, for all terms M:*

- *$F\,M =_\beta \ulcorner 0 \urcorner$ if M is an application*
- *$F\,M =_\beta \ulcorner 1 \urcorner$ otherwise*

Proof. Assume there is such a term F. Then $\ulcorner 0 \urcorner =_\beta F\,(\mathbf{II}) =_\beta F\,(\mathbf{I}) =_\beta \ulcorner 1 \urcorner$, contradicting Lemma 8.7. □

9. Computability

The week we will see that the lambda calculus is capable of expressing any computable function. Now, when I speak of computability and the notion of computable function, you probably have in mind the Turing machine, but there are many equivalent definitions of what it means to compute a function. Indeed, Turing's machines were not even the first, historically, of these various equivalent formalisations.

In the late 1920s and early 1930s a great unsolved problem preoccupied mathematical logicians. This problem is called the *entscheidungsproblem*, which is German for "decision problem". Nowadays we are very familiar with decision problems, but back then the phrase referred to one in particular, which was the problem of determining the validity of a statement of first-order logic. The fact that there might be a step-by-step method, requiring no insight on the part of the practicioner, and by which one could determine the truth of any mathematical conjecture was an *entirely plausible* to the mathematicians at the turn of the 20th century.

The entscheidungsproblem was one pillar of the famous programme of David Hilbert. Hilbert really put meta-mathematics on the map. In the half-century before, mathematicians had become interested in putting more rigour into mathematics. On the one hand they were concerned by certain infinite constructions, like the limits of infinite series that arise in analysis, on the other there had been many important developments in mathematical logic which put a greater emphasis on axiomatisation. Hilbert was interested in analysing the axiomatisations themselves to understand when axioms could be said to be independent, when systems of axioms could be said to be consistent or complete and whether there was a procedure that could be followed in order to calculate the consequences of a set of axioms — hence the entsheidungsproblem.

If you believed that the decision problem has a positive solution, you need only propose an algorithm that will solve any instance. Even though the digital computer would not be invented for many years to come, logicians of the 1930s were quite able to recognise an algorithm when they saw one — (roughly speaking) a procedure of discrete steps, requiring no insight and which, if followed faithfully, would yield the desired result. However, if you believed that the decision problem has a negative solution things are a little more difficult. To show that there can be *no* algorithm capable of solving the problem requires that you

can make precise the class of all algorithms. What else can you do? You would need to give a mathematical definition of the intuitive notion of, as it was sometimes called at the time, *effective calculability*, which was thought of as meaning the quality of being solvable by an algorithm.

The first serious proposal for a definition of effective calculability was given by Alonzo Church, using his recently invented λ-calculus. At the start of the 1930s Church, who was faculty in Princeton, had designed a new system of logic, which I will say a little more about in Week 7. This system contained, at its very heart, the lambda notation and the notion of β-convertibility. Church took on a PhD student, Stephen Kleene and set him the task of developing number theory within his system of logic. This required making precise the idea that certain lambda terms *define* certain number theoretic functions. This is called λ-definability, and we have used it already, implicitly, when we looked at Church's system of numerals.

Definition 9.1 (λ-definability)**.** *A function $f : \mathbb{N} \times \cdots \times \mathbb{N} \to \mathbb{N}$ on k-tuples of natural numbers is said to be* ***λ-definable*** *just if there exists a λ-term F that satisfies the equation:*

$$F \ulcorner n_1 \urcorner \cdots \ulcorner n_k \urcorner =_\beta \ulcorner f(n_1, \ldots, n_k) \urcorner$$

We have already seen that, for example, addition is λ-definable. The λ-definition that you developed as part of the Week 3 exercises was actually Kleene's first piece of published research!

However, Kleene soon discovered that it was possible to encode general principles of function definition using λ-terms, such as primitive recursion, rather than having to define each individual number-theoretic function from scratch.

Primitive recursion can be thought of as looping where you know at the point you begin the loop how many iterations you will go through. In imperative programming, primitive recursion is realised through for-loops, *for* $i = 1$ *to* x. As long as you don't assign to i or x during the body of the loop, then the number of times you will go around is exactly the value of x at the beginning. In functional programming, definitions of primitive recursive functions are recognisable because the clause for defining the function f on $n+1$ depends only on recursive calls to $f(n)$.

Almost all the "standard" numeric functions are definable using the principle of primitive recursion. For example, addition is primitive recursive. Intuitively, it can be defined by the functional program:

$$\begin{array}{rcl} \mathsf{add}(0, x) & = & x \\ \mathsf{add}(n+1, x) & = & \mathsf{add}(n, x) + 1 \end{array}$$

which only uses a recursive call to n in the defining clause for input $n+1$.

However, Kleene knew that there were other functions that are not primitive recursive, such as Ackermann's function. At some point he realised that he could make definitions in λ-calculus of functions of the form "f maps input x to the least number y such that some primitive recursive relation holds between x and y".

Although it looks a bit strange, this mode of definition, called *minimisation*, has become a fundamental operation in recursion theory. It's exactly what you need in order to give the definition of functions that are computable but not primitive recursive, like Ackermann's function.

His student's success was starting to make Church think that perhaps λ-definability was a good match for this intuitive idea of effective calculability. In other words, that every "algorithm" can be described by a λ-term. Kleene writes that Gödel, who had recently published his seminal incompleteness results, came to visit Princeton. Church met with him and told him of this idea of *defining* the idea effective calculability using λ-definability.

Gödel was not at all convinced. I think that this must have quite annoyed Church because his response, Kleene writes, was to challenge Gödel to invent his own definition: whatever you come up with will be shown, Church promised, to already be subsumed by my notion of λ-definability.

The following year Gödel gave a series of lectures on his own idea, based partly on the work of Herbrand, which was a precisely defined class of functions that were called *general recursive* and nowadays are usually called *total recursive*. Kleene studied this class and realised that any general recursive function can be defined using a combination of primitive recursion and his minimisation operator. This observation not only gives a somewhat cleaner formulation of the Gödel's general recursive functions but also, since primitive recursion and minimilisation are both λ-definable, it made good on Church's promise.

9.1 Total recursive functions

This reformulation of general recursive functions is nowadays sometimes called the μ-recursive functions or just the *total recursive functions*. It is important to remember that this is a collection of *total* functions. At the start of its life, computability theory was only concerned with total functions (recursiveness), only later was it generalised to partial functions which is how we think of it today.

The idea is to give a kind of *axiomatisation* of what it means to be a computable function. A function is total recursive if it is either one of the initial functions, or can be obtained from them by certain methods of combination.

Definition 9.2 (Initial functions)**.** *The **initial functions** are a set of numeric functions consisting of:*

- *The **zero** is the function* $\mathsf{z} : \mathbb{N} \to \mathbb{N}$ *satisfying* $\mathsf{z}(n) = 0$.
- *The **successor** is the function* $\mathsf{s} : \mathbb{N} \to \mathbb{N}$ *satisfying:* $\mathsf{s}(n) = n + 1$.
- *The family of **projection** functions, written* $\mathsf{p}^i_m : \mathbb{N}^m \to \mathbb{N}$*, satisfy:* $\mathsf{p}^i_m(n_1, \ldots, n_m) = n_i$.

Notice that the identity function $\mathsf{id} : \mathbb{N} \to \mathbb{N}$ is a trivial instance p^1_1 of projection.

Then there are three methods of combination, by which we may obtain new recursive functions from old. The first method for combining functions should come as no surprise.

Definition 9.3 (Composition)**.** *Given a function $g : \mathbb{N}^m \to \mathbb{N}$ and a family of functions $f_1, \ldots, f_m : \mathbb{N}^k \to \mathbb{N}$, their **composition**, written $\mathsf{C}[g, f_1, \ldots, f_m] : \mathbb{N}^k \to \mathbb{N}$, is the unique function satisfying:*

$$\mathsf{C}[g, f_1, \ldots, f_m](n_1, \ldots, n_k) = g(f_1(n_1), \ldots, f_k(n_k))$$

It looks a bit complicated at first sight, but if you look again you will see it's just a straightforward extension of the composition $g \circ f$ of two functions $f : A \to B$ and $G : B \to C$ to the case in which f and g take tuples of inputs.

More interesting is the combination of functions by primitive recursion. Recall that the idea is that a function is primitive recursive if it can be defined on $n+1$ using only recursive calls to input n, or alternatively, if it can be computed using a sensible for loop (no assignment allowed to the loop counter variables in the body). So, for example, a definition of the form:

$$\begin{aligned} f(0, y) &= e_1 \\ f(n+1, y) &= e_2 \end{aligned}$$

is primitive recursive whenever e_1 contains no recursive calls and e_2 contains only recursive calls of the form $f(n, y)$. We can turn this idea into a precise method for obtaining new functions as a combination of old. Consider e_1 and e_2 above. The expression e_1 depends only on y, in the sense that, to evaluate this expression e_1 you just need to know the value of y (in general). Therefore, we can think of e_1 as specifying a function of one argument. The expression e_2 also depends on y but, additionally, it may contain occurrences of n and occurrences of the recursive call $f(n, y)$. Therefore, there is a sense in which e_2 is a function of three arguments. Given two functions $f_1 : \mathbb{N} \to \mathbb{N}$ (think of e_1) and $f_2 : \mathbb{N}^3 \to \mathbb{N}$ (think of e_2), we can make a primitive recursive function $f : \mathbb{N}^2 \to \mathbb{N}$ by defining:

$$\begin{aligned} f(0, y) &= f_1(y) \\ f(n+1, y) &= f_2(f(n, y), n, y) \end{aligned}$$

A good analogy is to compare the explicit recursive definition of the sum of a list of integers with the definition using foldr.

In general, there is no harm in allowing as many "non-recursive" parameters y as you like.

Definition 9.4 (Primitive Recursion)**.** *Let $f : \mathbb{N}^m \to \mathbb{N}$ and $g : \mathbb{N}^{m+2} \to \mathbb{N}$ be two numeric functions of m and $m+2$ arguments respectively. Let us write $\mathsf{R}[f, g] : \mathbb{N}^{m+1} \to \mathbb{N}$ for the unique numeric function of $m+1$ arguments that satisfies the following equations:*

$$\begin{aligned} \mathsf{R}[f, g](0, y_1, \ldots, y_m) &= f(y_1, \ldots, y_m) \\ \mathsf{R}[f, g](k+1, y_1, \ldots, y_m) &= g(\mathsf{R}[f, g](k, y_1, \ldots, y_m), k, y_1, \ldots, y_m) \end{aligned}$$

For example, addition can be defined as a primitive recursive combination of the identity $\mathsf{p}_1^1 : \mathbb{N} \to \mathbb{N}$ and the function $\mathsf{C}[\mathsf{s}, \mathsf{p}_3^1] : \mathbb{N}^3 \to \mathbb{N}$ which takes a triple of integers and returns the successor of the first component.

The final method of combination is Kleene's minimisation (this was one of the areas where Kleene improved on Gödel's original formulation). Given a function f that is known to return 0 for some input, minimisation is the function that computes the smallest input for which f returns 0.

Definition 9.5 (Minimisation)**.** *Let $f : \mathbb{N}^{m+1} \to \mathbb{N}$ be such that, for all m-tuples $\langle n_1, \ldots, n_m \rangle$, there is some k such that $f(k, n_1, \ldots, n_k) = 0$. Then the function $\mathsf{M}[f] : \mathbb{N}^m \to \mathbb{N}$ obtained by **minimisation** is the unique function defined by:*

$$f(m, n_1, \ldots, n_m) = 0 \qquad \textit{iff} \qquad \mathsf{M}[f](n_1, \ldots, n_m) \leq m$$

This is just a neat way of saying that $\mathsf{M}[f](n_1, \ldots, n_m)$ is the least m such that $f(m, n_1, \ldots, n_m) = 0$.

Where primitive recursion corresponds to bounded looping (the number of iterations is already determined at the start of the loop), minimisation corresponds to *unbounded* looping. I don't know how Kleene thought up this minimisation operation, but one way to think about it is as follows. In imperative programming languages, unbounded looping is implemented through while loops. Unlike for loops, the number of iterations of a while loop is not necessarily known at the point at which the loop is entered, because termination is predicated on some condition being satisfied. Think of a while loop:

$$\mathsf{while}\ e_1\ \mathsf{do}\ e_2$$

Even if you know the values of the all the variables involved at the start of the loop, it's not generally possible to determine the number of iterations that the loop will go around without actually executing it, because it depends on falsifying the condition e_1. On the other hand, *if* we somehow knew at the start of the while loop how many iterations were required for the condition to be violated, *then* we could implement the while loop using a simple for loop.

Let's suppose I have a programming language *without* while loops, but which does have for loops and also has some special syntax for determining the smallest number of executions $e_2; \cdots; e_2$ of a given expression e_2 required in order to make a condition e_1 false:

$$\mathsf{minexec}\ e_2\ \mathsf{violating}\ e_1$$

Then I can simulate a loop of the form $\mathsf{while}\ e_1\ \mathsf{do}\ e_2$ by:

$$\mathsf{for}\ i = 1\ \mathsf{to}\ (\mathsf{minexec}\ e_2\ \mathsf{violating}\ e_1)\ \mathsf{do}\ e_2$$

The minimisation operation is the recursion theoretic equivalent of this special syntax. The moral is: primitive recursion + minimisation = all recursive functions.

Definition 9.6. *The **total recursive functions** is the set $\mathscr{R}_0$ of numeric functions defined inductively by the following rules:*

$$\frac{}{\mathsf{z} \in \mathscr{R}_0}\ (\mathit{Zero}) \qquad \frac{}{\mathsf{s} \in \mathscr{R}_0}\ (\mathit{Succ}) \qquad i \in [1..k]\ \frac{}{\mathsf{p}^i_k \in \mathscr{R}_0}\ (\mathit{Proj})$$

$$\frac{g : \mathbb{N}^k \to \mathbb{N} \in \mathscr{R}_0 \quad f_1, \ldots, f_k : \mathbb{N}^m \to \mathbb{N} \in \mathscr{R}_0}{\mathsf{C}[g, f_1, \ldots, f_k] \in \mathscr{R}_0}\ (\mathit{Comp})$$

$$\frac{f : \mathbb{N}^m \to \mathbb{N} \in \mathscr{R}_0 \quad g : \mathbb{N}^{m+2} \to \mathbb{N} \in \mathscr{R}_0}{\mathsf{R}[f, g] \in \mathscr{R}_0}\ (\mathit{Prim})$$

$$\forall n_1 \cdots n_m . \exists k . f(k, n_1, \ldots, n_m) = 0\ \frac{f : \mathbb{N}^{m+1} \to \mathbb{N} \in \mathscr{R}_0}{\mathsf{M}[f] \in \mathscr{R}_0}\ (\mathit{Min})$$

So, this set $\mathscr{R}_0$ of numeric functions, was Gödel's counter-suggestion as a definition of what it means to be a computable function (or as Church would have said, effectively calculable). I think it has a lot to recommend it. After all, computable functions are meant to be functions on natural numbers and this definition gives a rather clean characterisation, which does not rely on any particular computing device or programming language — there is no syntax involved whatsoever.

In 1936 Church and Kleene published proofs showing that every total recursive function was lambda definable (and vice versa). So, indeed, Church made good on his promise.

Theorem 9.1. *Every total recursive function is already λ-definable:*

$$f \in \mathscr{R}_0 \qquad \textit{implies} \qquad f \textit{ is } \lambda\textit{-definable}$$

Proof. The zero function z is λ-definable by $\lambda n f x . x$. The successor s is λ-definable by $\lambda n f x . f\ (n\ f\ x)$. Assuming that $g : \mathbb{N}^k \to \mathbb{N}$ is λ-definable by some term N and each $f_i : \mathbb{N}^m \to \mathbb{N}$ is λ-definable by some term M_i, the composition $\mathsf{C}[g, f_1, \ldots, f_k]$ is λ-definable by:

$$\lambda n_1, \ldots, n_m . N\ (M_1\ n_1\ \cdots n_m)\ \cdots (M_k\ n_1 \cdots n_m)$$

The λ-definability of primitive recursion and minimisation is the subject of this week's exercises. □

This result, together with its converse, must together have as good a claim as any as marking the birth of theoretical computer science.

9.2 The Church-Turing Thesis

In Church's paper he proposed his thesis, which he now believed more than ever: the precise notion of a λ-definable function is exactly the same as the intuitive idea of an effectively calculable function.

If you believe in Church's thesis, then λ-definability captures precisely the notion of computable function or algorithm. Church showed that the function mapping each first-order sentence to its truth was not λ-definable. Therefore, if you believe the thesis, this constitutes a negative answer to Hilbert's entschiedungsproblem: there can be no decision procedure for first-order validity. Thus, at a stroke, Church defines computability and proves the first undecidability result[1].

We will deviate a little bit from the history at this point and instead prove a much more general result about the decidability of certain sets of λ-terms.

First, I want to talk a little bit about the fact that, throughout what we have done so far, we have assumed that the functions that we are interested in computing are functions on natural numbers. This is completely standard, and it makes sense if you think about it. Your experience of digital computers should suggest to you that all kinds of data — strings, arrays, trees, graphs etc — can all ultimately be represented using sequences of bits. Since we can think of a sequence of bits as a binary numeral, we can thus encode any kind of data as some natural number. Operations on data can then be implemented as operations on natural numbers, and to all intents and purposes basic arithmetic is all that is needed.

Consequently, formal systems (comprising syntax and rules) are especially powerful if they allow for the representation of numbers and the implementation of arithmetic. One way to see this precisely is to look at the consequences of encoding parts of the system within itself. This can typically done because the a formal system is just a collection of strings and strings can be encoded as numbers.

This was the approach of Gödel for obtaining his famous incompleteness results about the theory of arithmetic in the early 1930s. In the theory of arithmetic, the central concept is provability. By encoding the formulas and proof system of arithmetic within the theory of arithmetic he was able to derive the conclusion that there are some formulas which, although true, are not provable. In the λ-calculus the concept we are interested in is computability (stated precisely as λ-definability), and the same technique will allow us to conclude that there are some functions that are not computable.

The technique of encoding syntax by numbers takes Gödel's name. Here we are interested in encoding λ-terms.

Definition 9.7. *A* ***λ-Gödel numbering*** *is an injection $\# : \Lambda \to \mathbb{N}$, such that:*

- *There is a computable function* $\mathsf{app} : \mathbb{N}^2 \to \mathbb{N}$ *satisfying:*

$$\mathsf{app}(\#M, \#N) = \#(MN)$$

[1] Actually, the undecidability of the entschidungproblem followed from the undecidability of normalisation for λ-terms, which he proved first.

- *There is a computable function* $\mathsf{gnum} : \mathbb{N} \to \mathbb{N}$ *satisfying:*

$$\mathsf{gnum}(n) = \#(\ulcorner n \urcorner)$$

- *There is a computable function* $\mathsf{term?} : \mathbb{N} \to \mathbb{N}$ *satisfying:*

$$\mathsf{term?}(n) = \begin{cases} 1 & \text{if } \exists M \in \Lambda.\, \#M = n \\ 0 & \text{otherwise} \end{cases}$$

Sometimes it is required that a Gödel encoding be a bijection, in which case the final function is not necessary because it is constantly true. Also, you might reasonably expect that the definition would require certain other operations be computable, like an operation called abs corresponding to the syntactic operation of forming an abstraction; but we won't need them for what we are going to be doing.

There are actually lots of injections $\#$ that we could define in order to satisfy the required properties and, for the rest of our development, it doesn't matter which one we choose. However, it is worth seeing one of them to make sure we all believe that it is possible. Let's consider an adaptation of Gödel's original.

We start by making an arbitrary assignment code of natural numbers to the letters that make up the strings in the language Λ. For example maybe we decide to assign $\mathsf{code}(\text{`}\lambda\text{'}) = 0$, $\mathsf{code}(\text{`(')} = 1$, $\mathsf{code}(\text{`)'}) = 2$, $\mathsf{code}(\text{`.'}) = 3$, and then enumerate the countably many variables as $\mathsf{code}(\text{`}x_1\text{'}) = 4$, $\mathsf{code}(\text{`}x_2\text{'}) = 5$, $\mathsf{code}(\text{`}x_3\text{'}) = 6$ and so on. The key is then to map a whole string to a single number in such a way that we ensure that distinct strings get mapped to distinct numbers. The way Gödel did this was to use products of primes, with each prime raised to the power corresponding to the code of the letter:

$$\#(\ell_1, \ell_2, \ell_3, \ldots, \ell_k) = 2^{\mathsf{code}(\ell_1)} \cdot 3^{\mathsf{code}(\ell_2)} \cdot 5^{\mathsf{code}(\ell_3)} \cdot \ldots \cdot p_k^{\mathsf{code}(\ell_k)}$$

where p_i is the ith prime number. It then follows from the fundamental theorem of arithmetic that different strings will give rise to different Gödel numbers. I won't do it, so I hope you can believe that it is possible to compute the functions app and gnum according to this numbering.

Since we can represent numbers in the λ-calculus by Church numerals, let us investigate what we can say about λ-terms computing with codings of λ-terms. Our main aim is to be able to deduce some undecidability results about problems to do with λ-calculus. For example, we will show that the problem of determining whether a λ-term M terminates is undecidable.

To that end, we first need to make precise what we mean by property of terms and what it means for such a property to be decidable. In mathematics, a property of a class of objects is often just represented as a the set of those objects which have the property. For example, the property of being even is just the set of even numbers. Then we can say that an object satisfies a property just if it is a member of the set. We will say that a property is decidable just when membership in the set is decidable.

Definition 9.8 (Decidability)**.** *Let $\Phi \subseteq \Lambda$ be a property of λ-terms. We say that Φ is **decidable** just if the characteristic function of Φ is computable, i.e. there is a total computable function $\chi : \mathbb{N} \to \mathbb{N}$ satisfying, for all terms $M \in \Lambda$:*

$$\chi(\#M) = \begin{cases} 1 & \text{if } M \in \Phi \\ 0 & \text{if } M \notin \Phi \end{cases}$$

Since app and gnum are computable, it follows from the Church's thesis that there are λ-terms **app** and **gnum** to define them in the λ-calculus, so that:

$$\mathbf{app} \ulcorner \#M \urcorner \ulcorner \#N \urcorner =_\beta \ulcorner \#(NM) \urcorner \qquad \text{and} \qquad \mathbf{gnum} \ulcorner n \urcorner =_\beta \ulcorner \#\ulcorner n \urcorner \urcorner$$

It is worth considering the two expressions M and $\ulcorner \#M \urcorner$. *Both of these expressions are λ-terms*. The best way to think about the situation is that M is a program, $\#M$ is its source code and $\ulcorner \#M \urcorner$ is it's source code represented as a datatype that we can compute with in the λ-calculus (i.e. a function). Therefore, we can understand the combinator **app** as being the program that takes the source code of M and the source code of N and returns the source code for MN. If we specialise the equation for gnum so that n is, in fact, $\#M$, the code for some program, then we can understand gnum as being the program that takes the source code for some program M (represented as a Church numeral) and returns the source code for that numeral.

The following result is quite cute. It says that there is a λ-term that computes its own source code.

Lemma 9.1. *There is a term M satisfying $M =_\beta \ulcorner \#M \urcorner$.*

Proof. To see this, first define N as $\lambda x.\mathbf{app}\ x\ (\mathbf{gnum}\ x)$. Then set M to be $N \ulcorner \#N \urcorner$. It follows that:

$$\begin{array}{rcl} N \ulcorner \#N \urcorner & =_\beta & \mathbf{app} \ulcorner \#N \urcorner (\mathbf{gnum} \ulcorner \#N \urcorner) \\ & =_\beta & \mathbf{app} \ulcorner \#N \urcorner (\ulcorner \#\ulcorner N \urcorner \urcorner) \\ & =_\beta & \ulcorner \#(N \ulcorner \#N \urcorner) \urcorner \end{array}$$

□

Its construction is a variation of the combinator $\Omega = (\lambda x.xx)(\lambda x.xx)$, in which we use $\lambda x.\mathbf{app}\ x(\mathbf{gnum}\ x)$ rather than $\lambda x.xx$. Where $(\lambda x.xx)(\lambda x.xx)$ reduces to itself, $(\lambda x.\mathbf{app}\ x(\mathbf{gnum}\ x))(\lambda x.\mathbf{app}\ x(\mathbf{gnum}\ x))$ reduces to its own code.

Recall that a different variation on Ω, this time using $\lambda x.f(x\ x)$ gave us a fixed point combinator: $(\lambda x.f(x\ x))(\lambda x.f(x\ x))$ is a fixed point of f. This was the essence of the First Recursion Theorem (**Y** is just this term with f abstracted out as an input). If we play the coding trick with this term, we can deduce a similarly important result, which is due to Kleene.

Theorem 9.2 (Second Recursion Theorem)**.** *For all terms F, there exists a term M such that:*

$$F \ulcorner \#M \urcorner =_\beta M$$

Proof. Let F be an arbitrary term. Then define N as $\lambda x.F\,(\mathbf{app}\ x\ (\mathbf{gnum}\ x))$. Finally, take $N\ulcorner\#N\urcorner$ to be the witness M: it is easily seen that this term satisfies $F\ulcorner\#M\urcorner =_\beta M$. □

First, pay special attention to the quantification here: it says that *for all* terms *there exists* an input which behaves in a certain way. A common mistake is to look at the defining equation and assume that the theorem is stating the existence of an self-interpreter F for terms. There is such a theorem, but this is not it. It's a bit more difficult to attach an intuitive meaning to the theorem, except that it is stating the existence of fixed points *modulo coding*.

The theorem is valuable because it says that, if you construct a program that is supposed to compute with the codes of λ-terms, then no matter how you implement it (for all F), there will always be some input $\ulcorner\#M\urcorner$ about which you have no control over the corresponding output — it is always equal to M, whether that is what you want or not!

If you want to show that some set Φ is decidable then you must exhibit an F computing the characteristic function of Φ, that is it must be equal to $\ulcorner 0\urcorner$ or $\ulcorner 1\urcorner$ depending on whether the input is the code of a term M in Φ or not. What is the chance that you can construct an F with this property given that there will always be some input $\ulcorner\#M\urcorner$ over which you no control about the corresponding output? I'll tell you, it's not good.

In fact, for any property worth its salt, we can construct a rather tricksy term that precisely demonstrates the folly. When we say "any property worth its salt" what we mean is that the property should be a *non-trivial* and *behavioural*. Trivial properties are properties which are satisfied by no terms or by all terms. These kinds of properties are too easy to decide, because the characteristic function is constantly 0 or constantly 1 and these are obviously computable. Non-behavioural properties concern only the syntax of terms and not *what they compute*. For example, the property $\{M \in \Lambda \mid M \text{ is an abstraction }\}$ is not behavioural because there are terms like **IK** which do not satisfy the property and yet are convertible with a term **K** that does. Non-behavioural properties are easy to compute because we just need to interrogate the syntax. If a property is both non-trivial and behavioural then it is undecidable.

Theorem 9.3 (Scott-Curry)**.** *Let $\Phi \subseteq \Lambda$ satisfy the following properties:*

- *Φ is **non-trivial**: $\emptyset \neq \Phi \neq \Lambda$*
- *Φ is **behavioural**: if $M \in \Phi$ and $M =_\beta N$ then $N \in \Phi$*

Then it follows that Φ is undecidable.

Proof. Since Φ is non-trivial, there is some $P \in \Phi$ and some $Q \notin \Phi$. Suppose, for the purposes of obtaining a contradiction, that Φ is decidable. Then it follows that

there is some term F that computes its characteristic function, i.e. satisfying:

$$F \ulcorner \#M \urcorner =_\beta \begin{cases} \ulcorner 1 \urcorner & \text{if } M \in \Phi \\ \ulcorner 0 \urcorner & \text{if } M \notin \Phi \end{cases}$$

Define $G := \lambda x.\textbf{ifzero}\ (F\ x)\ P\ Q$. It follows from the Second Recursion Theorem that there is some term N such that $G \ulcorner \#N \urcorner = N$. One of $N \in \Phi$ or $N \notin \Phi$ is certainly true, so let us examine the two cases:

- If $N \in \Phi$, then by the properties of N, F and **ifzero**:

  $$N =_\beta G \ulcorner \#N \urcorner =_\beta \textbf{ifzero}\ (F \ulcorner \#N \urcorner)\ P\ Q =_\beta P$$

  but we assumed that $N \in \Phi$ and Φ is closed under $=_\beta$, so we are forced to conclude that $P \in \Phi$ which is a contradiction.

- If $N \notin \Phi$, then by the properties of N, F and **ifzero**:

  $$N =_\beta G \ulcorner \#N \urcorner =_\beta \textbf{ifzero}\ (F \ulcorner \#N \urcorner)\ P\ Q =_\beta Q$$

  but we assumed that $N \notin \Phi$ and Φ is closed under $=_\beta$, so we are forced to conclude that $Q \notin \Phi$ which is a contradiction.

□

The theorem is a direct analogue of Rice's Theorem for properties of Turing machines. Whether or not you have come across it, I recommend you look it up and compare its statement to the Scott-Curry Theorem. So which reasonable properties of λ-terms are decidable? Essentially none of them.

Corollary 9.1. *The problem* $\Phi_{norm} := \{M \in \Lambda \mid M \text{ is normalising}\}$ *is undecidable.*

Proof. The set Φ_{norm} is non-trivial because $\mathbf{I}$ is a member but $\mathbf{\Omega}$ is not. Moreover, it is closed under $=_\beta$. To see this let M be normalising and $M =_\beta N$. Since M is normalising, there is a normal form P with $M =_\beta P$. Therefore, also $N =_\beta P$ and it follows from the Church-Rosser theorem that, therefore $N \twoheadrightarrow_\beta P$. Consequently, N is normalising. Hence, undecidability of Φ_{norm} follows from the Scott-Curry theorem. □

9.3 Turing Machines

You may wonder, then, where Turing fits into the picture. No-one disputed the technical content of Church's paper: *if* you believe Church's thesis, *then* it followed that the decision problem is undecidable. But it was not clear to many people that Church's thesis was really believable.

At some point someone told you, or perhaps you read somewhere, that any function that can be computed can be computed by a Turing machine. Why did

you believe them? Probably because by the time you learned this you already had an excellent grasp of the intuitive notion of computation, as a result of extensive experience with digital computers, programming languages and so on. The mathematicians of the 1930s had none of that.

The only evidence for the thesis was that, firstly, all the computable number functions that one could think of had been shown by Kleene to be lambda definable and, secondly, that this other notion called total recursiveness has been shown to define the same class.

This is where Turing makes his mark. He had been working, independently of Church, Kleene and Gödel, on a precise characterisation of effective computability during his masters degree in Cambridge. He was just about to submit his paper for publication, also in 1936, when he read Church and Kleene's work on the lambda calculus. Consequently, he delayed slightly and wrote up a proof, which is in an appendix to that paper, showing that lambda-definability and Turing-computability are also equivalent.

> In a recent paper Alonzo Church has introduced an idea of "effective calculability", which is equivalent to my "computability", but is very differently defined. Church also reaches similar conclusions about the Entscheidungsproblem. The proof of equivalence between "computability" and "effective calculability" is outlined in an appendix to the present paper.

Turing showed that the class of functions definable in Church's λ-calculus exactly the same class as those definable by his machines: thus the thesis that this single class is representative of all algorithms takes both of their names.

One of the outstanding contributions of Turing's paper was his analysis of *why* we should believe that every effective function is Turing computable. Where the lambda calculus evolved naturally out of a desire to formalise functions and was later repurposed to explain computability, Turing's machines were designed from first-principles to capture exactly that notion. I highly recommend to anyone who has not read this part of Turings 1936/37 paper to have a look at it.