COMS30040

Types & λ-calculus

Lecture Notes for 2020/21

Steven Ramsay
Department of Computer Science
University of Bristol

Contents

1. Introduction

This unit is about *type systems* and *the λ-calculus*. I am sure you will be familiar with type systems from your experience coding in typed programming languages like C and Haskell. Over the next 7 weeks we will develop a very precise understanding of how type systems work, their key properties, limitations and connections with other parts of computer science and mathematics.

Type systems only make sense if you have something you want to give types to, e.g. the expressions of a programming language or a logic. So, for our study, we will fix an extremely simple programming language: the λ-calculus.

There are fourteen core lectures and a further seven supplementary lectures. The core lectures cover the classical theory of pure λ-calculus and (prenex-polymorphic) type systems. The breakdown of the core lectures is as follows:

- Lectures 1 – 3: basics of the pure, untyped λ-calculus
- Lectures 4 – 5: programming with pure λ-terms
- Lecture 6: the structural induction proof principal
- Lectures 7 – 9: type systems basics
- Lectures 10 – 11: type inference
- Lectures 12 – 14: connection to mathematical logic

You will only be assessed on the content of the core lectures.

The supplementary lectures are, nevertheless, *essential*. They provide the connection of the pure theory to applications in programming languages, to the history of the subject and to your experience of logic and proof. This split keeps the core part pure and simple: all the key, intellectually challenging concepts are present, but the definitions are less complicated and cumbersome than if we analysed a complex programming language. If you can master the pure theory, using it to understand programming languages and logics is quite straightforward, and what you actually work with and are assessed on is much more pleasant.

λ-calculus

We start our study with the basics of the pure, untyped λ-calculus, so let me say a bit more about λ-calculus in general.

The λ-calculus holds an important position at the intersection of programming languages, the theory of computation and the foundations of mathematics because it codifies the notion of *function* in the purest sense.

This codification is achieved by providing a language — a syntax — for writing functions and a calculus — some rules — that say how functions work. It is pure in the sense that the language and the calculus don't make any assumptions about what kinds of objects your functions are manipulating: they could be truth valued functions (predicates) in the domain of logic, they could be linear transformations in algebra, they could be isomorphisms of groups or functions you defined in your Haskell program. The syntax and rules of the λ-calculus "work" in any of these domains and many others besides.

The only thing that the λ-calculus does suppose is that you use variables to describe objects in that domain. To the notion of *variable*, the λ-calculus adds another two syntactic forms: the *abstraction*, for introducing new functions, and the *application*, for applying them to inputs.

Abstraction. Suppose M is an expression denoting some object of interest. It doesn't matter what kind of object this expression is describing 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:

$$\lambda x.M$$

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 $f(x) = x^2-1$.
$\lambda x.\,\mathsf{reverse}\ x\ \mathsf{++}\ x$	The function that takes a string input x and appends it to its own reversal. Haskell programmers may write this as `\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.

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:

$$(\lambda x.M)N$$

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 $f(x)=x^2-1$.
$(\lambda x.\text{reverse } x +\!\!+ x)$ "d or even"	Supplying "d or even" as an input to the function $\lambda x.\text{reverse } x +\!\!+ x$.
$(\lambda x.\emptyset)\,(y \cap z)$	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:

$$(\lambda x.M)N =_\beta M[N/x]$$

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:

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

Each area of computer science and mathematics has its own special language and its own notation, but the development of functions described by the λ-calculus makes sense in all of them because all it pressuposes is that they represent abstract objects using variables.

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). 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 $(\lambda x.M)N =_\beta M[N/x]$.

1.1 Terms

I've been talking about the λ-calculus as a means to describe functions in some setting — number theory, Haskell, set theory, logic, probability — this is sometimes called *applied* λ-calculus. However, what we're actually going to study in the *core* part of this unit is the *pure* λ-calculus. This is what you get if you try to understand functions apart from any particular domain.

Variables. Variables are still going to be crucial, but in my previous examples the variables stood for objects in the domain: numbers, sets, Haskell values. What is the point of the variables if you want to study the pure calculus of functions set apart from any particular domain? In the pure calculus, the variables themselves stand for functions: there's nothing but functions!

It doesn't matter what the variables 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 the language of the pure (untyped) λ-calculus – its syntax – 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 1.1 (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:*

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

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.*

Except for the variable immediately following a λ*, any substring of a term* M *that is itself a term is called a* ***subterm****.*

You should think of Λ as the set containing all and only those strings that can be formed according to the given rules. Here are some examples of terms:

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

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.

Some examples of what we mean by subterms:

- The subterms of $((\lambda x.x)y)$ are: $((\lambda x.x)y)$, $(\lambda x.x)$, x and y.
- The subterms of $(\lambda x.y)$ are: $(\lambda x.y)$ and y.
- The subterms of $((x(\lambda y.y))z)$ are: $((x(\lambda y.y))z)$, $(x(\lambda y.y))$, x, $(\lambda y.y)$, y and z.
- The only subterm of x is x itself.

Membership Principle for Inductively Defined Sets

In this unit we will see many examples of "inductive" definitions, such as that of the set Λ of λ-terms, which are given using 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, the (App) rule says that whenever we know that M and N are terms, then also we can conclude that (MN) is a term. The rules define which strings constitute terms and which do not.

For example I can convince you that the string $((\lambda z.z)y)$ is a term (i.e. is a member of the set Λ) by the following reasoning. Rule (Var) says that I can conclude that z is term even when I don't know anything else and (Abs) says that, now that I know z is a term, I can conclude that $(\lambda z.z)$ is a term. Here I used the rule (Abs) with the *metavariable* M chosen to be the particular term z. Similarly, I can also conclude by (Var) that y is a term. Then, now that I know that $(\lambda z.z)$ is a term and y is a term, I can conclude by (App) that $((\lambda z.z)y)$ is a term. In this last step, I used (App) choosing M to be $(\lambda z.z)$ and N to be y.

This argument for the string $((\lambda z.z)y)$ being a term can be summed up very neatly by the following *proof tree*:

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

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:

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

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, we used the following instance of the rule (Abs):

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

with $M := z$ and $x := z$ 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. However, you *must* use the rule exactly as it is presented, if you write something like:

$$\frac{y \in \Lambda \quad (\lambda z.z) \in \Lambda}{((\lambda z.z)y) \in \Lambda}$$

This is not correct, and it will cause me or the TAs a headache because we won't be sure whether you meant to conclude $((\lambda z.z)y) \in \Lambda$, but mixed up the premises, or whether you meant to conclude, the equally valid, $(y(\lambda z.z)) \in \Lambda$ and mixed up your conclusion.

We say that membership in Λ is governed by the following *principle*:

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

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 string "(" is not a term. You can convince yourself of this informally by observing that in the conclusion of each rule (the bit below the line) any parentheses will be properly matched: for every left parenthesis, there will be a corresponding right parenthesis. However, we will see how to argue this more convincingly later.

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 above to be rendered unambiguously as:

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

Except if we refer to it explicitly, from now on we will write terms with the minimum number of parentheses, according to the conventions. There is one exception: the convention would have us

render a term like $(x(\lambda y.\, y))$ as $x\lambda y.\, y$. Whilst this is unambiguous, it is not easy to read and, in practice, I don't know of anyone who would write this term like that. Therefore, I ask you to always retain the parentheses around an abstraction, except when it constitutes the whole term, as in:

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

2. Alpha

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:

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

However, according to our definition, terms are just certain strings and these three are different strings (assuming x, y and z are different variables). Let's rectify this.

2.1 Capture avoiding substitution

Consider the term $\lambda x.\, yx$. Our intuition is that 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 in the output. 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 2.1. *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 $(\lambda x.(\lambda y.(yx))$ is closed, because:

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

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.

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. To remember that it is x that is being replaced by N, it is a good idea to read $[N/x]$ as "with N for x".

The idea is fine, but it leaves a question mark over certain edge cases to do with binding: does $(\lambda x.zx)[x/z]$ mean $(\lambda x.zx)$ or $(\lambda x.xx)$ or something else? A potential problem with $(\lambda x.zx)[x/z]$ is that the term that we are introducing in place of z, namely x, contains a free variable that is already bound in $\lambda x.zx$. 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".

If we really want to treat $\lambda x.zx$ and $\lambda y.zy$ as identical, as we stated earlier, then we cannot allow variable capture because we end up with inconsistent results depending on whether we substitute x for z in $\lambda x.zx$ or x for z in $\lambda y.zy$. If we replace z by x in the former it is captured and yields $\lambda x.xx$. If we replace z by x in the latter, we get $\lambda y.xy$. However, $\lambda x.xx$ and $\lambda y.xy$ do *not* only differ in the names of their bound variables — there's no way I can rename x in the former or y in the latter and end up with identical terms. So allowing capture would lead to a situation in which two terms that differ only in the names of their bound variables (and thus we think of as being equal) undergo the same substitution and become two terms that are completely different from each other.

The version of substitution that we will define is called "capture-avoiding substitution" because we will outlaw such possibilities by leave substitution undefined if a case like this would arise.

Definition 2.2. *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.

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!

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

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).

2.2 α-equivalence

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, we say they are α-equivalent.

Definition 2.3. *Suppose $\lambda x.P$ is a term and $y \notin \mathsf{FV}(P)$. Then the act of replacing $\lambda x.P$ by $\lambda y.P[y/x]$ is called a* ***change of bound variable name****. If two terms M and N can be made identical just by changing bound variable names to fresh names (not already used inside them), we say they are* ***α-equivalent****.*

Intuitively, the terms $\lambda x.xy$ and $\lambda z.zy$ are α-equivalent because, if I pick a fresh variable y' to use instead of x in the body of the first term and use the same y' instead of z in the second term, the terms become equal: $\lambda y'.y'y$.

Most of the time it will be obvious when two terms are α-equivalent, 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-equivalent 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.

Assumption: α-equivalent terms are identical

From this point on, we are going to build α-equivalence 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 2.4. *The set of λ-terms is the set Λ with all α-equivalent terms identified.*

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

This is the only part of the unit where I am going to push something under the carpet (so to speak), but if that makes you uncomfortable, you may wish to read Appendix **??**. In practice we are still going to treat λ-terms as strings, we are going to make definitions by recursion over their syntax and so on. However, there is a great advantage to identifying all terms that are α-equivalent, and that is that we are *free to rename bound variables whenever convenient* (i.e. change to a different, but α-equivalent string). If we are considering a term like $\lambda x.\, x$, and the bound variable being called x is inconvenient (perhaps because we already have an x hanging around somewhere else) then we can simply change it, no questions asked, to e.g. $\lambda y.\, y$. It is particularly helpful during a proof, because you may be given an abstraction $\lambda x.\, M$ that you know nothing about and, separately, a variable y, again which you know nothing about and the identification allows you to simply assume that the bound variable x and the other variable y are different. The idea is so useful that it has 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 any others.

You will appreciate this convention more fully once we have done some proofs where it is useful.

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. Since the strings are λ-terms, you get to use this definition 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-equivalent 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.

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 1.1 as the *raw terms* and the terms of Definition 2.4 as the *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[1] 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 write raw terms, but think λ-terms, just as we write fractions but think ratios.*

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 $\lambda x.P$ with x different from any other variable name that might currently be in scope.

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 strings but in which bound variable names are eliminated in favour of indexing the inputs from left to right.[2]

[1] Identifying here means "making identical".

[2] 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. Beta

We bring terms to life by describing how they compute and we consider what it means for two terms to be equal.

3.1 One-step β

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 3.1. *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:

$$(\lambda x.x)(\lambda y.y) \quad \to_\beta \quad \lambda y.y$$

$$(\lambda z.(\lambda x.yx)z) \quad \to_\beta \quad \lambda z.yz$$

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

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

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.

$$\lambda x.(\lambda z.x)y((\lambda z.z)x) \quad \not\to_\beta \quad \lambda x.(\lambda z.x)y((\lambda z.z)x)$$

$$\lambda x.(\lambda z.x)y((\lambda z.z)x) \quad \not\to_\beta \quad \lambda x.xx$$

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 as 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 3.2. *The **one-step β-reduction** relation, written infix as $\rightarrow_\beta$, is inductively defined by the following rules:*

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

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

*If $M \rightarrow_\beta N$ then we say that N is a **reduct** of M. A term M is said to be in **β-normal form** just if there is no term N for which $M \rightarrow_\beta N$.*

The one-step β-reduction relation $\rightarrow_\beta$ is another inductively defined set, it just happens to be a set of pairs of strings rather than a set of strings like the set Λ of λ-terms. Consequently, membership works in exactly the same way as for Λ, only this time when we argue that $(M, N) \in \rightarrow_\beta$ we write it infix as $M \rightarrow_\beta N$. We have the following principle:

> There is a one-step reduction $M \rightarrow_\beta N$ iff there is a proof tree built from the rules (Redex), (AppL), (AppR) and (Abs) above, with conclusion $M \rightarrow_\beta N$.

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. For example:

$$\textsf{(Abs)}\ \frac{\textsf{(Step)}\ \dfrac{}{(\lambda x.yx)z \rightarrow_\beta yz}}{(\lambda z.(\lambda x.yx)z) \rightarrow_\beta \lambda z.yz}$$

You may wish to do some more for practice, but I will not generally 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.

Reduction and Conversion

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:

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

Definition 3.3 (β-reduction)**.** *Whenever there is a possibly empty sequence of consecutive one-step reductions:*

$$M_0 \to_\beta M_1 \to_\beta \cdots \to_\beta M_{k-1} \to_\beta M_k$$

for $k \geq 0$, we say that M_0 β-reduces to M_k and write $M_0 \twoheadrightarrow_\beta M_k$. Note, we include the case that $k = 0$ and hence $M_0 = M_k$ (i.e. we include 0-step reductions).

Usually we will want to use $\twoheadrightarrow_\beta$ to abbreviate a reduction sequence, such as the sequence written above, by its starting and finishing terms:

$$(\lambda x.(\lambda z.x)yy)(\lambda x.xy) \twoheadrightarrow_\beta yy$$

However, there is no requirement that the right hand side of $\twoheadrightarrow_\beta$ be a normal form, $M \twoheadrightarrow_\beta N$ just means that M reduces to N using 0, 1 or many steps. By the above definition, it is also true that, for example:

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

Definition 3.4. *Some nomenclature:*

- *If $M \twoheadrightarrow_\beta N$ then we say that N is a **reduct** of M. If also $M \neq_\beta N$, we say that it is a **proper reduct**.*
- *A term M such that $M \twoheadrightarrow_\beta N$ for some normal form N is said to **have a normal form** or be **normalisable**.*
- *A term M for which there is no infinite reduction sequence $M \to_\beta M_1 \to_\beta M_2 \to_\beta \cdots$ is said to be **strongly normalisable**.*

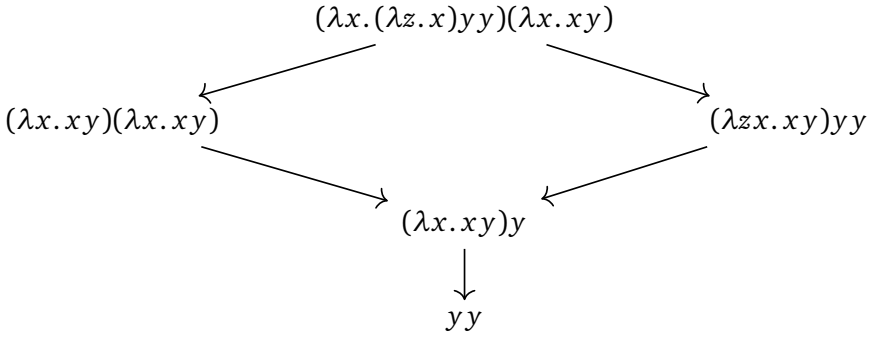
Now that we know how to compute with λ-terms, let's look at some interesting programs and their names. On the left are the combinators and how we shall abbreviate them. On the right an illustrative example of how they behave.

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

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

Confluence of β-reduction

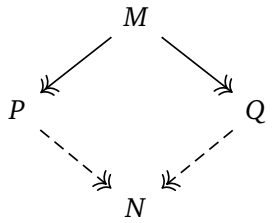
There is a sense in which one-step β-reduction is *non*deterministic: as we saw in the earlier examples, for certain terms there is more than one possible reduct. This can be pictured very nicely if we draw the reduction graph of a term. The reduction graph of a term P is just a rooted directed graph. 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 $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$:



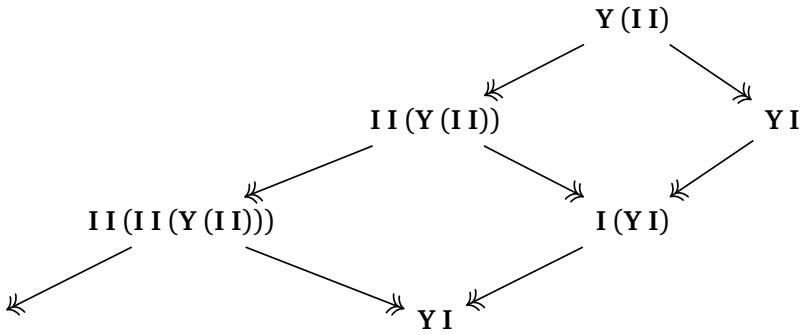
However, in this example, we can see that, although there are several possible ways to reduce $(\lambda x.(\lambda z.x)yy)(\lambda x.xy)$, they all eventually lead to yy. This is actually an important property of β-reduction, called *confluence*. We will state the theorem now and you will be expected to make use it from now on (i.e. you may assume it), but we will defer the proof until later.

Theorem 3.1 (Confluence of β)**.** *If $M \twoheadrightarrow_\beta P$ and $M \twoheadrightarrow_\beta Q$ then there exists a term N (not necessarily a normal form) such that $P \twoheadrightarrow_\beta N$ and $Q \twoheadrightarrow_\beta N$.*

This can be nicely illustrated by the following diagram, in which a solid line is a given and a dashed line is what maybe concluded (but N itself should really be dashed):



It is a bit more nuanced than simply "if there are two different ways to reduce a term then they will always reach the same normal form" because it applies even when the term in question fails to have a normal form. Consider e.g. $\mathbf{Y}\,(\mathbf{I}\,\mathbf{I})$. An illustration:



Use two fingers to trace out any two paths on this reduction graph for a while. Wherever each of your fingers got to, there is guaranteed to be a way to get to a common reduct.

β-Conversion

When you learned about functional programming, you probably did some short proofs using equational reasoning, like:

$$\begin{aligned}\mathsf{map}\ (+1)\ [3,8,6] &= 4 : \mathsf{map}\ (+1)\ [8,6] \\ &= 4 : 9 : \mathsf{map}\ (+1)\ [6] \\ &= 4 : 9 : 7 : \mathsf{map}\ (+1)\ [\,] \\ &= 4 : 9 : 7 : [\,] \\ &= [4,9,7]\end{aligned}$$

What do we really mean by $\mathsf{map}\ (+1)\ [3,8,6] = 4 : \mathsf{map}\ (+1)\ [8,6]$? When I am doing arithmetic, I know what I mean by $3+1=4$, I mean that the left and right sides of the $=$ are literally the same natural number, they are identical: $3+1$ is a number and it is the very same number has 4. However, that's not what I mean in $\mathsf{map}\ (+1)\ [3,8,6] = 4 : \mathsf{map}\ (+1)\ [8,6]$: it's plain to see that the left and right sides of this equation, which are Haskell expressions (i.e. strings), are *not* identical: the string on the left starts with 'm' whereas the one on the right starts with '4'.

You might complain at that and say that we are not talking about Haskell expressions as strings, but rather Haskell expressions as (compiled) programs, and that the left and right hand sides are identical programs. However, that is clearly untrue also, since the program on the left has more steps to evaluate than the one on the right before reaching a normal form.

What we need is some kind of "equality modulo computation".

We could perhaps make sense of these equations by justifying them by saying $\mathsf{map}\ (+1)\ [3,8,6] = 4 : \mathsf{map}\ (+1)\ [8,6]$ because the left-hand side *evaluates to* the right-hand side. However, this too seems like a dishonest way to interpret an equation, because if

$$\mathsf{map}\ (+1)\ [3,8,6] = 4 : \mathsf{map}\ (+1)\ [8,6]$$

then it should also be that

$$4 : \mathsf{map}\ (+1)\ [8,6] = \mathsf{map}\ (+1)\ [3,8,6]$$

but surely it is not the case that $4 : \mathsf{map}\ (+1)\ [8,6]$ evaluates to $\mathsf{map}\ (+1)\ [3,8,6]$.

A more satisfying approach is to say that two expressions are equal modulo computation when they *each evaluate to the same thing.* We make this precise in λ-calculus by the notion of β*-convertibility*.

Definition 3.5 (β-convertibility)**.** *Let M and N be terms. If M and N have a common reduct P, i.e. there is a term P such that $M \twoheadrightarrow_\beta P$ and $N \twoheadrightarrow_\beta P$, then we say that M and N are β-convertible and write $M =_\beta N$.*

This formulation has the advantage that we can say that $\mathbf{K}\ \Omega\ \mathbf{I}$ and $\mathbf{I}\ \Omega$ are the same modulo computation, i.e. β-convertible, even though neither has a normal form. Here are some other examples:

$$
\begin{array}{rll}
(\lambda xy.\,yx)\,z\,\mathbf{I} =_\beta z & \text{because} & (\lambda xy.\,yx)\,z\,\mathbf{I} \twoheadrightarrow_\beta z \twoheadleftarrow_\beta z \\
z =_\beta (\lambda xy.\,yx)\,z\,\mathbf{I} & \text{because} & z \twoheadrightarrow_\beta z \twoheadleftarrow_\beta (\lambda xy.\,yx)\,z\,\mathbf{I} \\
\mathbf{K}\Omega\mathbf{I} =_\beta \Omega & \text{because} & \mathbf{K}\Omega\mathbf{I} \twoheadrightarrow_\beta \Omega \twoheadleftarrow_\beta \Omega \\
\mathbf{I}\Omega =_\beta \mathbf{K}\Omega\mathbf{I} & \text{because} & \mathbf{I}\Omega \twoheadrightarrow_\beta \Omega \twoheadleftarrow_\beta \mathbf{K}\Omega\mathbf{I} \\
\mathbf{K}\Omega\mathbf{I} =_\beta \mathbf{SKK}\Omega & \text{because} & \mathbf{K}\Omega\mathbf{I} \twoheadrightarrow_\beta \Omega \twoheadleftarrow_\beta \mathbf{SKK}\Omega
\end{array}
$$

4. Definability

In this lecture, I want to talk about data and how to implement datatypes when all you have available is functions.

To do that, we have to ask ourselves: what does it mean to implement a datatype, e.g. what does it mean to implement a type of natural numbers, or a type of lists in a given programming language (or logic)? Suppose I give you some collection of expressions in that programming language and told you that they are my implementation of natural numbers, how could you verify that they work as natural numbers?

For example, suppose I told you that I have implemented natural numbers in Haskell using successively longer versions of the string "hello", "helloo", "hellloo", "heellloo", "hheellloo", "hheelllooo"... does this work?

Suprisingly, there is no canonical answer to this question. However, if you ask a slightly different question, the situation is much clearer. If I gave you a collection of expressions and told you that they are an implementation of natural numbers *and* I also gave you some other expressions that I claimed implemented some standard operations on natural numbers: addition, subtraction etc. *Then* you could verify that the implementation of the operations work correctly with respect to the implementation of the natural numbers.

The next question is: how do I verify that an implementation of an operation "works correctly" with respect to an implementation of natural numbers?

Suppose we are given a Haskell function f and a collection of Haskell expressions e_0, e_1, e_2, ... that are meant to represent the natural numbers 0, 1, 2, ... (i.e. e_i is supposed to represent natural number i) and suppose I claim that f is an implementation of addition. One way to verify the claim is to check that f satisfies the following equation, for all n and m:

$$f\ e_n\ e_m \quad = \quad e_{n+m}$$

If we can show that f and this number representation scheme e_0, e_1, e_2 etc. work together correctly, in the precise sense of satisfying this equation, then we will know that it works. If it's clear how to get from n to it's representation e_n and back again, then we can use f to compute addition for us: we just take the two numbers n and m we want to add, construct their representations e_n and e_m, apply f and decode the output.

This idea that operations are *specified* using equations (or other logical descriptions) is very important in the following two lectures.

4.1 Church's Numeral System

In Lecture 3, 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, clay tokens were used as numerals (at least, according to Wikipedia). In Church's numeral system, the numerals are certain λ-terms.

Definition 4.1. *The **Church numeral** for the number n, abbreviated $\ulcorner n \urcorner$, is:*

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

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:

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

An equivalent way to define the numerals is to ask that they satisfy the following two recursive β-equations:

$$\begin{array}{rcl}
\ulcorner 0 \urcorner & =_\beta & \lambda f\, x.\, x \\
\ulcorner k+1 \urcorner & =_\beta & \lambda f\, x.\, f\ (\ulcorner k \urcorner\ f\ x)
\end{array}$$

This way we can calculate $\ulcorner 3 \urcorner$ by equational reasoning:

$$\begin{array}{rl}
 & \ulcorner 1+1+1+0 \urcorner \\
=_\beta & \lambda f\, x.\, f\ (\ulcorner 1+1+0 \urcorner\ f\ x) \\
=_\beta & \lambda f\, x.\, f\ ((\lambda f' x'.\, f'\ (\ulcorner 1+0 \urcorner\ f'\ x'))\ f\ x) \\
=_\beta & \lambda f\, x.\, f\ (f\ (\ulcorner 1+0 \urcorner\ f\ x)) \\
=_\beta & \lambda f\, x.\, f\ (f\ ((\lambda f' x'.\, f'\ (\ulcorner 0 \urcorner\ f'\ x'))\ f\ x)) \\
=_\beta & \lambda f\, x.\, f\ (f\ (f\ (\ulcorner 0 \urcorner\ f\ x))) \\
=_\beta & \lambda f\, x.\, f\ (f\ (f\ ((\lambda f' x'.\, x')\ f\ x))) \\
=_\beta & \lambda f\, x.\, f\ (f\ (f\ x))
\end{array}$$

You might ask whether this is a good way to represent numbers in the λ-calculus? 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 a point in its favour is that every number can be represented as a numeral 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 pure 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 and each of them allows us to implement operations on natural numbers that work correctly. In fact, we will go on to show that all computable operations on natural numbers (e.g. all operations on natural numbers that a Turing machine can compute, or a Haskell program can compute) can be implemented using λ-terms working on Church numerals. Let's make the notion of "can be implemented using λ-terms" precise.

Definition 4.2 (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:*

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

Addition is λ-definable. The λ-term $\lambda yz.\lambda fx.yf(zfx)$ 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.

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

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 exponentiation 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.

$$\mathbf{Pred} := \lambda z.\lambda fx.z(\lambda gh.h(gf))(\lambda u.x)(\lambda u.u) \qquad \begin{array}{l} \mathbf{Pred} \ulcorner 0 \urcorner =_\beta \ulcorner 0 \urcorner \\ \mathbf{Pred} \ulcorner n + 1 \urcorner =_\beta \ulcorner n \urcorner \end{array}$$

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 this λ-term works (although you can prove it satisfies the given equations). 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 $\mathbf{Pred}\ulcorner 0\urcorner =_\beta \ulcorner 0\urcorner$ and $\mathbf{Pred}\ulcorner n+1\urcorner =_\beta \ulcorner n\urcorner$. 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.

Let's put it to use in λ-defining "truncated" subtraction — subtraction on natural numbers where $m-n=0$ whenever $m \leq n$. We can define subtraction as iterated predecessor, to compute $m-n$ we just want to perform the predecessor operation n-times on m.

$$\mathbf{Sub} \coloneqq \lambda mn.\, n\ \mathbf{Pred}\ m \qquad \begin{array}{ll} \mathbf{Sub}\ulcorner m\urcorner\ulcorner n\urcorner =_\beta \ulcorner 0\urcorner & \text{if } m \leq n \\ \mathbf{Sub}\ulcorner m\urcorner\ulcorner n\urcorner =_\beta \ulcorner m-n\urcorner & \text{otherwise} \end{array}$$

We can prove it has the required properties by induction on n, but I'll just give you a demonstration, observe how we avoid the messy definition of **Pred** and instead just use its key properties:

$$\begin{array}{rcl} \mathbf{Sub}\ulcorner 5\urcorner\ulcorner 3\urcorner & =_\beta & \ulcorner 3\urcorner\ \mathbf{Pred}\ulcorner 5\urcorner \\ & = & (\lambda fz.f(f(f\ z)))\ \mathbf{Pred}\ulcorner 5\urcorner \\ & =_\beta & \mathbf{Pred}\ (\mathbf{Pred}\ (\mathbf{Pred}\ulcorner 5\urcorner)) \\ & =_\beta & \mathbf{Pred}\ (\mathbf{Pred}\ulcorner 4\urcorner) \\ & =_\beta & \mathbf{Pred}\ulcorner 3\urcorner \\ & =_\beta & \ulcorner 2\urcorner \end{array}$$

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 $\lambda xyz.\, x(\mathbf{K}z)y$. 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.

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

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

4.2 Church Lists

A second example is lists, encoded according to the same principles. The idea is to represent a lists of numbers as follows:

$$\begin{array}{rcl} \ulcorner[]\urcorner & = & \lambda cn.\, n \\ \ulcorner n_1 : []\urcorner & = & \lambda cn.\, c\ \ulcorner n_1\urcorner\ n \\ \ulcorner n_1 : n_2 : []\urcorner & = & \lambda cn.\, c\ \ulcorner n_1\urcorner\ (c\ \ulcorner n_2\urcorner\ n) \\ \ulcorner n_1 : n_2 : n_3 : []\urcorner & = & \lambda cn.\, c\ \ulcorner n_1\urcorner\ (c\ \ulcorner n_2\urcorner\ (c\ \ulcorner n_3\urcorner\ n)) \\ & \vdots & \end{array}$$

So, in general we can define the encoding of an arbitrary list by the following recursive definition.

Definition 4.3 (Church Lists)**.** *The **Church encoding** of a list xs is the term $\ulcorner xs \urcorner$ defined recursively by:*

$$\begin{array}{rcl} \ulcorner [] \urcorner & = & \lambda cn.\, n \\ \ulcorner n : ns \urcorner & = & \lambda cn.\, c\, \ulcorner n \urcorner\, (\ulcorner ns \urcorner\, c\, n) \end{array}$$

For example we can calculate $\ulcorner n_1 : n_2 : [] \urcorner$ above by using this definition:

$$\begin{array}{rl} & \ulcorner n_1 : n_2 : [] \urcorner \\ = & \lambda cn.\, c\, \ulcorner n_1 \urcorner\, (\ulcorner n_2 : [] \urcorner\, c\, n) \\ = & \lambda cn.\, c\, \ulcorner n_1 \urcorner\, ((\lambda c' n'.\, c'\, \ulcorner n_2 \urcorner\, n')\, c\, n) \\ = & \lambda cn.\, c\, \ulcorner n_1 \urcorner\, (c\, \ulcorner n_2 \urcorner\, n) \end{array}$$

To achieve this, we define the empty list and cons by:

$$\mathsf{Nil} := \lambda cn.\, n \qquad \mathsf{Cons} := \lambda xy.\, \lambda cn.\, c\, x\, (y\, c\, n)$$

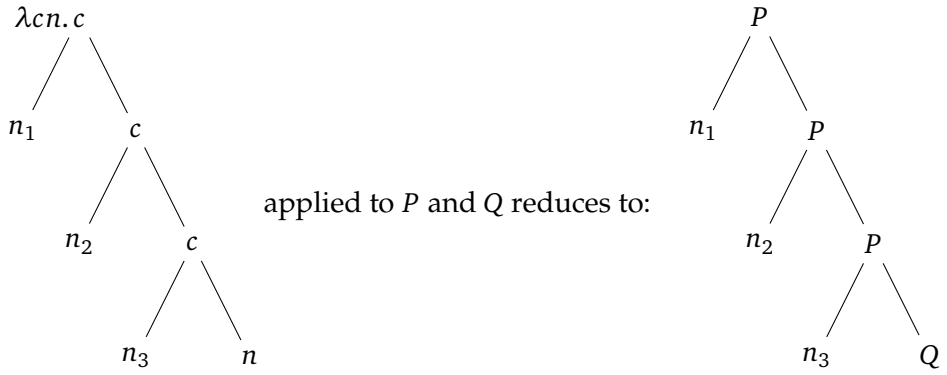
Let's check it works for our example, we should be able to build a representation of $[\ulcorner 1 \urcorner, \ulcorner 2 \urcorner, \ulcorner 3 \urcorner]$:

$$\begin{array}{rl} & \mathsf{Cons}\, \ulcorner 1 \urcorner\, (\mathsf{Cons}\, \ulcorner 2 \urcorner\, (\mathsf{Cons}\, \ulcorner 3 \urcorner\, \mathsf{Nil}) \\ =_\beta & \mathsf{Cons}\, \ulcorner 1 \urcorner\, (\mathsf{Cons}\, \ulcorner 2 \urcorner\, (\mathsf{Cons}\, \ulcorner 3 \urcorner\, (\lambda c' n'.\, n')) \\ =_\beta & \mathsf{Cons}\, \ulcorner 1 \urcorner\, (\mathsf{Cons}\, \ulcorner 2 \urcorner\, (\lambda cn.\, c\, \ulcorner 3 \urcorner\, ((\lambda c' n'.\, n')\, c\, n))) \\ =_\beta & \mathsf{Cons}\, \ulcorner 1 \urcorner\, (\mathsf{Cons}\, \ulcorner 2 \urcorner\, (\lambda cn.\, c\, \ulcorner 3 \urcorner\, n)) \\ =_\beta & \mathsf{Cons}\, \ulcorner 1 \urcorner\, (\lambda c' n'.\, c'\, \ulcorner 2 \urcorner\, ((\lambda cn.\, c\, \ulcorner 3 \urcorner\, n)\, c'\, n') \\ =_\beta & \mathsf{Cons}\, \ulcorner 1 \urcorner\, (\lambda c' n'.\, c'\, \ulcorner 2 \urcorner\, (c'\, \ulcorner 3 \urcorner\, n') \\ =_\beta & \lambda cn.\, c\, \ulcorner 1 \urcorner\, ((\lambda c' n'.\, c'\, \ulcorner 2 \urcorner\, (c'\, \ulcorner 3 \urcorner\, n')\, c\, n) \\ =_\beta & \lambda cn.\, c\, \ulcorner 1 \urcorner\, (c\, \ulcorner 2 \urcorner\, (c\, \ulcorner 3 \urcorner\, n)) \end{array}$$

Indeed, the following can be easily checked by evaluation:

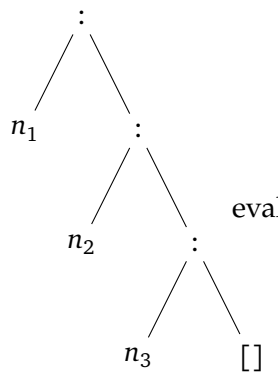
$$\mathsf{Cons}\, \ulcorner n \urcorner \ulcorner xs \urcorner =_\beta \ulcorner n : xs \urcorner$$

If you think about the representation of a Church list, it is again a kind of iterator of functions, only now the first function takes two arguments. Let's draw out what happens when you apply a Church list considering the tree structure of the terms involved:

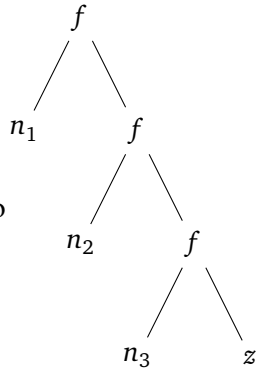


I hope it will remind you of the following picture, which you may have seen when you studied functional programming:

foldr f z applied to



evaluates to



In words, the Haskell expression foldr f z will replace every occurrence of cons in its input by f and every occurrence of nil by z, keeping the other structure the same: foldr f z is a *list homomorphism*. This is just what our Church encoded list does when applied to two terms P and Q, in other words, we can think of a Church encoding of a list xs as the foldr function for xs, i.e.

think of $\ulcorner xs \urcorner$ as the Haskell expression $\backslash f\ z\ \rightarrow$ foldr f z xs

This analogy gives us an easy way to see how to implement the summation of a list of numbers, which is a typical example of a fold:

$$\mathsf{Sum} \coloneqq \lambda xs.\, xs\ \mathbf{Add}\ \ulcorner 0 \urcorner$$

5. 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:

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

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?

5.1 Recursive functions

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:

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

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: 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 5.1. *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 5.1 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.

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.

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:*

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

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 $(\lambda yz.z(yyz))(\lambda yz.z(yyz))$. 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? To show that it has *no* solution, we assume it has one, so suppose M is a solution. Then we have $(\lambda x.\lambda y.y)M =_\beta (\lambda x.\lambda yz.y)M$ is true of this M. By equational reasoning:

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

Since the term on the left and the term on the right are convertible, by definition there must be a common a reduct Q. But these two terms are not identical, so Q must be a proper reduct of at least one of them. However, they are both in normal form: a contradiction.

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

Anyway, 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.

5.2 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 5.2. *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 later. 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 5.1 (First Recursion Theorem)**.** *Every λ-term posesses a fixed point.*

Proof. Let M be an arbitrary λ-term and let $\mathbf{Y}$ be the following "Y-combinator":

$$\lambda f.(\lambda x.f(xx))(\lambda x.f(xx))$$

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.

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 $x_1,\ldots,x_n$. 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.

6. Induction

In the unit we will see many examples of "inductive" definitions, such as that of the set Λ of λ-terms. Each of them comes equipped with two principles for reasoning. We have already seen the membership principle that we have already been using to argue that certain items are in the set by constructing a proof tree. The second principle is the induction principle which we can use to argue that all elements of the inductively defined set satisfy some property.

6.1 The Induction Principle for Λ

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

> 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 P and Q, if Φ holds of P and Φ holds of Q then Φ holds of PQ.
>
> (LI3) For all terms P and variables x, if Φ holds of P then Φ holds of $\lambda x.P$.
>
> Then it follows that Φ holds of all λ-terms.

The conditions "Φ holds of P and Φ holds of Q" in the antecedent of (LI2) and "Φ holds of P" 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).

For example we can prove the following statement by induction on λ-terms.

Lemma 6.1. *For all λ-terms M, for all λ-terms N: if $x \notin \mathsf{FV}(M)$ then $\mathsf{FV}(M[N/x]) = \mathsf{FV}(M)$*

A commentary on how to construct the proof.

A crucial point is that we can use induction because the statement has the form "for all λ-terms M, Φ holds of M". If would like to prove something by induction, then it is essential that it has this form or, more likely, that you can put it into this form, in order to identify exactly what Φ is in (LI1), (LI2) and (LI3). In this case, the form of the statement is correct already and we can immediately see that Φ is the statement "for all λ-terms N: if $x \notin \mathsf{FV}(M)$ then $\mathsf{FV}(M[N/x]) = \mathsf{FV}(M)$". Therefore, we can apply induction. According to the induction principle, we need to prove three things separately.

- The first is when we take M to be an arbirary variable, let's say y because we have already used x. We have to prove "for all λ-terms N: if $y \notin \mathsf{FV}(x)$ then $\mathsf{FV}(y[N/x]) = \mathsf{FV}(y)$". The statement to be proven is also of the form "for all λ-terms..." so induction is one possibility, but I can tell you that we will not have to use induction this time and typically (in this unit) it is not necessary to nest inductions. Instead we proceed with the standard approach to proving anything of the form "for all X...", i.e. we consider an arbitrary X. So, let N be an arbitrary λ-term. Then having picked an arbitrary N, we have to show an impliction "if ... then ...", so we assume the antecedent, in this case that $y \notin \mathsf{FV}(x)$, and our goal is to show that the consequent $\mathsf{FV}(y[N/x]) = \mathsf{FV}(y)$ follows from this assumption. To prove that $\mathsf{FV}(y[N/x]) = \mathsf{FV}(y)$ we have to look at the definition of FV. By definition, $\mathsf{FV}(y) = \{y\}$, but we don't immediately know what $\mathsf{FV}(y[N/x])$ really looks like. We assumed that $y \notin \mathsf{FV}(x)$, i.e. $y \notin \{x\}$ so we know that $x \neq y$, so the definition of substitution gives us that $y[N/x]$ is therefore equal to y. Hence, we can deduce that also $\mathsf{FV}(y[N/x]) = \{y\}$ and so we have established that $\mathsf{FV}(y[N/x]) = \{y\} = \mathsf{FV}(y)$.

- In the second statement, we are proving Φ but with M taken to be an application PQ. This statement is an implication "if ... then ..." so we get to assume the antecedent (the *induction hypotheses*), that Φ holds of P and Φ holds of Q, i.e.:

(IH1) for all λ-terms N': if $x \notin \mathsf{FV}(P)$ then $\mathsf{FV}(P[N'/x]) = \mathsf{FV}(P)$

(IH2) for all λ-terms N': if $x \notin \mathsf{FV}(Q)$ then $\mathsf{FV}(Q[N'/x]) = \mathsf{FV}(Q)$

Here I have renamed the quantified variable to N' so it doesn't clash with anything else. Then we have to prove the consequent, that Φ holds of PQ, i.e. "for all λ-terms N: if $x \notin \mathsf{FV}(PQ)$ then $\mathsf{FV}((PQ)[N/x]) = \mathsf{FV}(PQ)$". This statement is of form "for all N", and I don't think another induction is necessary, so I will take the more direct route again by just assuming an arbitrary N: let N be an arbitrary term. Now my goal is to prove the implication, so: assume the antecedent $x \notin \mathsf{FV}(PQ)$, let's label this assumption (H1) for use later. Now the goal is to prove the consequent $\mathsf{FV}((PQ)[N/x]) = \mathsf{FV}(PQ)$. I can simplify this statement by referring to the definition of substitution: $\mathsf{FV}((PQ)[N/x] = \mathsf{FV}(P[N/x]Q[N/x])$. The goal is now to prove $\mathsf{FV}(P[N/x]Q[N/x]) = \mathsf{FV}(PQ)$. This can immediately be simplified using the definition of FV on both sides:

$$\mathsf{FV}(P[N/x]) \cup \mathsf{FV}(Q[N/x]) = \mathsf{FV}(P) \cup \mathsf{FV}(Q)$$

That's about as far as the definitions will take us, so we need to look around at what else we have in order to try to make some progress at proving this simplified goal.

Well, we assumed the two induction hypotheses and, on inspection, they are clearly going to be useful because the conclusion of the statement (IH1), with $N' \coloneqq N$, would allow us to simplify $\mathsf{FV}(P[N/x])$ to $\mathsf{FV}(P)$. Let's aim to use that. Since we have assumed that (IH1) is true, it follows that "if $x \notin \mathsf{FV}(P)$ then $\mathsf{FV}(P[N/x]) = \mathsf{FV}(P)$" is true, by instantiating the for all N by $N' \coloneqq N$. Then it follows that $\mathsf{FV}(P[N/x]) = \mathsf{FV}(P)$ is true, because we know that the antecedent in the implication, $x \notin \mathsf{FV}(P)$, is indeed true — it follows from our assumption (H1). Therefore, we have shown that $\mathsf{FV}(P[N/x]) = \mathsf{FV}(P)$ is true, let's label this (H2). Similarly, we can obtain $\mathsf{FV}(Q[N/x]) = \mathsf{FV}(Q)$ as follows. First observe that if $x \notin \mathsf{FV}(Q)$ then $\mathsf{FV}(Q[N'/x]) = \mathsf{FV}(Q)$ follows from (IH2) by instantiating $N' \coloneqq N$. Then we observe that the antecedent $x \notin \mathsf{FV}(Q)$ is

indeed true, since it follows from (H1). Therefore, we can conclude that $\mathsf{FV}(Q[N/x]) = \mathsf{FV}(Q)$, let's refer to this fact as (H3).

By that detour, we have established (H2) and (H3) since they follow logically from our induction hypotheses. Let's go back to proving our original goal, we got as far as simplifying it to: $\mathsf{FV}(P[N/x]) \cup \mathsf{FV}(Q[N/x]) = \mathsf{FV}(P) \cup \mathsf{FV}(Q)$. Using (H2) we can simplify it to: $\mathsf{FV}(P) \cup \mathsf{FV}(Q[N/x]) = \mathsf{FV}(P) \cup \mathsf{FV}(Q)$ and using (H3), to:

$$\mathsf{FV}(P) \cup \mathsf{FV}(Q) = \mathsf{FV}(P) \cup \mathsf{FV}(Q)$$

By the reflexivity of equality, we are done.

- The final statement we have to prove is when we take M in Φ to be an abstraction $\lambda y.P$. By the variable convention, we can assume that y is different from any other variable we have seen so far, so, in particular, we can immediately assume that $x \neq y$ (H1). We get to assume the induction hypothesis:

(1H1) for all N': if $x \notin \mathsf{FV}(P)$ then $\mathsf{FV}(P[N'/x]) = \mathsf{FV}(P)$.

The goal is to prove "for all N: if $x \notin \mathsf{FV}(\lambda y.P)$" then $\mathsf{FV}((\lambda y.P)[N/x]) = \mathsf{FV}(\lambda y.P)$". This is a "for all N" statement, and I don't think we need to do another induction, so I will take the direct route: let N be an arbitrary term. I now have to prove "if $x \notin \mathsf{FV}(\lambda y.P)$" then $\mathsf{FV}((\lambda y.P)[N/x]) = \mathsf{FV}(\lambda y.P)$" for this arbitrary N that I chose. So, I assume the antecedent: $x \notin \mathsf{FV}(\lambda y.P)$. I note that, it immediately follows from this that $x \notin (\mathsf{FV}(P) \setminus \{y\}$ by the definition of substitution, and therefore, by (H1), $x \notin \mathsf{FV}(P)$ (do you see why we need to use (H1) here?). Let's call this fact (H2). The goal is now to prove the consequent $\mathsf{FV}((\lambda y.P)[N/x]) = \mathsf{FV}(\lambda y.P)$. We can use the definition of substitution on the right hang side to obtain:

$$\mathsf{FV}(\lambda y.P[N/x]) = \mathsf{FV}(\lambda y.P)$$

(remember we assumed that y was different from any other variable under consideration, so in particular $y \notin \mathsf{FV}(N)$ and $y \neq x$). Then we can use the definition of FV on both sides to obtain the simplified goal:

$$\mathsf{FV}(P[N/x]) \setminus \{y\} = \mathsf{FV}(P) \setminus \{y\}$$

The definitions will not allow us to make any more progress so we should look to what else we know is already true and see if that can be applied to the goal. It follows from (IH1) by instantiating N' by N that the following implication is true: "if $x \notin \mathsf{FV}(P)$ then $\mathsf{FV}(P[N/x]) = \mathsf{FV}(P)$". Then, it follows from (H2) that the consequent $\mathsf{FV}(P[N/x]) = \mathsf{FV}(P)$ must be true. Therefore, we can use this to simplify our goal to:

$$\mathsf{FV}(P) \setminus \{y\} = \mathsf{FV}(P) \setminus \{y\}$$

This is true by the reflexivity of equality.

We have proven the three statements required by the induction principle. The principle tells us that, therefore, the following statement is true:

for all λ-terms M, for all λ-terms N: if $x \notin \mathsf{FV}(M)$ then $\mathsf{FV}(M[N/x]) = \mathsf{FV}(M)$.

This is exactly the statement of the lemma, so we are done.

What you might actually write down.

This is what the proof could look like if you extract it from my commentary.

Proof. The proof is by induction on $M \in \Lambda$.

- When M is a variable y we have to show "for all λ-terms N: if $y \notin \mathsf{FV}(x)$ then $\mathsf{FV}(y[N/x]) = \mathsf{FV}(y)$". So let N be a term and assume that $y \notin \mathsf{FV}(x)$. It follows from this assumption that $y \neq x$. We have to show $\mathsf{FV}(y[N/x]) = \mathsf{FV}(y)$ but, since $x \neq y$, by definition $\mathsf{FV}(y[N/x]) = \mathsf{FV}(y)$ and so we are left with $\mathsf{FV}(y) = \mathsf{FV}(y)$.

- When M is an application PQ, assume the induction hypotheses:

(IH1) for all λ-terms N': if $x \notin \mathsf{FV}(P)$ then $\mathsf{FV}(P[N'/x]) = \mathsf{FV}(P)$

(IH2) for all λ-terms N': if $x \notin \mathsf{FV}(Q)$ then $\mathsf{FV}(Q[N'/x]) = \mathsf{FV}(Q)$

We have to show "for all λ-terms N: if $x \notin \mathsf{FV}(PQ)$ then $\mathsf{FV}((PQ)[N/x]) = \mathsf{FV}(PQ)$". So let N be a term and assume $x \notin \mathsf{FV}(PQ)$ (H1). We have to prove $\mathsf{FV}((PQ)[N/x]) = \mathsf{FV}(PQ)$, which is, by definition:

$$\mathsf{FV}(P[N/x]) \cup \mathsf{FV}(Q[N/x]) = \mathsf{FV}(P) \cup \mathsf{FV}(Q)$$

From (H1) and the definition of FV we have that $x \notin \mathsf{FV}(P)$ and $x \notin \mathsf{FV}(Q)$. Therefore, it follows from (IH1) that $\mathsf{FV}(P[N/x]) = \mathsf{FV}(P)$ and, from (IH2) that $\mathsf{FV}(Q[N/x]) = \mathsf{FV}(Q)$. Therefore the goal is proven.

- When M is an abstraction $\lambda y.P$, we can assume that y is different from any other variable under consideration, so in particular $y \neq x$ (H1). Assume the induction hypothesis:

(1H1) for all N': if $x \notin \mathsf{FV}(P)$ then $\mathsf{FV}(P[N'/x]) = \mathsf{FV}(P)$.

Our goal is to show "for all N: if $x \notin \mathsf{FV}(\lambda y.P)$" then $\mathsf{FV}((\lambda y.P)[N/x]) = \mathsf{FV}(\lambda y.P)$" so let N be a term and assume $x \notin \mathsf{FV}(\lambda y.P)$". It follows that from this and (H1) that, therefore, $x \notin \mathsf{FV}(P)$. Our goal is to show $\mathsf{FV}((\lambda y.P)[N/x]) = \mathsf{FV}(\lambda y.P)$, but observe that this is just, by definition:

$$\mathsf{FV}(P[N/x]) \setminus \{y\} = \mathsf{FV}(P) \setminus \{y\}$$

Since we know that $x \notin \mathsf{FV}(\lambda y.P)$, it follows by definition that $x \notin \mathsf{FV}(P)$. Therefore, it follows from (IH1) that $\mathsf{FV}(P[N/x]) = \mathsf{FV}(P)$. Hence, our goal is proven.

□

6.2 Induction Principles

Each inductively defined set comes with an induction principle. You have seen Λ and you should remember $\mathbb{N}$:

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.

We get this principle because $\mathbb{N}$ can be characterised as the subset of all integers that is defined inductively by:

$$\text{(Zero)}\ \frac{}{0 \in \mathbb{N}} \qquad \text{(Succ)}\ \frac{n \in \mathbb{N}}{n+1 \in \mathbb{N}}$$

I hope you can start to see how the induction principle relates to the rules defining the inductive set. After this lecture, I will expect you to derive the induction principle from the rules, which you can do mechanically as follows:

Principle 6.1 (Induction metaprinciple)**.** *Suppose we have an inductive definition of a set S using rules $R_1, \ldots, R_k$. 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:

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

we can think of any free metavariables (e.g. M, N, x and so on) as being universally quantified at the front[1]*.*

As another example, the induction principle for $\rightarrow_\beta$ is as follows:

Suppose Φ is a property of pairs of λ-terms. If the following conditions are all met:

(i) For all terms M and N, Φ holds of $((\lambda x.M)N, M[N/x])$.

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

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

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

Then it follows that, for all pairs of λ-terms $(M, N) \in \rightarrow_\beta$, Φ holds of (M, N).

As in the case for Λ-induction, the antecedents "Φ holds of (M, N)" and "Φ holds of (N, P)", of conditions (iii) and (iv) respectively, are known as the *induction hypotheses*. Have a look back at the definition of β-conversion to see how the rules and induction principle are related.

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". I have some notes on this that you can have, just let me know if you want them.

[1] You may recall from math methods or similar that a formula of first-order logic containing free variables is, by convention, interpreted as if those variables were universally quantified.

7. Types

The untyped λ-calculus is a theory of functions over functions, in which no distinctions are made between this kind of function and that. However, in many applications we have in mind that a function has associated a certain domain and codomain over which it is intended to operate — its *type*. Typically, we are only interested in applications of a function to data that is within its domain and which yield data from its codomain — applications that are *well typed*.

Type theory concerns the formalisation of this idea of type and rules for deducing well-typedness. Of course, we shall mainly be interested in the association of types with functions, expressed as λ-terms. In the two type theories that we study in this second half of the unit, this is made precise by the expression:

$$M : A \to B$$

which we can read as "M has type $A \to B$". There are two particularly fruitful views of type theory, which help to explain the meaning of the words "has type" intuitively.

Types describe abstract properties

In this view, "type" and "property" are taken to be synonymous. Type theories are a syntax for reasoning about the properties of terms. To say $M : A \to B$ is to assert that M has a certain property, namely: given any input satisfying the property A, its output will satisfy the property B. An example is the following:

$$\lambda x.\, x + 1 : \{n \in \mathbb{Z} \mid n \text{ is even}\} \to \{n \in \mathbb{Z} \mid n \text{ is odd}\}$$

The successor function has the property that, given an even integer, it will always return an odd integer. In this view it is typical that a term may enjoy the association of many different types. For example, the same term also satisfies the property $\{n \in \mathbb{Z} \mid n \geq 0\} \to \{n \in \mathbb{Z} \mid n \neq 0\}$.

This view finds important applications in the *analysis of programs*. The second property is exactly what you need to know about $\lambda x.\, x + 1$ in order to deduce that the following program does not raise an exception:

$$100 \div (\mathsf{if}\ n \geq 0\ \mathsf{then}\ (\lambda x.\, x + 1)\ n\ \mathsf{else}\ n)$$

Types prescribe abstract classifications

In this view, "type" and "classification" are taken as synonymous. Type theories are a syntax for enforcing the classification of terms. To say $M : A \to B$ is to assert that M belongs to a certain class, namely: the class of functions with domain A and codomain B. An example is the following:

$$\lambda x.\, x +\!\!+ [1] : [\mathsf{int}] \to [\mathsf{int}]$$

The append-one function is a function from list of integers to lists of integers. In this view it is typical that we think of the type as being an intrinsic part of the description of the function. For example, we consider $\lambda x.\, x \mathbin{+\!\!+} [1] : [\mathsf{float}] \to [\mathsf{float}]$ to be a *different function*, it's just a coincidence that you happen to follow the same rule in order to compute its output.

Of course, these two views are far from mutually exclusive! On the one hand, it may only be sensible to infer properties of terms that are already classified, for example, the properties above only make sense for functions of type $\mathbb{Z} \to \mathbb{Z}$. On the other, a classification is a kind of very coarse property, for if M is classified as a function of type $[\mathsf{int}] \to [\mathsf{int}]$ then it certainly has the property that it maps inputs with the property of being a list of integers to outputs with the same property.

Just as there are an amazing variety of programming languages out there, there is an even greater variety of type systems. Each have their own set of features and good and bad properties. We will study the pure version of a family of type systems variously called Hindley-Milner, Damas-Milner, let-polymorphic, prenex polymorphic, or just: ML-style. You know that an idea is influential when it has 5 different names. This type system forms the core of all the type systems used in modern programming languages.

The original system was created by Turing-award winner Robin Milner as part of the programming language ML (hence *ML-style*) in the late 70s. It is essentially an extension of the system of Simple Types, which went all the way back to Church's efforts at creating a consistent logic in the 40s. One of the incredible properties of this family of systems is that it is possible to compute the type of a given term without needing any help from the user, and the ideas for the algorithm by which that happens are due to Milner and Roger Hindley, but the proof is due to Milner's student Luis Damas. Finally, the key idea of the system is a restriction to how polymorphic types can be formed, sometimes called the prenex polymorphism restriction, and how polymorphic types can only be assigned at lets (i.e. local definitions).

7.1 Types

We start by saying exactly what the types of this system actually look like. There are two kinds of types in this system, the *monotypes* and the *type schemes*.

Definition 7.1. *We assume a countable set of* ***type variables*** $\mathbb{A}$*, ranged over by* a*,* b*,* c *and other lowercase letters from the start of the alphabet. The* ***monotypes****, written* $\mathbb{T}$ *are a set of strings defined inductively by the following rules:*

$$\textit{(TyVar)}\ \frac{}{a \in \mathbb{T}}\ a \in \mathbb{A} \qquad \textit{(Arrow)}\ \frac{A \in \mathbb{T} \quad B \in \mathbb{T}}{(A \to B) \in \mathbb{T}}$$

We immediately adopt the following conventions to make our life less tedious:

- *We omit the outermost parenthesis when writing types.*
- *We assume that the arrow associates to the right: i.e. when we write* $A_1 \to A_2 \to A_3$*, the type that we mean is* $(T_1 \to (A_2 \to A_3))$*.*

The ***type schemes*** *are pairs consisting of a finite set of type variables* $a_1, \ldots, a_m$ *and a monotype* A*, that we write suggestively as:*

$$\forall a_1 \ldots a_m.\, A$$

Here are some examples of types and type schemes written with and without all their parentheses.

$$\begin{array}{rl} a \to b \to b & (a \to (b \to b)) \\ (a \to a) \to b & ((a \to a) \to b) \\ \forall ab.\, a \to a \to b & \forall ab.\, (a \to (a \to b)) \end{array}$$

We will consistently use A, B, C and other upper case letters at the start of the alphabet to refer to monotypes generically. We will write type schemes generically by abbreviating the quantified variables by a vector notation, e.g. $\forall \overline{a}.\, A$. It is very important that we stick to this convention because the distinction between types and type schemes is crucial to the design of the system. For this reason we will sometimes emphasize the distinction even more by referring to types as *monotypes*. However, we will consider every monotype A to be a type scheme with an empty set of quantified type variables, so when we write $\forall \overline{a}.\, A$ we include the possibility that we actually mean the monotype A (i.e $\overline{a}$ is empty), but *not conversely*: when we write A we mean a monotype that absolutely does not have any quantifiers.

Monotypes that are of the form $A_1 \to A_2$ are variously called *arrow types* or *function types* and the idea is that if M has type $A_1 \to A_2$ then it represents a function that maps terms of type A_1 to terms of type A_2. A type scheme $\forall a.\, S$ is sometimes also called a *universal type* or just *forall type*. The idea is that if a term M has type $\forall a_1 \ldots a_m.\, A$ then it can be used at any type B where B is A but with each occurrence of each a_i replaced by some monotype.

We write the type scheme $\forall a_1 \ldots a_m.\, A$ using a universal quantifier because we want to treat the type variables $a_1, \ldots, a_m$ as being bound in A. Therefore, we need to go through the same process that we did with λ-terms of making precise the notion of *free variable*, *capture-avoiding substitution* and finally *identifying expressions that differ only in the bound variable names*.

Definition 7.2. *We define the set of a type (scheme) $\forall \overline{a}.\, A$, written $\mathsf{FTV}(\forall \overline{a}.\, A)$, recursively on the syntax:*

$$\begin{array}{rcl} \mathsf{FTV}(a) & = & \{a\} \\ \mathsf{FTV}(A \to B) & = & \mathsf{FTV}(A) \cup \mathsf{FTV}(B) \\ \mathsf{FTV}(\forall a_1 \ldots a_m.A) & = & \mathsf{FTV}(A) \setminus \{a_1, \ldots, a_m\} \end{array}$$

From now on, let us consider type schemes that differ only in the choice of bound variable names to be identified. In other words, for example, $\forall a.\, a \to a = \forall b.\, b \to b$ — they are literally the same type scheme. As with terms, this allows us to choose whichever of the infinitely many representations $\forall a_1.\, a_1 \to a_1$, $\forall a_2.\, a_2 \to a_2$, $\forall a_3.\, a_3 \to a_3$,... that is most useful to us at any given time.

7.2 Type Substitutions

We said that the idea of a type scheme $\forall \overline{a}.\, A$ is that any term of this type can be used at any of the instances B that are A but with each a_i everywhere replaced by some other monotype. Therefore, to describe this precisely we will need a kind of substitution on monotypes.

When we defined term substitution, the situation was particularly simple because we only ever needed to substitute a single term for a single variable at any one time, e.g. $M[N/x]$. For types, however, we will want to substitute a whole collection of monotypes $A_1, \ldots, A_n$ for a corresponding

collection of variables $a_1, \ldots, a_n$. We will use a similar notation to terms, writing $B[A_1/a_1, \ldots, A_n/a_n]$ for the simultaneous replacement of each a_i by A_i in B.

When we come to inference, we will want to treat type substitutions, i.e. the $[A_1/a_1, \ldots, A_n/a_n]$ bit, as objects in their own right that we can combine and compare with each other. So, for types, we are going to give a definition of substituion as a first-class object. The essential content of $[A_1/a_1, \ldots, A_n/a_n]$ is that it tells you, for each a_i what should be the replacment, i.e. it's a map from type variables to types. To make life easier later, we regard it as a *total* map, i.e. defined for all possible type variables (not just those that occur in any particular term you might be considering), but which, for most variables, just says to replace that variable by itself (i.e. do nothing).

Definition 7.3. *A **type substitution** is a total map $\sigma : \mathbb{A} \to \mathbb{T}$ from type variables to monotypes, with the property that $\sigma(a) \neq a$ only for finitely many $a \in \mathbb{A}$.*

We will use σ, τ and θ to stand for type substitutions generically.

Since a type substitution σ only acts in a non-trivial way on finitely many inputs, in order to describe one it suffices to give an explanation of how it behaves just on those finitely inputs, i.e. those a that are not fixed by σ (fixed means $\sigma(a) = a$). This we will continue to do using our existing notation for substitutions. For example, we view $[b/a_1, c \to c/a_2]$ as a description of the type substitution θ that satisfies:

$$\theta(a) = \begin{cases} b & \text{if } a = a_1 \\ c \to c & \text{if } a = a_2 \\ a & \text{otherwise} \end{cases}$$

The notation $[b/a_1, c \to c/a_2]$ works as a description of θ as long as we adopt the convention that, if a type variable b is not listed on the right of any $/$, then it is fixed by the substitution. Since the definition of substitution requires that the map fixes all except finitely many type variables, this notation will work for all type substitutions.

We will also continute to write substitutions immediately after the term that we are substituting into to describe performing a substitution. For example, with θ as above, we can write $A[b/a_1, c \to c/a_2]$ for $A\theta$. The meaning of this should be intuitive:

Definition 7.4. *Given monotype A and a type substitution σ, we write $A\sigma$ for the monotype which is defined recursively as follows:*

$$\begin{aligned} a\sigma &= \sigma(a) \\ (A_1 \to A_2)\sigma &= A_1\sigma \to A_2\sigma \end{aligned}$$

So, for example, taking the substitution θ discussed above, this means that e.g. (assuming a_3 is different from the other variables)

$$\begin{aligned} (a_1 \to a_2)[b/a_1, c \to c/a_2] &= b \to (c \to c) \\ ((a_1 \to a_1) \to a_2)[b/a_1, c \to c/a_2] &= (b \to b) \to c \to c \\ (a_3 \to a_1 \to a_1)[b/a_1, c \to c/a_2] &= a_3 \to b \to b \end{aligned}$$

For the substitution $\tau := [a \to b/a,\ c/b]$ we have, e.g.

$$\begin{array}{rcl} (a \to b)\tau & = & (a \to b) \to c \\ ((a \to d) \to d)\tau & = & ((a \to b) \to d) \to d \\ ((a \to b \to b)\tau)\tau & = & ((a \to b) \to c) \to c \to c \end{array}$$

The phenomenon exhibited in the last example, where a series of type substitutions are applied to a term in sequence (in this case, the same substitution τ is applied twice) will be quite important to type inference. We abstract it out as a way of combining substitutions together:

Definition 7.5. *We write $\sigma_1\sigma_2$ for the substitution obtained by **composing** σ_2 after σ_1, defined as the following total function on type variables:*

$$(\sigma_1\sigma_2)(a) := (\sigma_1(a))\sigma_2$$

In other words, $(\sigma_1\sigma_2)(a)$ is the *type* $\sigma_1(a)$ obtained by looking up the replacement for a in σ_1 and then substituting into it according to σ_2.

The real purpose of this definition is to ensure that when you apply the composition $\sigma_1\sigma_2$ of two substitutions to a type A, it behaves like applying σ_1 to A and then applying σ_2 to the result:

$$A(\sigma_1\sigma_2) = (A\sigma_1)\sigma_2$$

You will prove this in an exercise.

8. System

The type system we are going to present is a pure version of the Damas-Milner type system.

8.1 The Type System

The whole point of types is to classify terms, so we next need to say how that comes about. We want to be able to define the notion that M has type S. Let's first agree on how to write it.

Definition 8.1. *A **type assignment** is a pair of a term M and a type A, written $M : A$. The term part of a type assignment is called the **subject** and the type part the **predicate**.*

Clearly, not all type assignments $M : A$ should be allowed: we would not expect $\lambda x\, y.\, x$ to have type $a \to a$. On the other hand, we would expect that $\lambda x.\, x : a \to a$.

The purpose of a *type system* is to provide a collection of rules in order to determine when $M : A$ is valid. In general M may contain free variables and then the type of M will usually depend on the type of the free variables. For example, term $\lambda x.\, y$ should be allowed to have a type of the form $A \to B$ where B is the type of y. So, this begs the question, what should be the type of a free variable?

We allow for *hypothetical* deductions. That is, we define when $M : A$ is valid only *with respect to* some given *assumptions* (also known as *hypotheses*) concerning the types of the free variables. These hypotheses are just type assignments of the form $x : \forall \overline{a}.\, A$, meaning that free variable x is assumed to have type (scheme) $\forall \overline{a}.\, A$, which are collected together in a *type environment*. We require that deductions can be made only under consistent assumptions.

Definition 8.2. *A **type environment**, generically written Γ, is a finite set of type assignments of the form $x : \forall \overline{a}.A$ which is, moreover, **consistent**, in the sense that if $x : \forall \overline{a}.A \in \Gamma$ and $x : \forall \overline{b}.B \in \Gamma$, then $\forall \overline{a}.A = \forall \overline{b}.B$.*

*The **subjects** of Γ is the set, written* $\mathsf{dom}\ \Gamma$*, of those term variables x for which there is some $\forall \overline{a}.A$ such that $x : \forall \overline{a}.A \in \Gamma$.*

We extend type substitution to type environments, defining:

$$\Gamma\sigma = \{x : S\sigma \mid x : S \in \Gamma\}$$

Similarly, we extend the notion of free type variables also to type environments, writing:

$$\mathsf{FTV}(\Gamma) = \bigcup \{\mathsf{FTV}(S) \mid x : S \in \Gamma \text{ for some } x\}$$

A concrete example of a type environment is $\{x : a \to b,\ y : \forall a.\ a \to a,\ z : c\}$ (whenever we write a type environment explicitly like this, we will assume that the *subjects* x, y and z are distinct variables).

The notion that a term may be assigned a type under some assumptions is made precise through the notion of *typing judgement*. A *type system* gives rules by which typing judgements can be deduced.

Definition 8.3 (Type System)**.** *A **type judgement** is a triple consisting of a type environment Γ, a λ-term M and a monotype A, which we write:*

$$\Gamma \vdash M : A$$

*Then the **type system** is the following collection of rules by which one can justify such type judgements using proof trees:*

$$x : \forall \overline{a}.\, A \in \Gamma \quad \frac{}{\Gamma \vdash x : A[\overline{B}/\overline{a}]}\ \textit{(TVar)}$$

$$\frac{\Gamma \vdash M : B \to A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : A}\ \textit{(TApp)} \qquad x \notin \operatorname{dom} \Gamma\ \frac{\Gamma \cup \{x : B\} \vdash M : A}{\Gamma \vdash \lambda x.\, M : B \to A}\ \textit{(TAbs)}$$

*In type theory, a proof tree justifying a type judgement is usually called a **type derivation**.*

Since we are never in any danger of confusion, whenever we write out a type environment explicitly as part of a judgement, we will omit the braces part of the set notation. So, for example, we will write $x : A \to B,\ y : A \vdash xy : B$ instead of the more cumbersome $\{x : A \to B,\ y : A\} \vdash xy : B$. For the same reason, if we make no assumptions then we will simply write $\vdash M : A$ rather than $\emptyset \vdash M : A$.

The rule (TVar) allows us to make use of our assumptions, it says that if we assumed $x : \forall \overline{a}.\, A$ then we can conclude any monotype instance $x : A[\overline{B}/\overline{A}]$ in which we have substituted monotypes for all the quantified variables. The following is a complete type derivation:

$$\frac{}{x : \forall ab.\, a \to b \to b,\ z : c \to c \vdash x : (c \to c) \to (c \to c) \to d}\ \text{(TVar)}$$

It's perfectly fine to make trivial substitutions, e.g. $[c \to c/a,\ b/b]$:

$$\frac{}{x : \forall ab.\, a \to a \to b,\ z : c \to c \vdash x : (c \to c) \to (c \to c) \to b}\ \text{(TVar)}$$

If the type assumed of the subject is a monotype in the environment, then there is nothing to substitute for:

$$\frac{}{x : \forall a.\, a \to a \to b,\ z : c \to c \vdash z : c \to c}\ \text{(TVar)}$$

The rule (TApp) allows us to make use of function types. It says that if we can conclude that M has a function type $A_1 \to A_2$ and N has a matching type A_1 then it follows that the application MN can be assigned the type according to what M is known to return, which is A_2. A concrete instance of this is as follows, let us write Γ for $\{x : \forall ab.\, a \to a \to b,\ z : c \to c\}$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : (c \to c) \to (c \to c) \to b}\ \text{(TVar)} \quad \dfrac{}{\Gamma \vdash z : c \to c}\ \text{(TVar)}}{\Gamma \vdash xz : (c \to c) \to b}\ \text{(TApp)} \quad \dfrac{}{\Gamma \vdash z : c \to c}\ \text{(TVar)}}{\Gamma \vdash xzz : b}\ \text{(TApp)}$$

The rule (TAbs) says that, in order to conclude that $\lambda x.\, M$ has type $A_1 \to A_2$ from some assumptions Γ, I just have to show that I can conclude that M has type A_2 under the additional assumption that x has type A_1.

$$\dfrac{\begin{array}{c}\vdots \\ x : \forall ab.\, a \to a \to b, z : c \to c \vdash xzz : b\end{array}}{x : \forall ab.\, a \to a \to b \vdash \lambda z.\, xzz : (c \to c) \to b}\ (\mathsf{TAbs})$$

Note: the type that is brought into the environment (reading from bottom to top) through (Abs) is always a monotype.

When you want to construct a type derivation for a judgement, it is usually best to start from the bottom and build upwards. This type system has a particularly nice property that makes this approach work very well. In this system, there is exactly one rule for each possible shape of term. So given $\Gamma \vdash M : A$ there is only one rule that can be used to obtain this judgement in the conclusion, and it is completely determined by the shape of M. Type systems with this property are called *syntax-directed* because the syntax of the term directs you in building a derivation.

However, that is not to say that every type derivation is completely mechanical. The rule (TApp) requires some creativity. Imagine building a derivation that has $\Gamma \vdash MN : A$ in the conclusion. The shape MN determines that we must use the (TApp) rule, but to use this rule we have to somehow invent the type B. The type B is not necessarily mentioned in the conclusion (below the line) so it requires a little bit of thought (but usually not too much) to come up with a correct instantiation. Consider building the following derivation from the ground up, with $\Gamma = \{x : \forall a.\ a \to a\}$ (I have omitted the rule names so that it fits on the page, make sure you can reconstruct them):

$$\dfrac{\dfrac{}{\Gamma \vdash x : d \to d}\qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma, y : c \to c \vdash x : (c \to c) \to d}\quad \dfrac{}{\Gamma, y : c \to c \vdash y : c \to c}}{\Gamma, y : c \to c \vdash xy : d}}{\Gamma \vdash \lambda y.\, xy : (c \to c) \to d}\qquad \dfrac{\dfrac{}{\Gamma, z : c \vdash z : c}}{\Gamma \vdash \lambda z.\, z : c \to c}}{\Gamma \vdash (\lambda y.\, xy)(\lambda z.\, z) : d}}{\Gamma \vdash x\,((\lambda y.\, xy)(\lambda z.\, z)) : d}$$

The subtree rooted at $\Gamma \vdash (\lambda y.\, xy)(\lambda z.\, z) : d$ is concluded using an instance of the rule (TApp), with $B := c \to c$, but there is no clue to why you should pick $c \to c$ in the tree below that point.

8.2 Typability and other Problems

When we introduced the system, we said that not all type assignments $M : S$ should be allowed (to be derived).

Definition 8.4. *We say that a closed term M is **typable** just if some type S such that $\vdash M : S$ is derivable in the type system.*

It's possible to talk about the typability of open terms too, by extending this definition, but a little less neat, so we will stick to closed terms. To show that a closed term M is typable requires us to exhibit a type and to show that the corresponding judgement is derivable.

For example, the closed term $\lambda xyz.xz(yz)$ is typable. Since space on the page is short, let me abbreviate the environment $\{x : a \to b \to c,\ y : a \to b, z : a\}$ as Γ and omit the rule names.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : a \to b \to c} \quad \dfrac{}{\Gamma \vdash z : a}}{\Gamma \vdash xz : b \to c} \quad \dfrac{\dfrac{}{\Gamma \vdash y : a \to b} \quad \dfrac{}{\Gamma \vdash z : a}}{\Gamma \vdash yz : b}}{\Gamma \vdash xz(yz) : c}}{x : a \to b \to c,\ y : a \to b \vdash \lambda z.xz(yz) : (a \to b) \to a \to c}}{x : a \to b \to c \vdash \lambda yz.xz(yz) : (a \to b) \to a \to c}}{\vdash \lambda xyz.xz(yz) : (a \to b \to c) \to (a \to b) \to a \to c}$$

Not all closed terms are typable. For example, the term $\lambda x.xx$ is not typable. To prove it, we will rely on analysing the possible shape of *any* typing derivation that has $\lambda x.xx$ in the conclusion.

Lemma 8.1. *The term $\lambda x.xx$ is untypable.*

Proof. We suppose $\lambda x.xx$ is typable and try to obtain a contradiction. Since $\lambda x.xx$ is typable, there is a type A such that $\vdash \lambda x.xx : A$ is derivable. Since this judgement is derivable, and its subject is an abstraction, the derivation must conclude using (TAbs) at the root:

$$\dfrac{\begin{array}{c}\vdots\\ x : B \vdash xx : C\end{array}}{\vdash \lambda x.xx : A}\ (\mathsf{TAbs})$$

but, for this to be a correct derivation, it must be that A has shape $B \to C$ for some monotypes B and C. Now consider the next part of this derivation, rooted at $x : B \vdash xx : C$. This can only be justified using the (TApp) rule:

$$\dfrac{\dfrac{\begin{array}{c}\vdots\\ x : B \vdash x : D \to C\end{array} \quad \begin{array}{c}\vdots\\ x : B \vdash x : D\end{array}}{x : B \vdash xx : C}\ (\mathsf{TApp})}{\vdash \lambda x.xx : B \to C}\ (\mathsf{TAbs})$$

which means that there must be some type D such that the operator (on the left of the application) has type $D \to C$ and the operand (on the right of the application) has type D. In this case, both operator and operand are x. Both of the remaining judgements have x as the subject, and so must be justified using instances of (TVar):

$$\dfrac{\dfrac{\dfrac{}{x : B \vdash x : D \to C}\ (\mathsf{TVar}) \quad \dfrac{}{x : B \vdash x : D}\ (\mathsf{TVar})}{x : B \vdash xx : C}\ (\mathsf{TApp})}{\vdash \lambda x.xx : B \to C}\ (\mathsf{TAbs})$$

Although we don't know what B is exactly, we know it is a *monotype*, so there are no quantified type variables to be instantiated in B. Hence, for the left-hand instance of (TVar) to be a correct, it can only be that B and $D \to C$ are the same type. Similarly, from the right-hand instance of (TVar), it must be

that B and D are the same type. Therefore, if this was a valid derivation, we would have that type $D = B = D \to C$. This is impossible! A type is just a string, and there is no finite string D that is equal to a string $D \to C$ that properly contains it. □

These insights into what shape the various types have based on the form of a given judgement, which we used throughout the foregoing proof, are neatly summarised in the following *Inversion* theorem.

Theorem 8.1 (Inversion)**.** *Suppose $\Gamma \vdash M : A$ (is derivable), then:*

- *If M is a variable x, then there is a type scheme $\forall \overline{a}.B$ (with $\overline{a}$ possibly empty) and $A = B[\overline{C}/\overline{a}]$ for some monotypes $\overline{C}$.*
- *If M is an application PQ, then there is a type B such that $\Gamma \vdash P : B \to A$ and $\Gamma \vdash Q : B$.*
- *If M is an abstraction $\lambda x.P$, then there are types B and C such that $A = B \to C$, and $\Gamma, x : B \vdash P : C$.*

Proof. By inspection of the rules. □

I want to conclude this section by describing three of the main computational problems that are considered in connection to type systems. For simplicity, we give their definitions for the case of closed terms. The problems all extend to open terms, but the solutions are slightly messier.

- **Typability.** Given a closed term M, is M typable?
- **Type checking.** Given a closed term M and a type A, is $\vdash M : A$ derivable?
- **Type inference.** Given a closed term M, compute all types A such that $\vdash M : A$.

The three problems are all related, and it is likely that if one is decidable/computable for a given type system, then all of them are. For the Damas-Milner system that we study, all three are decidable/computable.

Moreover, the Damas-Milner system occupies a kind of sweet spot among all type systems with good algorithmic properties. Any attempt to extend the Damas-Milner system to make it more powerful has resulted in one or more of these problems becoming undecidable. Hence, although more powerful type systems certainly exist, to retain decidability they require the programmer to help out the type checker by, for example, first annotating their program terms with the types that they expect them to have.

9. Analysis

We prove some useful properties of this system by analysing the rules.

9.1 Fundamental Properties

Here are some observations about type derivations in this system. Although they may not be obvious to you at this time, after you have used the system to perform a good number of derivations you will find them to be quite intuitive. Since they are quite basic observations, feel free to use them in proofs without naming them explicitly.

Lemma 9.1. *Suppose M is a term, Γ an environment and S a type. Then:*

(Subterm Closure) *If $\Gamma \vdash M : S$ is derivable and N is a subterm of M then there is some $\Gamma' \supseteq \Gamma$ and some S' such that $\Gamma' \vdash N : S'$.*

(Relevance-1) *If $\Gamma \vdash M : S$, then $\mathsf{FV}(M) \subseteq \mathsf{dom}(\Gamma)$*

(Relevance-2) *If $\Gamma \vdash M : S$, then $\{x : A \mid x : A \in \Gamma \wedge x \in \mathsf{FV}(M)\} \vdash M : S$*

(Weakening) *If $\Gamma \vdash M : S$ and $\Gamma \subseteq \Gamma'$ then $\Gamma' \vdash M : S$.*

Proof. Each of these could be proven by induction on the typing relation. However, I think the following arguments are perhaps more revealing:

- For *(Subterm Closure)*, we just observe that, in every rule, all strict subterms of the term in the conclusion are required to be typed in hypotheses. Therefore, every subterm will appear in its own node of the typing derivation tree.
- For *(Relevance-1)*, by (Subterm Closure), for every free variable x in M, we have some judgement $\Gamma' \vdash x : S'$, which requires that x be in one of the subjects of Γ'. The only rule that allows us to put new subjects into the environment as we build the tree upwards is (TAbs) and this only allows us to add variables that are not free in M. Hence, it must be that free variable x was already in Γ.
- For *(Relevance-2)*, just observe that the only rules that are predicated on the environment are (TVar) and (TAbs). If we have a derivation of $\Gamma \vdash M : S$ and Γ includes some subject that is not a free variable of M, then it can be removed without affecting the derivation. This is because the applicability of (TVar) only concerns variables that actually occur in M, and (TAbs) will only

become more applicable if subjects are removed since they require that certain things are *not in* the environment.

- The property *Weakening* is an exercise on the next problem sheet.

□

One more property I want to look at in this part is particularly important. It goes by the name of *preservation* or *subject reduction*. What is the point of a type system? A type system allows us to derive typing judgements $\Gamma \vdash M : A$, but so what? What good to us is it to know that, e.g in Haskell $\Gamma \vdash x \mathbin{++} \text{"}\texttt{foo}\text{"} : \mathsf{String}$?

The reason that it is useful to know $\Gamma \vdash x \mathbin{++} \text{"}\texttt{foo}\text{"} : \mathsf{String}$ in Haskell is that it tells us something about how $x \mathbin{++} \text{"}\texttt{foo}\text{"}$, the subject, will evaluate. If it terminates, this program expression will evaluate to an string. This is good to know because it means that when I feed it to the function $\mathsf{words} :: \mathsf{String} \to [\mathsf{String}]$ during evaluation, I can be sure that it will not to crash due to expecting one kind of input to being given another.

In other words, a key property of the type system is that, if the type system says that some complicated program expression is an A, then by the time it has finished evaluating it will still be an A. You would not have the guarantee that $\mathsf{words}(x \mathbin{++} \text{"}\texttt{foo}\text{"})$ does not crash when $x \mathbin{++} \text{"}\texttt{foo}\text{"} : \mathsf{String}$ if it could be that, despite being typed as a string, $x \mathbin{++} \text{"}\texttt{foo}\text{"}$ evaluates to an integer.

This property is called the *preservation of types under subject reduction* or more simply either *preservation* (particularly in literature on programming languages) or *subject reduction* (particularly in literature on pure λ-calculus).

Theorem 9.1 (Subject Reduction)**.** *If* $\Gamma \vdash M : A$ *and* $M \to_\beta N$ *then* $\Gamma \vdash N : A$

I leave the proof to you as part of the problem sheet. It is straightforward as long as you have the following fact to hand: types are also preserved under term substitutions. This can be formulated in the following way:

Lemma 9.2. *If* $\Gamma, x : B \vdash M : A$ *and* $\Gamma \vdash N : B$ *then* $\Gamma \vdash M[N/x] : A$.

Proof. The proof is by induction on the typing relation. Technically, we prove: forall $(\Gamma', M, A) \in \vdash$ and all Γ, if $\Gamma' = \Gamma \cup \{x : B\}$ and $\Gamma \vdash N : B$ then $\Gamma \vdash M[N/x] : A$.

(TVar) In the (Var) case, let Γ', y, A and $\overline{B}$ be arbitrary and assume $y : \forall \overline{a}.A$ is in Γ'. Then let Γ be arbitrary and suppose $\Gamma' = \Gamma \cup \{x : B\}$ and $\Gamma \vdash N : B$. We have to show $\Gamma \vdash y[N/x] : A[\overline{B}/\overline{a}]$. We analyse whether or not $x = y$:

- If $x = y$ then $y[N/x] = N$ and $\forall \overline{a}.\, A = B$ (i.e. it must be that $\overline{a}$ is empty and $A = B$). Hence $A[\overline{B}/\overline{a}] = B$ and we already have $\Gamma \vdash N : B$.
- If $x \neq y$ then $y[N/x] = y$. Since $y : \forall \overline{a}.A$ is in Γ, we have $\Gamma \vdash y : A[\overline{B}/\overline{a}]$ by (TVar).

(TApp) In this case, let P, Q, C and A be arbitrary and assume the induction hypotheses:

(IH1) For all Γ, if $\Gamma' = \Gamma \cup \{x : B\}$ and $\Gamma \vdash N : B$ then $\Gamma \vdash P[N/x] : C \to A$.

(IH2) For all Γ, if $\Gamma' = \Gamma \cup \{x : B\}$ and $\Gamma \vdash N : B$ then $\Gamma \vdash Q[N/x] : C$.

Let Γ be arbitrary and assume $\Gamma = \Gamma' \cup \{x : B\}$ and $\Gamma \vdash N : B$. It follows immediately from the induction hypotheses that $\Gamma \vdash P[N/x] : C \to A$ and $\Gamma \vdash Q[N/x] : C$. Therefore, we obtain $\Gamma \vdash P[N/x]Q[N/x] : A$ by (TApp) and the desired goal follows by the definition of substitution.

(TAbs) In this case, let y, P, C and D be arbitrary and assume the induction hypotheses:

(IH1) For all Γ'', if $\Gamma' \cup \{y : C\} = \Gamma'' \cup \{x : B\}$ and $\Gamma'' \vdash N : B$ then $\Gamma'' \vdash P[N/x] : D$.

Let Γ be arbitrary and assume $\Gamma' = \Gamma \cup \{x : B\}$ and $\Gamma \vdash N : B$. We have to show $\Gamma \vdash (\lambda y.P)[N/x] : C \to D$. We can assume by the variable convention that y does not occur outside P. In particular, $y \neq x$ and $y \notin \mathsf{FV}(N)$ (*). Hence, take $\Gamma'' = \Gamma \cup \{y : C\}$ in (IH1). Since $\Gamma \vdash N : B$, it follows from Weakening that also $\Gamma'' \vdash N : B$. Hence, (IH1) yields $\Gamma'' \vdash P[N/x] : D$, which is to say that $\Gamma \cup \{y : C\} \vdash P[N/x] : D$. Hence we obtain $\Gamma \vdash \lambda y.P[N/x] : C \to D$ by (TAbs) and the desired result follows by the definition of substitution and (*).

□

9.2 Subformula Property

The next property I want to look at is a bit less obvious. We already observed that, given a judgement $\Gamma \vdash M : A$, building a type derivation for the judgement is not a completely trivial process, because using the rule (TApp) requires you to invent a type B that works as both the left hand side of the arrow type and the type of the operand.

$$\frac{\Gamma \vdash M : B \to A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : A}\ (\mathsf{TApp})$$

Sometimes, this B will be determined by the environment. For example, with the environment $\Gamma = \{x : ((b \to b) \to (b \to b)) \to a\}$, consider the following derivation:

$$\dfrac{\dfrac{}{\Gamma \vdash x : ((b \to b) \to (b \to b)) \to a}\ (\mathsf{TVar}) \quad \dfrac{\dfrac{}{\Gamma,\, y : b \to b \vdash y : b \to b}\ (\mathsf{TVar})}{\Gamma,\, y : b \to b \vdash \lambda y.\, y : (b \to b) \to (b \to b)}\ (\mathsf{TAbs})}{\Gamma \vdash x(\lambda y.\, y) : a}\ (\mathsf{TApp})$$

Clearly, in this instance of the (TApp) rule, there is no choice over what type to use as B. On the other hand, consider the following schematic derivation for the judgement closed term $\lambda z.(\lambda x.z)(\lambda y.y)$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{z : a,\, x : C \to C \vdash z : a}\ (\mathsf{TVar})}{z : a \vdash \lambda x.z : (C \to C) \to a}\ (\mathsf{TAbs}) \quad \dfrac{\dfrac{}{z : a,\, y : C \vdash y : C}\ (\mathsf{TVar})}{z : a \vdash \lambda x.z : (C \to C)}\ (\mathsf{TAbs})}{z : a \vdash (\lambda x.z)(\lambda y.y) : a}\ (\mathsf{TApp})}{\vdash \lambda z.(\lambda x.z)(\lambda y.y) : a \to a}\ (\mathsf{TAbs})$$

This type derivation will be correct for *any* choice of type C. Therefore, if we consider building this derivation from the ground up, at the point where we are about to use (TApp) and we need a type B

to form the arrow $B \to a$ that will be the type of $\lambda x.z$, we can choose any of the infinitely many that have shape $C \to C$ for some type C.

However, this freedom disappears if we restrict our attention to closed terms in β-normal form. We prove a more general property, in which the term may contain free varibales, but they are only assigned monotypes by the environment. The proof is quite combinatorial in nature, that is, we have to analyse all the different cases (*combinations* of terms and types and rules) that can occur in order to obtain the desired conclusion.

Theorem 9.2 (The Subformula Property)**.** *Suppose* Γ *assigns only monotypes to its subjects and suppose* M *is in* β*-normal form and* $\Gamma \vdash M : A$*. Then the derivation of this judgement is unique and all of the types mentioned in the derivation are substrings of the types mentioned in the conclusion.*

Proof. The proof is by induction on the typing relation.

(Var) In this case, M is a variable x. There is only one derivation tree that concludes $\Gamma \vdash x : A$, which is:

$$\frac{}{\Gamma \vdash x : A}\ (\mathsf{TVar})$$

Clearly, all the types mentioned in the whole derivation are substrings of those in the conclusion because the whole derivation is the conclusion!

(Abs) In this case, M is an abstraction $\lambda x.P$ and A has shape $B \to C$. Assume the induction hypothesis:

(IH) if $\Gamma \cup \{x : B\}$ contains only monotypes and P is in normal form then there is a unique derivation of $\Gamma \cup \{x : B\} \vdash P : C$ and all the types in this derivation already occur as substrings of those in the conclusion.

It is true that $\Gamma \cup \{x : B\}$ contains only monotypes because, by definition, B is a monotype. Moreover, it is also true that P is in normal form because $\lambda x.P$ is (and any subterm of a term in normal form must itself be in normal form — otherwise the original term could make a step). Hence, we can use the induction hypothesis to conclude that $\Gamma \cup \{x : B\} \vdash P : C$ has a unique derivation and all of the types occurring anywhere in that derivation already occur (as substrings) of the types in the conclusion. Let us call this derivation δ'. Now, to create a derivation for $\Gamma \vdash \lambda x.P : B \to C$ we have no choice but to use (TAbs) as follows:

$$\frac{\begin{array}{c}\vdots\ \delta' \\ \Gamma \cup \{x : B\} \vdash P : C\end{array}}{\Gamma \vdash \lambda x.P : B \to C}\ (\mathsf{TAbs})$$

In other words, it can only be that the derivation continues by deriving the conclusion $\Gamma \cup \{x : B\} \vdash P : C$. Since δ' is the only way to derive this conclusion, there can only be one way to derive $\Gamma \vdash \lambda x.P : B \to C$. Let us call this larger derivation δ, i.e. δ is the proof tree obtained by glueing $\Gamma \vdash \lambda x.P : B \to C$ onto the bottom of δ'.

Now, suppose that D is a type occurring anywhere inside δ. We have to show that D therefore also occurs in δ's conclusion $\Gamma \vdash \lambda x.P : B \to C$, possibly as a substring. Well, either D occurs in

δ' or it occurs in the conclusion $\Gamma \vdash \lambda x.P : B \to C$. In the latter case we are done. In the former case, we know from the induction hypothesis that D must therefore occur as a substring of one of the types occurring in $\Gamma \cup \{x : B\} \vdash P : C$. However, every type in that judgement also occurs in the final conclusion $\Gamma \vdash \lambda x.P : B \to C$, so we are done.

(TApp) In this case M is of shape PQ. Let B be arbitrary. We assume the induction hypotheses:

(IH1) If Γ contains only monotypes and P is in normal form, then there is a unique derivation of $\Gamma \vdash P : B \to A$ and every type occurring anywhere in this derivation already occurs (possibly as a substring) in the conclusion.

(IH2) If Γ contains only monotypes and Q is in normal form, then there is a unique derivation of $\Gamma \vdash Q : B$ and every type occurring anywhere in this derivation already occurs (possibly as a substring) in the conclusion.

Suppose Γ contains only monotypes and PQ is in normal form. Then P and Q are in normal form separately, and it follows from the induction hypotheses that $\Gamma \vdash P : B \to A$ and $\Gamma \vdash Q : B$ each have a unique derivation satisfying in which every type that occurs already occurs in the respective conclusions. Now consider all possible ways of deriving $\Gamma \vdash PQ : A$. Any derivation of this conclusion must proceed by (TApp):

$$\frac{\Gamma \vdash P : C \to A \quad \Gamma \vdash Q : C}{\Gamma \vdash PQ : A}\ (\mathsf{TApp})$$

for some choice of type C. We are going to show that, under the given circumstances, there is exactly one C that will work. To see this, consider looking "inside" P. This term P can only either be another application or a variable. It is not possible for P to be an abstraction because then PQ would be a redex, but PQ is in normal form. If P is an application $P_1 P_2$, consider looking again inside P_1, this can also only either be an application or a variable, for the same reason. If we continue to unfold all the left operators so long as they are applications, eventually we will stop when we find a variable. Hence, the general shape of PQ is $((\cdots((x\,Q_1)Q_2)\cdots)Q_{n-1})Q_n$ for some n (possibly 0). Correspondingly, the shape of the subderivation concluding $\Gamma \vdash P : C \to A$ is:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : C_1 \to \cdots \to C_n \to C \to A}\ (\mathsf{TVar}) \quad \vdots \quad \Gamma \vdash Q_1 : C_1}{\Gamma \vdash x\,Q_1 : C_2 \to \cdots \to C_n \to C \to A}\ (\mathsf{TApp}) \quad \vdots \quad \Gamma \vdash Q_2 : C_2}{\Gamma \vdash x\,Q_1\,Q_2 : C_3 \to \cdots \to C_n \to C \to A}\ (\mathsf{TApp}) \quad \vdots \quad \vdots}{\dfrac{\Gamma \vdash x\,Q_1\,Q_2 \cdots Q_{n-1} : C_n \to C \to A \qquad \Gamma \vdash Q_n : C_n}{\Gamma \vdash x\,Q_1\,Q_2 \cdots Q_{n-1}\,Q_n : C \to A}\ (\mathsf{TApp})}$$

And, since Γ contains only monotypes, it must be that $x : C_1 \to \cdots \to C_n \to C \to A$ is contained in Γ. The shape of this derivation is, as usual, determined by the shape of the term $x\,Q_1\,Q_2 \cdots Q_{n-1}\,Q_n$ in the conclusion, but also, because this term is headed by a variable x, all

the types $C_1, \ldots, C_n$ and C in this subderivation are determined by the type assigned to this x in the environment. Hence, in particular, there is no choice over what C can be, for the derivation to work you have to choose C to be the same type as the $(n+1)$th argument of x, according to the type assigned to x in the environment.

We have earlier shown that, for the particular type B (which was arbitrary), $\Gamma \vdash P : B \to A$ and $\Gamma \vdash Q : B$ have unique derivations satisfying the subformula property. However, we now know that B can only be C, namely the $(n+1)$th argument of x, according to the environment. Hence, we have that there is a unique derivation δ_1 of $\Gamma \vdash P : C \to A$ and δ_2 of $\Gamma \vdash Q : C$ in which all the types that occur already occur in the respective conclusions. Hence, there is only one way to derive $\Gamma \vdash PQ : A$, namely to use (TApp) with $\Gamma \vdash P : C \to A$ and $\Gamma \vdash Q : C$ as premises, derived by δ_1 and δ_2.

Now suppose D is a type occurring anywhere in this whole derivation. Then either D occurs in δ_1, δ_2 or in the conclusion. In the latter case, we are already done. In the former cases, it follows that D occurs as a substring of a type in $\Gamma \vdash P : C \to A$ or $\Gamma \vdash Q : C$, but since C occurs in Γ (as the $n+1$th argument type of x), it follows that all such types already occur in $\Gamma \vdash PQ : A$.

□

10. Constraints

In this and the next chapter we will show how we can compute, algorithmically, the type of a term in a given environment, or otherwise conclude its untypability. Being able to do this automatically, i.e. by an algorithm, is a key feature of Damas-Milner type systems. Without this property, you would be forced to annotate your programs with the types of certain expressions. Although it is good practice to write the types of top-level definitions anyway, without a general algorithm that the compiler can use to infer types, you would also need to annotate every local definition (e.g. in a where clause) with its type, every lambda abstraction, every use of a polymorphic function etc. etc.

The goal is develop an algorithm to solve the following problem.

> Given a closed term M, what are the types A such that $\vdash M : A$.

It is actually more convenient to solve a slightly more general problem, in which the term may be open and an environment given as part of the input. However, this alternative problem is less simple to state, so we will first give the algorithm.

Let's begin by making some simple observations so that we understand the problem better. A first observation is that there is not generally a single unique type that we can assign to a given term in a given environment. For example, the term $\lambda x.\, x$ in the empty environment can be given infinitely many types:

$$\begin{array}{lcl} \vdash & \lambda x.\, x & : \quad a \to a \\ \vdash & \lambda x.\, x & : \quad (b \to b) \to b \to b \\ \vdash & \lambda x.\, x & : \quad ((c \to c) \to c) \to (c \to c) \to c \\ \vdash & \lambda x.\, x & : \quad ((d \to d \to d) \to d \to d \to d \\ & \vdots & \qquad \vdots \end{array}$$

A second observation is that the type of a compound term depends on the types of its subterms. For example, to give a type to $(\lambda x.\, x)\ (\lambda x.\, y)$, we will need to know the type of $\lambda x.\, y$ and to give a type to $\lambda x.\, y$ we will need to know the type of y.

A third observation is that when we consider the type of subterms, we really need to know about *all* the types of each subterm. For example, if we are interested in typing $(\lambda x.\, x)\ (\lambda y.\, y)$, it is not enough simply to know that $\lambda x.\, x$ can be given the type $a \to a$, because $\lambda y.\, y$ cannot be given the type a. We need to know that another possible typing for $\lambda x.\, x$ is $(a \to a) \to a \to a$. When typing $(\lambda x.\, x)(\lambda y.\, y)(\lambda z.\, z)$ the types that we assign to each of these identities will depend on the types of the others, so we really need to know all the possibilties.

10.1 Intuitions

The syntax-directed nature of the rules tells us a lot about how to build a derivation mechanically. If we want to build a type derivation for e.g. $\lambda z.\, z\ x$ in the environment $\{x : f \to f\}$, then we know immediately that we can restrict our attention to derivations with the following shape:

$$\dfrac{\dfrac{\dfrac{}{x : f, z :? \vdash z :?}\ (\mathsf{TVar}) \quad \dfrac{}{x : f, z :? \vdash x :?}\ (\mathsf{TVar})}{x : f, z :? \vdash z\ x :?}\ (\mathsf{TApp})}{x : f \vdash \lambda z.\, z\ x :?}\ (\mathsf{TAbs})$$

Therefore, working out the set of possible monotypes that we can assign to this term is just a matter of working out all possible ways of filling out the question marks.

Let's do this in a principled fashion. Each question mark stands for a potentially different type (except the type of z in the environment which is always the same), so let's assign a distinct type variable to each of them.

$$\dfrac{\dfrac{\dfrac{}{z : b \vdash z : e}\ (\mathsf{TVar}) \quad \dfrac{}{z : b \vdash x : d}\ (\mathsf{TVar})}{z : b \vdash z\ x : c}\ (\mathsf{TApp})}{\vdash \lambda z.\, z\ x : a}\ (\mathsf{TAbs})$$

The idea is that some instantiations of the type variables will result in a correct type derivation and some intantitions will not. For example, the following assignment gives a correct derivation:

$$\begin{array}{lcl} a & \mapsto & ((f \to f) \to g) \to g \\ b & \mapsto & (f \to f) \to g \\ c & \mapsto & g \\ d & \mapsto & f \to f \\ e & \mapsto & (f \to f) \to g \\ f & \mapsto & f \end{array}$$

From which we can conclude that $\lambda z.\, z\ x$ can be assigned the type $((f \to f) \to g) \to g$. Whereas, as you can easily verify, the following assignment does not give a correct type derivation:

$$\begin{array}{lcl} a & \mapsto & ((f \to f) \to g \\ b & \mapsto & f \to f \\ c & \mapsto & g \\ d & \mapsto & f \to f \\ e & \mapsto & f \to f \\ f & \mapsto & f \end{array}$$

Why does this second assignment not give correct derivation? The reason is that it does not give a correct instance of the (TApp) rule:

$$\dfrac{x : f \to f, z : f \to f \vdash z : f \to f \quad x : f \to f, z : f \to f \vdash x : f \to f}{x : f \to f, z : f \to f \vdash z\ x : g}\ (\mathsf{TApp?})$$

Why is this not a correct instance of the (TApp) rule? Because the (TApp) rule requires that the type of the operator has shape $B \to A$, that the type of the operand is exactly B and the type in the conclusion is exactly A. Here, we do not have that $z : (f \to f) \to g$.

It is a remarkable fact that whether or not an assignment gives a correct type derivation or not can be entirely characterised in terms of whether the type variables a, b, c, d, e and f have certain shapes with respect to each other. Let me try to convince you.

As we saw above, every correct instance of (TApp) in the derivation above:

$$\text{(TApp)}\ \frac{x : f \to f, z : b \vdash z : e \quad x : f \to f, z : b \vdash \mathsf{Succ} : d}{x : f \to f, z : b \vdash z\ \mathsf{Succ} : c}$$

must have the type assigned to e being an arrow with the type assigned to d on the left hand side and the type assigned to c on the right. In other words, the assignment must make the following equation true:

$$e \stackrel{?}{=} d \to c$$

The first assignment makes this equation true because if we substitute g for c, $f \to f$ for d and $(f \to f) \to g$ for e in the equation we get:

$$(f \to f) \to g = (f \to f) \to g$$

which is obviously correct, but the second assignment gives the incorrect:

$$f \to f \neq (f \to f) \to g$$

Next consider the use of (TVar) in the derivation:

$$\text{(TVar)}\ \frac{}{x : f \to f, z : b \vdash z : e}$$

Clearly, every correct instance of (TVar) here requires that:

$$b \stackrel{?}{=} e$$

This is obviously satisfied by the first assignment, which has e and b both assigned to $f \to f$. It is also satisfied by the second assignment, but we already saw that that derivation has problems elsewhere. In the use of (TVar) in the derivation:

$$\text{(TVar)}\ \frac{}{x : f \to f, z : b \vdash x : d}$$

we have to have d being assigned to the type of x according to the environment, i.e. $f \to f$. In other words, the assignment must satisfy:

$$d \stackrel{?}{=} f \to f$$

Finally, in the use of (TAbs):

$$\text{(TAbs)}\ \frac{x : f \to f, z : b \vdash z\ x : c}{x : f \to f \vdash \lambda z.z\ x : a}$$

For the instance to be correct, we must have the type a being an arrow from b to c, i.e. the assignment must satisfy:

$$a \stackrel{?}{=} b \to c$$

It is easy to see that this is satisfied by both the given assignments.

In summary, we have surmised that every assignment to $a \multimap f$ that yields a correct type derivation must satisfy the equations:

$$\begin{aligned} e &\stackrel{?}{=} d \to c \\ b &\stackrel{?}{=} e \\ d &\stackrel{?}{=} \mathsf{Nat} \to \mathsf{Nat} \\ a &\stackrel{?}{=} b \to c \end{aligned}$$

In fact, a little thought should convince you that, conversely, any assignment to $a \multimap f$ that satisfies the equations will give a correct type derivation of the above shape. So, these type equations completely characterise typability for this combination of term and environment: any assignment we find that satisfies the equations will give a correct type derivation and, conversely, any correct type derivation will give an assignment that satisfies the equations. So, to do type inference, we don't need to think about derivations at all, since we can instead just think about sets of type equations, which are simpler.

Definition 10.1. *A **type constraint** is just a pair of monotypes (A, B) written suggestively as $A \stackrel{?}{=} B$.*

The idea of this $\stackrel{?}{=}$ notation is that we don't want to confuse a statement $A = B$ that two types A and B are identical, with a *constraint* $A \stackrel{?}{=} B$ which is an object in the type inference algorithm recording a *requirement* that A and B can be *made* identical by some assignment to the type variables therein.

Polymorphism

I now want to look at a simple example that includes polymorphism. What types can we infer for xx in the environment that has $x : \forall a.\ a \to a$? We know that the derivation will have the following shape.

$$\dfrac{\dfrac{}{x : \forall a.\ a \to a \vdash x : c}\ (\mathsf{TVar}) \qquad \dfrac{}{x : \forall a.\ a \to a \vdash x : d}\ (\mathsf{TVar})}{x : \forall a.\ a \to a \vdash xx : b}\ (\mathsf{TApp})$$

As before, to be a correct use of the (TApp) rule, we need that any assignment to b, c and d makes $c \stackrel{?}{=} d \to b$. What about c and d?

We know that c should be a monotype *instance* of $\forall a.\ a \to a$. In other words, of shape $A \to A$ for any monotype A. We can require that any type that is assigned to c by type inference has shape $A \to A$ by saying that the assignment as a whole should make the equation

$$c \stackrel{?}{=} e \to e$$

true, where e is just another fresh (i.e. not before used) type variable. This means we can assign e.g. $(b \to b) \to b \to b$ to c, in which case e is assigned to $b \to b$, or $a \to a$, in which case e is assigned to a.

It prevents us from assigning e.g. c to c because, if we fix c as c, there can be no assignment to e to satisfy the equation $c \stackrel{?}{=} e \to e$. Going back to the derivation, we will make the same kind of requirement on d, i.e, any assignment should satisfy:

$$d \stackrel{?}{=} f \to f$$

for some fresh type variable f.

Through our analysis we have summised that to be a correct derivation, we need the types that are assigned to b, c and d to satisfy the following equations:

$$\begin{array}{rcl} c & \stackrel{?}{=} & d \to b \\ c & \stackrel{?}{=} & e \to e \\ d & \stackrel{?}{=} & f \to f \end{array}$$

You can easily check that, for example, the assignment $c \mapsto (a \to a) \to a \to a$, $d \mapsto a \to a$ and $b \mapsto a \to a$ makes these equations true (with $e \mapsto a \to a$ and $f \mapsto a$). Therefore, a valid typing is $x : \forall a.\ a \to a \vdash xx : a \to a$. On the other hand, assigning c to $a \to a$ does not lead to any correct derivation because it implies that $d \stackrel{?}{=} a$ which makes the final equation unsatisfiable.

In general, the idea is that, to characterise the correct instances d of a type scheme $\forall a_1 \ldots a_k.\ T$, we will pick k fresh type variables $f_1, \ldots, f_k$ and require that $d \stackrel{?}{=} T[f_1/a_1, \ldots, f_k/a_k]$. For example, the instances d of $\forall ab.\ a \to (c \to a \to b) \to c$ are required to satisfy e.g. (with g and h fresh):

$$d \stackrel{?}{=} g \to (c \to g \to h) \to c$$

10.2 H-M Constraint Generation

Milner developed his seminal type inference algorithm for the programming language ML, which has a couple of extra complications compared to our pure calculus (but is conceptually the same). The algorithm was simply called W — for *well typed*: i.e. checking that a term can be well typed.

On our calculus, the algorithm is only slightly more sophisticated than a prior algorithm developed by Roger Hindley. For this reason, it is also called *Hindley-Milner Type Inference*. Our presentation of the algorithm is in two parts:

1. Constraint generation.

2. Constraint solving.

The part of the algorithm responsible for constraint generation is given as a kind of functional program in Figure 10.1.

A key feature of this part of the algorithm is the ability to generate fresh type variables. The meaning of the word *fresh* here is that, when let a be fresh is executed, a should be a new type variable never before used by the algorithm in the current execution. You could implement this by representing type variables as numbered atoms a_1, a_2, a_3 etc. and keeping a global counter n to track

$\mathsf{CGen}(\Gamma, x) =$
  let a be fresh
  let $\forall a_1 \ldots a_k. A = \Gamma(x)$
  let $b_1, \ldots, b_k$ be fresh
  $(a, \{a \stackrel{?}{=} A[b_1/a_1, \ldots, b_k/a_k]\})$

$\mathsf{CGen}(\Gamma, MN) =$
  let a be fresh
  let $(b, C_1) = \mathsf{CGen}(\Gamma, M)$
  let $(c, C_2) = \mathsf{CGen}(\Gamma, N)$
  $(a, C_1 \cup C_2 \cup \{b \stackrel{?}{=} c \to a\})$

$\mathsf{CGen}(\Gamma, \lambda x. M) =$
  let a be fresh
  let b be fresh
  let $(c, C) = \mathsf{CGen}(\Gamma \cup \{x : b\}, M)$
  $(a, C \cup \{a \stackrel{?}{=} b \to c\})$

Figure 10.1: Constraint Generation Algorithm

how many variables you have used so far. When a fresh type variable is required, the system gives a_{n+1} and increments the counter.

This presentation of constraint generation takes as input an environment Γ and a term M and yields as output a type variable and a set of constraints. We assume that $\mathsf{FV}(M) \subseteq \mathsf{dom}(\Gamma)$. The idea is as follows. Suppose $\mathsf{CGen}(\emptyset, M) = (a, C)$ and there is an assignment that satisfies all the constraints in C (makes them true) and sets a to A. Then it follows that $\vdash M : A$ is a valid typing for M.

This presentation of the algorithm peforms the labelling of subterms with type variables, that we described informally above, on-the-fly: whenever we recurse into a new subterm (which corresponds to moving up into a new hypothesis in the type derivation), we immediately generate a fresh type variable which represents the type of that subterm.

In Figure 10.2 I have traced out the execution of the algorithm on an example. The fresh variables are just numbered sequentially in the order that they are generated, a_0 is the first fresh variable, a_1 the second and so on. I hope that you can see the relationship between the call tree of the CGen algorithm and the type derivation for the subject $\lambda x. x(\lambda y. yx)$. At each level in the tree a fresh variable is immediately created, which is the variable labelling the type in the conclusion of the corresponding level of the type derivation. This is the type variable in the left component of the pair returned by each recursive call. Then the constraints associated with that level are just the union of the constraints from all lower levels and the constraints forced by the rule used in the derivation at that level. Ultimately we end up with a set of constraints C_0 as follows. We will see in the next part

how to solve such constraints mechanically.

$$
\begin{aligned}
a_3 &\stackrel{?}{=} a_1 \\
a_7 &\stackrel{?}{=} a_5 \\
a_8 &\stackrel{?}{=} a_1 \\
a_7 &\stackrel{?}{=} a_8 \to a_6 \\
a_4 &\stackrel{?}{=} a_5 \to a_6 \\
a_3 &\stackrel{?}{=} a_4 \to a_2 \\
a_0 &\stackrel{?}{=} a_4
\end{aligned}
$$

You will be asked to derive the type constraints (type equations) using the type inference algorithm as part of problems. It doesn't matter whether you explicitly follow the recursive algorithm or whether you simply draw and label the type derivation with fresh type variables, and then write down all the constraints arising (as we did in the previous sections). As long as you obtain all the constraints (7 in this case), it doesn't matter which approach you take or how you named the variables.

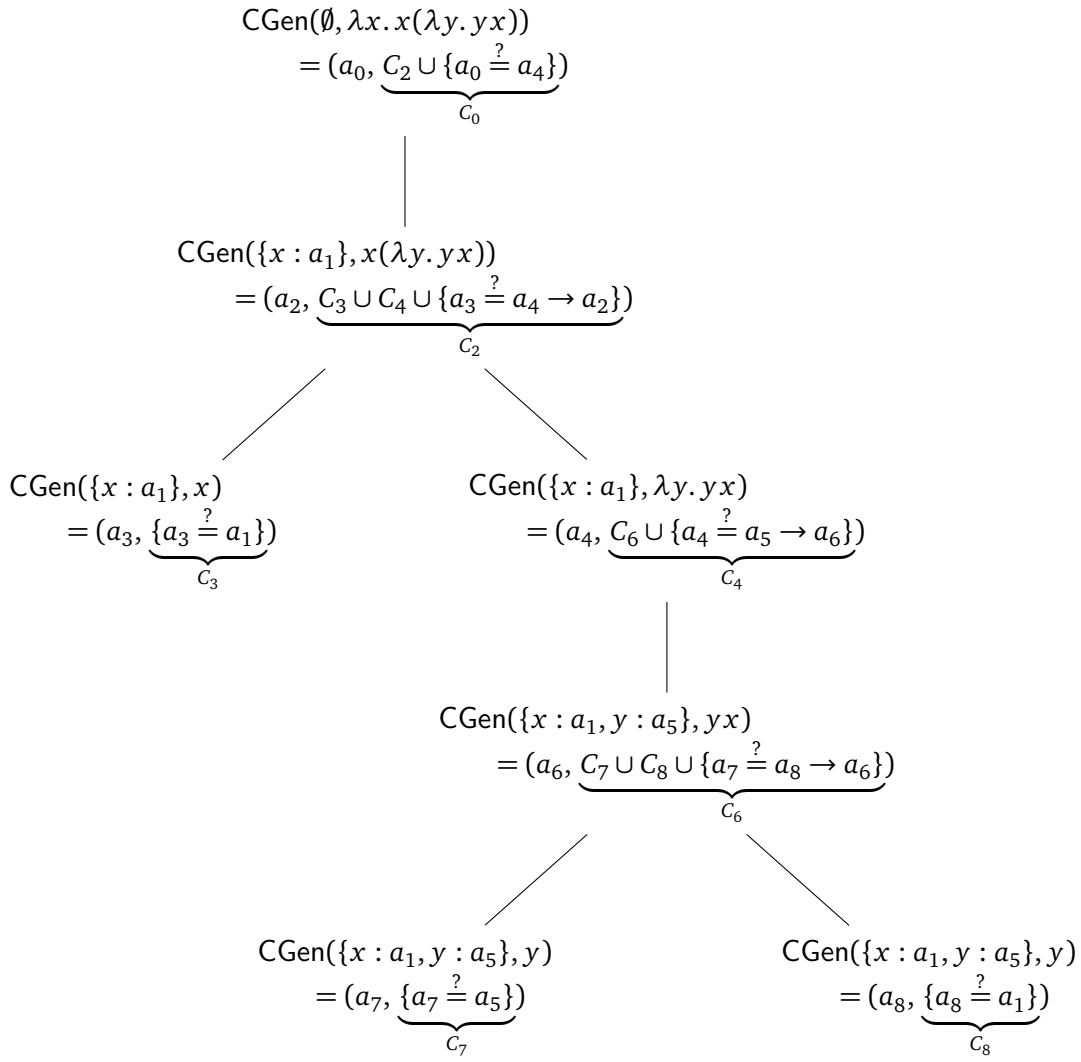


Figure 10.2: Sample run of constraint generation algorithm.

11. Unification

In this lecture, we look at how to solve sets of type constraints and how those solutions relate to type inference.

So far, we have talked about assigments of types to type variables that either do or do not satisfy sets of type constraints. However, an assignment of types to type variables is just a type substitution.

Definition 11.1. *A **unifier** or **solution** to a finite set of type constraints* $\{A_1 \stackrel{?}{=} B_1, \ldots, A_n \stackrel{?}{=} B_n\}$ *is a type substitution* σ *such that, for all* $1 \leq i \leq n$, $A_i\sigma = B_i\sigma$.

We will write $\mathscr{C}\sigma$ *for the application of* σ *to all the types involved in* $\mathscr{C}$.

For example, the following unification problem has a solution in the substitution σ defined to its the right:

$$\begin{array}{rcl} c & \stackrel{?}{=} & d \to b \\ c & \stackrel{?}{=} & e \to e \\ d & \stackrel{?}{=} & f \to f \end{array} \qquad \left[\begin{array}{rcl} a \to a & / & b \\ (a \to a) \to a \to a & / & c \\ a \to a & / & d \\ a \to a & / & e \\ a & / & f \end{array}\right]$$

This is because applying this substitution to both sides of each constraint results in a true equation, in the sense that:

$$\begin{array}{rl} & c\sigma = (a \to a) \to a \to a = d\sigma \to b\sigma = (d \to b)\sigma \\ \text{and} & c\sigma = (a \to a) \to a \to a = e\sigma \to e\sigma = (e \to e)\sigma \\ \text{and} & d\sigma = a \to a = f\sigma \to f\sigma = f\sigma \end{array}$$

This is not the only solution to this unification problem. The following substitution σ' is also a unifier of the given constraints:

$$\left[\begin{array}{rcl} (b \to b) \to b \to b & / & b \\ ((b \to b) \to b \to b) \to (b \to b) \to b \to b & / & c \\ (b \to b) \to b \to b & / & d \\ (b \to b) \to b \to b & / & e \\ b \to b & / & f \end{array}\right]$$

We can check by computing:

$$\begin{aligned} c\sigma = &\ ((b \to b) \to b \to b) \to (b \to b) \to b \to b \\ = &\ d\sigma \to b\sigma \\ = &\ (d \to b)\sigma \end{aligned}$$

$$\begin{aligned} c\sigma = &\ ((b \to b) \to b \to b) \to (b \to b) \to b \to b \\ = &\ e\sigma \to e\sigma \\ = &\ (e \to e)\sigma \end{aligned}$$

$$d\sigma = \ (b \to b) \to b \to b = f\sigma \to f\sigma = f\sigma$$

Actually, we didn't need to check explicitly that this σ' is a unifier. Instead we can just observe that σ' is $\sigma[b \to b/a]$, i.e. the substitution obtained by first doing σ and then replacing every a in the result by $b \to b$. Since σ already made all the equations true, i.e. $A_i\sigma = B_i\sigma$, then performing an extra substitution $[b \to b/a]$ onto *both sides* doesn't change the situation, $(A_i\sigma)[b \to b/a] = (B_i\sigma)[b \to b/a]$ must also be true.

The unification problem on the left has a solution in the substitution σ defined on the right:

$$\begin{array}{lcl} e & \stackrel{?}{=} & d \to c \\ b & \stackrel{?}{=} & e \\ d & \stackrel{?}{=} & f \to f \\ a & \stackrel{?}{=} & b \to c \end{array} \qquad \left[\begin{array}{rcl} ((f \to f) \to (g \to g)) \to (g \to g) & / & a \\ (f \to f) \to (g \to g) & / & b \\ (g \to g) & / & c \\ f \to f & / & d \\ (f \to f) \to (g \to g) & / & e \end{array}\right]$$

because the constraints are made true equations by the action of σ:

$$\begin{array}{ll} & e\sigma = (f \to f) \to (g \to g) = d\sigma \to c\sigma = (d \to c)\sigma \\ \text{and} & b\sigma = (f \to f) \to (g \to g) = e\sigma \\ \text{and} & d\sigma = f \to f \\ \text{and} & a\sigma = ((f \to f) \to (g \to g)) \to (g \to g) = b\sigma \to c\sigma = (b \to c)\sigma \end{array}$$

Finally, consider the following unification problem, consisting of only one constraint:

$$a \stackrel{?}{=} a \to b$$

This unification problems does not have a solution. To see why, we can reason as follows. We want to show that there is no solution, so we suppose there is one and seek a contradiction. Suppose there is a unifier σ for the problem. Then, in particular, $a\sigma = (a \to b)\sigma = a\sigma \to b\sigma$. Recall that a type is just a string, a piece of syntax. Is it possible for these two strings to be identical moduluo a change of bound variable names? Let $|A| \in \mathbb{N}$ be the number of arrow symbols in type A. Since $a\sigma = a\sigma \to b\sigma$ we must have that the following two natural numbers are identical $|a\sigma| = |a\sigma \to b\sigma|$. However, $|a\sigma \to b\sigma| = |a\sigma| + |b\sigma| + 1$ and subtracting $|a\sigma|$ from both sides we obtain $0 = |b\sigma| + 1$, our contradiction.

With this notion of unifier, we can state formally the invariant associated with the constraint generation part of type inference.

Lemma 11.1 (CGen: Recursive Invariant)**.** *Suppose* $\mathsf{CGen}(\Gamma, M) = (a, \mathscr{C})$.

- *If* σ *is a unifier for* $\mathscr{C}$ *then* $\Gamma\sigma \vdash M : \sigma(a)$ *is a derivable typing judgement.*
- *If* $\Gamma\sigma \vdash M : \sigma(a)$ *is a derivable typing judgement, then* σ *is a unifier for* $\mathscr{C}$.

In particular, when Γ is empty because M is closed, this says that σ is a unifier of $\mathscr{C}$ iff $\vdash M : \sigma(a)$.

11.1 Robinson's Algorithm

So the constraints generated by algorithm CGen characterise every possible valid typing for closed input M. The next step in the algorithm is to solve the constraints. For this purpose we adopt an even older idea, due to Robinson, for solving sets of equations over expressions of any kind. A version of this will be able to solve our type constraints or otherwise detecting their unsolvability. We give a presentation in terms of rewriting.

Definition 11.2. *A set of type constraints of the shape* $\mathscr{C} = \{a_1 \stackrel{?}{=} A_1, \ldots, a_m \stackrel{?}{=} A_m\}$ *is in* ***solved form*** *just if each of the* a_i *are pairwise distinct variables none of which occurs in any of the* A_i.

In such a case, we define $[\![\mathscr{C}]\!] := [A_1/a_1, \ldots, A_m/a_m]$.

Clearly, if $\mathscr{C}$ is in solved form, then $[\![\mathscr{C}]\!]$ is a solution. Thus, a strategy is to try to find a solved form corresponding to a given set of type constraints. The following rewrite rules implement such a strategy.

Definition 11.3. *Suppose* $\mathscr{C}$ *is a set of type constraints. The* ***unification algorithm*** *consists of applying the following rules to* $\mathscr{C}$ *as many times as possible.*

$$
\begin{array}{lrcll}
(1) & \{A \stackrel{?}{=} A\} \uplus \mathscr{C} & \Longrightarrow & \mathscr{C} & \\
(2) & \{A_1 \to A_2 \stackrel{?}{=} B_1 \to B_2\} \uplus \mathscr{C} & \Longrightarrow & \{A_1 \stackrel{?}{=} A_1, A_2 \stackrel{?}{=} B_2\} \uplus \mathscr{C} & \\
(3) & \{A \stackrel{?}{=} a\} \uplus \mathscr{C} & \Longrightarrow & \{a \stackrel{?}{=} A\} \uplus \mathscr{C} & \text{if } A \notin \mathbb{A} \\
(4) & \{a \stackrel{?}{=} A\} \uplus \mathscr{C} & \Longrightarrow & \{a \stackrel{?}{=} A\} \uplus \mathscr{C}[A/a] & \text{if } a \in \mathsf{FTV}(\mathscr{C}) \setminus \mathsf{FTV}(A)
\end{array}
$$

The symbol $\uplus$ here means disjoint union, i.e. its two arguments must have no element in common. The free type variables in $\mathscr{C}$ of shape $\{A_1 \stackrel{?}{=} B_1, \ldots A_k \stackrel{?}{=} B_k\}$ is the set:

$$\mathsf{FTV}(\mathscr{C}) := \bigcup\{\mathsf{FTV}(A_i) \mid 1 \le i \le k\} \cup \bigcup\{\mathsf{FTV}(B_i) \mid 1 \le i \le k\}$$

but notice that, since all the type variables occurring in a monotype are free, this set is nothing but the set of all type variables occurring in $\mathscr{C}$.

The way to read the rules is as follows.

- Rule (1) says that, to find a solution for $\{A \stackrel{?}{=} A\} \uplus C$ it is equivalent to find a solution for $\mathscr{C}$.
- Rule (2) says that, to find a solution for $\{A_1 \to A_2 \stackrel{?}{=} B_1 \to B_2\} \uplus \mathscr{C}$, one can equivalently just find a solution for the two parts of the arrow separately $\{A_1 \stackrel{?}{=} B_1, A_2 \stackrel{?}{=} B_2\} \uplus \mathscr{C}$.

- Rule (3) says that, the orientation of the equations should make no difference to solvability.
- Rule (4) says that, any solution to $\{a \stackrel{?}{=} A\} \uplus C$ must have a assigned to A as long as a is itself not in A. So, finding a solution to $\mathscr{C}$, in such a case, is equivalent to finding a solution to $\mathscr{C}[A/a]$.

Technically, this "algorithm" is non-deterministic, but we could always impose an order on the rules and the equations if we wanted (e.g. use the first applicable rule starting from the top).

Consider again the problem:

$$\begin{array}{rcl} c & \stackrel{?}{=} & d \to b \\ c & \stackrel{?}{=} & e \to e \\ d & \stackrel{?}{=} & f \to f \end{array}$$

Applying the last rule to the first equation we obtain:

$$\begin{array}{rcl} c & \stackrel{?}{=} & d \to b \\ d \to b & \stackrel{?}{=} & e \to e \\ d & \stackrel{?}{=} & f \to f \end{array}$$

Applying the last rule to the last equation, we obtain:

$$\begin{array}{rcl} c & \stackrel{?}{=} & (f \to f) \to b \\ (f \to f) \to b & \stackrel{?}{=} & e \to e \\ d & \stackrel{?}{=} & f \to f \end{array}$$

Applying the second rule to the second equation, we obtain:

$$\begin{array}{rcl} c & \stackrel{?}{=} & (f \to f) \to b \\ f \to f & \stackrel{?}{=} & e \\ b & \stackrel{?}{=} & e \\ d & \stackrel{?}{=} & f \to f \end{array}$$

Applying the last rule to the third equation we obtain:

$$\begin{array}{rcl} c & \stackrel{?}{=} & (f \to f) \to e \\ f \to f & \stackrel{?}{=} & e \\ b & \stackrel{?}{=} & e \\ d & \stackrel{?}{=} & f \to f \end{array}$$

Applying the third rule to the second equation we obtain:

$$\begin{array}{rcl} c & \stackrel{?}{=} & (f \to f) \to e \\ e & \stackrel{?}{=} & f \to f \\ b & \stackrel{?}{=} & e \\ d & \stackrel{?}{=} & f \to f \end{array}$$

Note: we are not yet in solved form, because a variable on the left appears also on the right. We can apply the last rule to the second equation:

$$\begin{array}{lcl} c & \stackrel{?}{=} & (f \to f) \to f \to f \\ e & \stackrel{?}{=} & f \to f \\ b & \stackrel{?}{=} & f \to f \\ d & \stackrel{?}{=} & f \to f \end{array}$$

No more rules are applicable and a quick glance verifies that we are in solved form. The substitution defined by this solved form is:

$$\left[\left[\begin{array}{lcl} c & \stackrel{?}{=} & (f \to f) \to f \to f \\ e & \stackrel{?}{=} & f \to f \\ b & \stackrel{?}{=} & f \to f \\ d & \stackrel{?}{=} & f \to f \end{array}\right]\right] = \left[\begin{array}{lll} (f \to f) \to f \to f & / & c \\ f \to f & / & b \\ f \to f & / & e \\ f \to f & / & d \end{array}\right]$$

Call this substitution σ. Obviously σ is a unifier for this solved form. More importantly, it is a unifier for the original set of type constraints, in fact it is essentially the unifier we discussed in the previous section (modulo variable names).

Let's look at the other problem we already discussed:

$$\begin{array}{lcl} e & \stackrel{?}{=} & d \to c \\ b & \stackrel{?}{=} & e \\ d & \stackrel{?}{=} & f \to f \\ a & \stackrel{?}{=} & b \to c \end{array}$$

Applying the last rule to the first equation we obtain:

$$\begin{array}{lcl} e & \stackrel{?}{=} & d \to c \\ b & \stackrel{?}{=} & d \to c \\ d & \stackrel{?}{=} & f \to f \\ a & \stackrel{?}{=} & b \to c \end{array}$$

Applying the last rule to the second equation, we obtain:

$$\begin{array}{lcl} e & \stackrel{?}{=} & d \to c \\ b & \stackrel{?}{=} & d \to c \\ d & \stackrel{?}{=} & f \to f \\ a & \stackrel{?}{=} & (d \to c) \to c \end{array}$$

Finally, applying the last rule to the third equation, we obtain:

$$\begin{array}{rcl} e & \stackrel{?}{=} & (f \to f) \to c \\ b & \stackrel{?}{=} & (f \to f) \to c \\ d & \stackrel{?}{=} & f \to f \\ a & \stackrel{?}{=} & ((f \to f) \to c) \to c \end{array}$$

No more rules are applicable to this set of type constraints and we can observe that it is in solved form since all the variables on the left are different to each other and none of them appear on the right. The substitution defined by the solved form is:

$$\left[\!\left[\begin{array}{rcl} e & \stackrel{?}{=} & (f \to f) \to c \\ b & \stackrel{?}{=} & (f \to f) \to c \\ d & \stackrel{?}{=} & f \to f \\ a & \stackrel{?}{=} & ((f \to f) \to c) \to c \end{array} \right]\!\right] = \left[\begin{array}{rcl} ((f \to f) \to c) \to c & / & a \\ (f \to f) \to c & / & b \\ f \to f & / & d \\ (f \to f) \to c & / & e \end{array} \right]$$

It is easy to verify that this is a unifier for the original set of type constraints.

Theorem 11.1. *Suppose $\mathscr{C}$ is a set of type constraints and $\mathscr{D}$ is the problem obtained from it by the unification algorithm. Then $\mathscr{C}$ is solvable iff $\mathscr{D}$ is in solved form.*

Proof. We will skip the proof because, although straightforward, it would take a little time that we don't have. □

However, it is *different* from the one we came up with earlier, which was:

$$\left[\begin{array}{rcl} ((f \to f) \to (g \to g)) \to (g \to g) & / & a \\ (f \to f) \to (g \to g) & / & b \\ (g \to g) & / & c \\ f \to f & / & d \\ (f \to f) \to (g \to g) & / & e \end{array} \right]$$

In fact, there is a sense in which the unifer that we computed using the unification algorithm, let's call it σ_2, is a better unifier than the one we discussed previously, let's call that one σ_1. The unifier σ_1 can be *obtained* from σ_2 by post composing it with $[g \to g/c]$, i.e. $\sigma_1 = \sigma_2[g \to g/c]$.

We should consider the unifier that we computed, σ_2, to be better because we know that if σ is a unifier for a given problem C, then we know that $\sigma[g \to g/c]$ will also be a unifier even without looking at the problem C: if σ makes all the constraints into equalities, then they will remain equalities no matter what substitution we apply to both sides afterwards. However, if all I know is that $\sigma[g \to g/c]$ is a unifier for some problem C, then I don't know whether or not σ on its own is a unifier without knowing the particulars of C. There are no guarantees. For example, $[c \to c/a][g \to g/c]$ is a unifier for $g \to g \to g \to g \stackrel{?}{=} a$, but $[c \to c/a]$ is not.

Hence, if I have two unifiers σ_1 and σ_2 to choose from for some problem, and it turns out that σ_1 can be obtained from σ_2 by post-composing σ_2 with some other substitution σ_3, i.e. $\sigma_1 = \sigma_2\sigma_3$, then I should always choose σ_2. Knowing that σ_2 is a unifier already tells me that σ_1 is also a unifier (and indeed that any $\sigma_2\sigma$ at all will be a unifier), but not conversely. This leads to the following idea.

Definition 11.4. *Let $\mathscr{C}$ be a set of type constraints. Then we say that σ is a* ***most general unifier*** *(mgu) of $\mathscr{C}$ just if:*

(i) σ is a unifier of $\mathscr{C}$

(ii) and every unifier σ' of $\mathscr{C}$ is of shape $\sigma\sigma''$ for some σ''

Being a most general unifier is quite a strong claim: if σ is most general then *every* other possible unifier for the given problem can be obtained from σ by post-composition. It is easy to see when a unifier is most general in simple cases. Consider the following problem:

$$\begin{array}{rcl} a & \stackrel{?}{=} & b \to b \\ b & \stackrel{?}{=} & c \to c \end{array}$$

A most general unifier for this problem is $\sigma = [(c \to c) \to c \to c/a, c \to c/b]$. Let's check. First (i) is obviously satisfied, this substitution makes the two equations true. Now for (ii), suppose σ' is some other unifier. That means that $a\sigma' = b\sigma' \to b\sigma'$ and $b\sigma' = c\sigma' \to c\sigma'$. Therefore, using the second equation in the first, $a\sigma' = (c\sigma' \to c\sigma') \to c\sigma' \to c\sigma'$. This type, $(c\sigma' \to c\sigma') \to c\sigma' \to c\sigma'$ is just $((c \to c) \to c \to c)\sigma'$, i.e. $a\sigma\sigma'$. So we have $a\sigma' = a\sigma\sigma'$ Similarly, $b\sigma' = b\sigma' \to b\sigma' = b\sigma\sigma'$. Since, for any other variable c, $c\sigma = c$, it follows trivially that $c\sigma' = c\sigma\sigma'$. Hence, for all variables $d \in \mathbb{A}$, $d\sigma' = d\sigma\sigma'$, i.e. $\sigma' = \sigma\sigma'$.

Theorem 11.2. *If $\mathscr{C}$ is solvable and $\mathscr{D}$ is the problem obtained by running the unification algorithm, then $[\![\mathscr{D}]\!]$ is an mgu.*

Proof. We only sketch the proof, which has two parts.

- Suppose $\mathscr{D}_1 \Longrightarrow \mathscr{D}_2$ for any set of type constraintss $\mathscr{D}_1$ and $\mathscr{D}_2$. It is easy to convince yourself that σ is a unifier for $\mathscr{D}_1$ iff σ is a unifier for $\mathscr{D}_2$. Therefore, if we start from $\mathscr{C}$, we neither lose nor gain any solutions by applying the rewrite rules: $\mathscr{D}$ has exactly the same solutions as $\mathscr{C}$.
- Suppose $\mathscr{C}$ is solvable. Then, $\mathscr{D}$ is in solved form. We argue that $[\![D]\!]$ is an mgu for $\mathscr{D}$ and, by the first part, $\mathscr{D}$ has all the same unifiers as $\mathscr{C}$, so it will therefore be an mgu for $\mathscr{C}$ too.

  First, we already saw that $[\![\mathscr{D}]\!]$ is obviously a unifier for $\mathscr{D}$. To see that it is most general, let $\mathscr{D} = \{a_1 = A_1, \ldots, a_m = A_m\}$ and suppose σ is a unifier for $\mathscr{D}$. Fix a type variable a. We proceed by cases:

  - If a is some a_i on the left of $\mathscr{D}$, then since σ is a unifier, we have $a_i\sigma = A_i\sigma$. But A_i is exactly $a_i[\![\mathscr{D}]\!]$, so $A_i\sigma = a_i[\![\mathscr{D}]\!]\sigma$.
  - If a is not on the left of $\mathscr{D}$, then $a[\![\mathscr{D}]\!] = a$ and trivially $a\sigma = a[\![\mathscr{D}]\!]\sigma$.

  Hence, $\sigma = [\![\mathscr{D}]\!]\sigma$.

□

Corollary 11.1. *Every solvable set of type constraints has a most general solution.*

Proof. If $\mathscr{C}$ is solvable then we can actually construct an mgu by running the unification algorithm. □

11.2 Principal Types

We have seen that the constraint generation part of type inference constructs a set of type constraints that completely characterises all the possible typings for a closed term M. That is, if $\mathsf{CGen}(\emptyset, M) = (a, \mathscr{C})$ then not only does every unifier σ give rise to a valid type judgement $\vdash M : \sigma(a)$, but every valid judgement $\vdash M : A$ arises as some solution $A = \sigma(a)$. We have also seen that there is always a *best* choice of solution, a most general unifier from which all other unifiers can be obtained via composition. Since unifiers are in correspondence with valid types, the most general unifier σ gives rise to a most general monotype $\sigma(a)$. Every valid monotype for M arises as $\tau(a)$ for some unifier τ, and every unifier τ can be obtained from the mgu σ by performing another substitution. Therefore, every valid monotype for M can be obtained from the monotype $\sigma(a)$ arising from the mgu σ, via substitution.

Definition 11.5. *This completes our presentation of the **Hindley-Milner Type Inference Algorithm**. On input closed term M:*

1. *Generate constraints $\mathscr{C}$ and type variable a using $\mathsf{CGen}(\emptyset, M)$.*
2. *Solve $\mathscr{C}$ using Robinson's algorithm to obtain mgu σ or deduce unsolvability.*
3. *If $\mathscr{C}$ has no solution then M is untypable. Otherwise return $\sigma(a)$.*

Theorem 11.3 (Principal Type Scheme Theorem)**.** *If closed term M is typable, then Hindley-Milner type inference returns a type A that is **principal** in the sense that:*

- *$\vdash M : A$ is derivable*
- *and, if some other $\vdash M : B$ is derivable, then there is a choice of monotypes $C_1, \ldots, C_k$ such that $B = T[C_1/a_1, \ldots, C_k/a_k]$.*

Proof. The first point follows immediately from the invariant of constraint generation expressed in Lemma 11.1. For the second point, suppose $\vdash M : B$, let $(a, \mathscr{C})$ be the pair returned by $\mathsf{CGen}(\emptyset, M)$ and let σ be the mgu returned by Robinson's algorithm, so that $\sigma(a) = A$. By Lemma 11.1, there is a unifier θ for $\mathscr{C}$ such that $\theta(a) = B$. Then, since σ is an mgu, there is a substitution τ such that $\theta = \sigma\tau$. Therefore, $(\sigma\tau)(a) = \theta(a) = B$ and it follows that $\tau(a_1), \ldots, \tau(a_k)$ is a choice of monotypes such that $A[\tau(a_1)/a_1, \ldots, \tau(a_k)/a_k] = B$. □

12. Normalisation

We have seen that type systems restrict the programs that you can write in an attempt to prevent you from doing "bad" things. For example, we saw that $\lambda x.xx$ is not typable in STLC. In this lecture I want to prove to you a remarkable result about the expressive power of the simply typed lambda calculus: no non-terminating program is typable! All the terms of the STLC are *strongly normalising*, we say the calculus itself is strongly normalising.

12.1 Logical Relation

It's not a completely straightforward result to prove, however, because the property of a term being strongly normalising is not inductive in the structure of the typing relation, in other words: you can't prove it (directly) by induction.

To see concretely why not, suppose you are trying to prove:

$\Gamma \vdash M : A$ implies M is strongly normalising

by induction on $\vdash$. According to the induction principle for $\vdash$, the (App) case requires a proof of the following statement:

P is SN and Q is SN implies PQ is SN

but this is not provable, because it is not true. Consider, for example, $P = Q = \lambda x.xx$. Both P and Q are SN (indeed, they are in normal form), but PQ is Ω.

Instead, as is typical, we must find a stronger property that *is* inductive, and which implies SN – the property we are interested in.

We will instead attempt to prove that $\Gamma \vdash M : A$ implies $M \in \mathsf{R}(A)$.

Definition 12.1. *Define* $\mathsf{R} : \mathbb{S} \to \mathscr{P}(\Lambda)$ *is a function from types to sets of terms defined recursively as follows.*

$$\begin{array}{rcl} \mathsf{R}(a) & = & \{M \mid M \textit{ is SN}\} \\ \mathsf{R}(A \to B) & = & \{M \mid \forall N \in \mathsf{R}(A).\, MN \in \mathsf{R}(B)\} \\ \mathsf{R}(\forall \overline{a}.\, A) & = & \{M \mid \textit{for all } \overline{B} \in \mathscr{P}(\mathbb{S}),\, M \in \mathsf{R}(A[\overline{B}/\overline{a}])\} \end{array}$$

That is: $M \in \mathsf{R}(a)$ just if M is strongly normalising and $M \in \mathsf{R}(A \to B)$ just if M returns $\mathsf{R}(B)$ things when given $\mathsf{R}(A)$ things and $M \in \mathsf{R}(\forall \overline{a}.\, A)$ just if M is in R at every instance of $\forall \overline{a}.\, A$. In fact, we shall now show that, for all A (not only for type variables a) $M \in \mathsf{R}(A)$ implies M is strongly normalising. With this extra information, we will be able to read $M \in \mathsf{R}(A \to B)$ as saying "M is SN *and*, moreover, M maps strongly normalising inputs to strongly normalising outputs". Hence, we are proving a

stronger property $\Gamma \vdash M : A$ implies $M \in \mathsf{R}(A)$, i.e. implies M is SN and, if M is of arrow type, also that M maps SN inputs to SN outputs.

Before we can show that $\mathsf{R}(A)$ contains only SN terms (no matter what A is), it will be first useful to show that $\mathsf{R}(A)$ contains a particular subset of SN terms, namely those that are headed by a variable:

$$x\ N_1\ \cdots\ N_k$$

for some variable x and some $k \geq 0$, where each N_i is a strongly normalising term. Let VHSN be the set of variable-headed strongly normalising terms:

$$\mathit{VHSN} := \{x\ N_1\ \cdots\ N_k \mid x \in \mathbb{V} \wedge k \geq 0 \wedge \forall i \in [1..k].\ N_i \in \mathit{SN}\}$$

It turns out it is easiest to prove that R contains all the variable headed terms at the same time as proving that R contains only strongly normalising terms. This is because, when we try to prove it by induction on the type A, in the (Arrow) case $B \to C$:

- To prove that all terms in R are SN it is useful to know that $x \in \mathsf{R}(B)$ for every variable x.
- To prove that all VHSN terms are in R, it is useful to know that any terms in $\mathsf{R}(B)$ are guaranteed to be SN.

Let's see how it plays out.

Lemma 12.1. *for all A: $\mathit{VHSN} \subseteq R(A) \subseteq \mathit{SN}$*

Proof. If we unpack the set theory, we get:

for all A:

(i) for all M: $M \in \mathit{VHSN}$ implies $M \in \mathsf{R}(A)$

(ii) for all M: $M \in \mathsf{R}(A)$ implies M is SN

and so we can invoke proof by induction on A.

(TyVar) When A is a type variable a, we reason as follows.

(i) For part (i), let $M \in \mathit{VHSN}$, so M is of shape $x\ N_1\ \cdots\ N_k$ with each N_i being SN. The only reductions that can happen in M happen inside each of the N_i, so it must be that M as a whole is also SN. Therefore, by definition, $M \in \mathsf{R}(a)$.

(ii) For part (ii), $M \in \mathsf{R}(a)$ implies M is SN by definition.

(Arrow) When A is an arrow $B \to C$, we reason as follows. Assume the induction hypotheses:

(IH1) for all P: $P \in \mathit{VHSN}$ implies $P \in \mathsf{R}(B)$.

(IH2) for all P: $P \in \mathit{VHSN}$ implies $P \in \mathsf{R}(C)$.

(IH3) for all P: $P \in R(B)$ implies P is SN

(IH4) for all P: $P \in R(C)$ implies P is SN

We prove conjuncts (i) and (ii) separately:

(i) For (i), let $M \in \mathit{VHSN}$ so that M has shape $x\ N_1\ \cdots\ N_k$ with each N_i being SN. To see that $M \in \mathsf{R}(B \to C)$, we suppose $Q \in \mathsf{R}(B)$ and look to show $MQ \in \mathsf{R}(C)$. It follows from (IH3) that Q must be SN and therefore MQ is the correct shape to be an element of VHSN. We then conclude by (IH2).

(ii) For (ii), assume $M \in \mathsf{R}(B \to C)$. Then, by definition, we have that for all $N \in \mathsf{R}(B)$, $MN \in \mathsf{R}(C)$. Take $N \coloneqq x$, a variable. Since $x \in \mathit{VHSN}$, it follows from (IH1) that $x \in \mathsf{R}(B)$ and therefore Mx is SN. Therefore, it must be that M is SN (otherwise: if M had an infinite reduction sequence, then you could replay it in Mx).

□

It is useful to know a kind of subject expansion property, namely if $N \in \mathsf{R}(B)$ and the contraction $M[N/x]$ of some redex is in $\mathsf{R}(A)$ then also the redex was $(\lambda x.M)N \in \mathsf{R}(A)$. The proof uses a kind of unfolding of the recursive definition of $\mathsf{R}(A)$. We can view any type A as having shape:

$$B_1 \to B_2 \to \cdots \to B_m \to b$$

for some $m \geq 0$ and some type variable b (the idea is that, if $m = 0$, then this expression is just b). If we perform a similar "unfolding" to definition of $\mathsf{R}(A)$, we get:

> $M \in \mathsf{R}(B_1 \to B_2 \to \cdots B_m \to b)$ just if for all $N_1 \in \mathsf{R}(B_1)$, for all $N_2 \in \mathsf{R}(B_2)$, $\ldots$, $N_m \in \mathsf{R}(B_m)$:
> $M\ N_1\ N_2\ \cdots N_m \in \mathsf{R}(b)$

but being an element of $\mathsf{R}(b)$, as in the conclusion, just means being SN.

Lemma 12.2. *If, for all $N \in \mathsf{R}(B)$: $M[N/x] \in \mathsf{R}(A)$, then for all N: $(\lambda x.M)N \in \mathsf{R}(A)$.*

Proof. Without loss of generality, we may imagine A to be of shape $B_1 \to \cdots \to B_m \to b$ for some $m \geq 0$ and $b \in \mathbb{A}$. Assume (*) for all $N \in \mathsf{R}(B)$: $M[N/x] \in \mathsf{R}(A)$. By Lemma 12.1, we can take $N \coloneqq x$ and obtain $M[x/x] = M \in \mathsf{R}(A)$. Let $N \in \mathsf{R}(B)$ and $N_i \in \mathsf{R}(B_i)$ for each $i \in [1..m]$. Then it follows from Lemma 12.1 that M, N and each N_i is SN. We must show that $(\lambda x.M)N\ N_1\ \cdots\ N_m$ is SN. To see this, assume that there is an infinite reduction sequence starting from this term and obtain a contradiction. We split this into two parts:

(i) We show any such infinite reduction sequence must have a finite prefix of the form:

$$(\lambda x.M)\ N\ N_1\ \cdots\ N_m \twoheadrightarrow_\beta (\lambda x.M')\ N'\ N_1'\ \cdots\ N_m' \to_\beta M'[N'/x]\ N_1' \cdots N_m'$$

with $M \twoheadrightarrow_\beta M'$, $N \twoheadrightarrow_\beta N'$ and each $N_i \twoheadrightarrow_\beta N_i'$. In other words: there are reduction steps (possibly none) inside each of the subterms until eventually the head redex is contracted.

(ii) We show that any term of the latter shape is SN (and therefore, the finite reduction sequence leading to this term can only be extended a finite amount, contradicting the assumption it was infinite).

For (i), we observe that the infinite reduction sequence *cannot only* contract redexes inside each of the subterms M, N and each N_i since each of these is SN. Therefore, it must at some point contract the head redex. In other words, after finitely many reduction steps, the sequence must reach a term of shape

$$(\lambda x.\, M')N'\; N_1'\; \cdots\; N_m'$$

where $M \twoheadrightarrow_\beta M'$, $N \twoheadrightarrow_\beta N'$ and each $N_i \twoheadrightarrow_\beta N_i'$ and then proceed by contracting the head redex, yielding a term of shape $M'[N'/x]\; N_1' \cdots N_m'$. For (ii), since all the primed terms are reducts of the unprimed ones, we can also obtain this same term by a finite reduction sequence from a different starting point:

$$M[N/x]\; N_1 \cdots N_m \twoheadrightarrow_\beta M'[N'/x]\; N_1' \cdots N_m'$$

The advantage in this view is that we know that $M[N/x] \in \mathsf{R}(A)$ and each $N_i \in \mathsf{R}(B_i)$ so we get, by definition, that the whole term $M[N/x]\; N_1 \cdots N_m \in \mathsf{R}(b)$ and hence is SN by definition. If a term is SN, then so are all its reducts, so we deduce that $M'[N'/x]\; N_1' \cdots N_m'$ is SN too. □

12.2 The Fundamental Theorem

Our ultimate aim is to show that every simply-typed λ-term is SN, by showing that they are all in R. We will do this next. It will be useful to use the following notation. Let N_x be a family of terms indexed by variables $x \in X$. Then $M[N_x/x \mid x \in X]$ is the term obtained by substituting each N_x for the corresponding x, for each x in X. For example, if $X = \{y, z\}$ and our family of terms indexed by the elements of X are $N_y = \mathbf{I}$ and $N_z = x$, then $(y\; (y\; z)))[N_x/x \mid x \in X] = \mathbf{I}\; (\mathbf{I}\; x)$.

Theorem 12.1. *Suppose* $\Gamma \vdash M : B$ *and, for each variable* $x \in \mathsf{dom}(\Gamma)$, *there is a term* N_x *such that* $N_x \in \mathsf{R}(\Gamma(x))$. *Then* $M[N_x/x \mid x \in \mathsf{dom}(\Gamma)] \in \mathsf{R}(B)$.

Proof. The proof is by induction on M. Let us abbreviate the above k-element type environment by Γ.

(Var) If M is a variable x, by inversion it must be in $\mathsf{dom}(\Gamma)$ and, consequently, $M[N_x/x \mid x \in \mathsf{dom}(\Gamma)] = N_x$ and $\Gamma \vdash x : B$. Without loss of generality, let $\Gamma(x)$ be of shape $\forall \overline{a}.\, A$. It follows from inversion that there are types $\overline{B}$ such that $A[\overline{B}/\overline{a}] = B$. By the assumption on N_x, it follows that $N_x \in \mathsf{R}(A[\overline{B}/\overline{a}]) = \mathsf{R}(B)$, which was the goal.

(App) In the application case, let P and Q be terms and assume the induction hypotheses:

(IH1) For all C, Γ: if $\Gamma \vdash P : C$ and for each $x \in \mathsf{dom}(\Gamma)$ there is a term $N_x \in \mathsf{R}(\Gamma(x))$ then $P[N_x/x \mid x \in \mathsf{dom}(\Gamma)] \in \mathsf{R}(C)$.

(IH2) For all C, Γ: if $\Gamma \vdash Q : C$ and for each $x \in \mathsf{dom}(\Gamma)$ there is a term $N_x \in \mathsf{R}(\Gamma(x))$ then $Q[N_x/x \mid x \in \mathsf{dom}(\Gamma)] \in \mathsf{R}(C)$.

Assume $\Gamma \vdash PQ : B$ and some $N_x \in \mathsf{R}(\Gamma(x))$ for each $x \in \mathsf{dom}(\Gamma)$. Then, by inversion, it must be that there is a type B' such that $\Gamma \vdash P : B' \to B$ and $\Gamma \vdash Q : B'$. It follows from (IH1) that $P[N_x/x \mid x \in \mathsf{dom}(\Gamma)] \in \mathsf{R}(B' \to B)$ and, by (IH2), that $Q[N_x/x \mid x \in \mathsf{dom}(\Gamma)] \in \mathsf{R}(B')$. By definition of R, we have $P[N_x/x \mid x \in \mathsf{dom}(\Gamma)]Q[N_x/x \mid x \in \mathsf{dom}(\Gamma)] = (PQ)[N_x/x \mid x \in \mathsf{dom}(\Gamma)] \in \mathsf{R}(B)$, which was our goal.

(Abs) In the abstraction case, let P be a term and assume the induction hypothesis:

(IH) For all C, Δ: if $\Delta \vdash P : C$ and, for each $x \in \mathsf{dom}(\Delta)$ there is a term $N_x \in \mathsf{R}(\Delta(x))$ then $P[N_x/x \mid x \in \mathsf{dom}(\Delta)] \in \mathsf{R}(C)$.

Assume $\Gamma \vdash \lambda y.P : B$. By inversion it must be of shape $B = B_1 \to B_2$ such that $\Gamma, y : B_1 \vdash P : B_2$. By the variable convention, we may assume that y does not occur outside of P. Assume there is some $N_x \in \mathsf{R}(\Gamma(x))$ for each $x \in \mathsf{dom}(\Gamma)$. Our goal is to show $(\lambda y.P)[N_x/x \mid x \in \mathsf{dom}(\Gamma)] \in \mathsf{R}(B_1 \to B_2)$, so let $Q \in \mathsf{R}(B_1)$ and we will attempt to show that $(\lambda y.P)Q \in \mathsf{R}(B_2)$. We will obtain this conclusion by instantiating Lemma 12.2 at $N \coloneqq Q$. For this we need to show that, for all $N \in \mathsf{R}(B_1)$, $P[N/y] \in \mathsf{R}(B_2)$. So let N_y be an arbitrary term in $\mathsf{R}(B_1)$ and then let us construct $\Delta = \Gamma \cup \{y : B_1\}$. Then we can invoke (IH) to obtain $P[N_x/x \mid x \in \mathsf{dom}(\Delta)] \in \mathsf{R}(B_2)$. Since y does not occur in any N_x for $x \in \Gamma$, this is the same as $P[N_x/x \mid x \in \mathsf{dom}(\Gamma)][N_y/y] \in \mathsf{R}(B_2)$. Hence, we conclude by Lemma 12.2.

□

Corollary 12.1 (Strong Normalisation)**.** *For all typable M, M is SN.*

Proof. Suppose M is typable. By definition, this means there is Γ and A such that $\Gamma \vdash M : A$. Define a family of terms N_x indexed by $x \in \mathsf{dom}(\Gamma)$ by $N_x = x$. It follows from Lemma 12.1 that $N_x \in \mathsf{R}(\Gamma(x))$ for each such x. Consequently, by Theorem 12.1, $M[N_x/x \mid x \in \mathsf{dom}(\Gamma)] \in \mathsf{R}(A)$, but this term is obviously just M. Then, it follows from 12.1 that M is SN. □

13. Curry-Howard

In this lecture, we see the incredible connection of typed λ-calculi and logic.

13.1 Inhabitation

I want to start by considering a problem that is, in some sense, dual to the problem of finding the type of a term. The "elements" of a type (thinking of a type as something like a set of terms) are more usually called the *inhabitants*.

Definition 13.1. *The **inhabitation problem**, is the problem of, given a type A, determining if there a closed term M such that* $\vdash M : A$.

Let's look at some examples. Which of the following types are inhabited and which are not?

(T1) $a \to a$

(T2) $(a \to a \to b) \to a \to b$

(T3) $((a \to a) \to b) \to b$

(T4) $a \to b$ (with $a \neq b$)

(T5) $(a \to b) \to b$

The first three are inhabited, example inhabitants include:

$$\vdash \lambda x.\, x : a \to a$$
$$\vdash \lambda xy.\, xyy : (a \to a \to b) \to a \to b$$
$$\vdash \lambda x.\, x(\lambda y.\, y) : ((a \to a) \to b) \to b$$

The final two types, however, are not inhabited. We can argue as follows.

Lemma 13.1. *Types $a \to b$ (with $a \neq b$) and $(a \to b) \to b$ are not inhabited.*

Proof. We only prove the first, because the second is similar. Assume $a \neq b$. Suppose for contradiction that $a \to b$ is inhabited. Then there is a closed term M such that $\vdash M : a \to b$. By Strong Normalisation, there is a normal form N of M and, by Subject Reduction, we know that $\vdash N : a \to b$. Since N is also closed (e.g. the type environment is empty), N cannot contain a free variable and since N is in normal form, N cannot contain a redex. Therefore, it can only be that N is an abstraction $\lambda x.\, P$. Therefore, by Inversion, it must be that $x : a \vdash P : b$. Now consider by cases the structure of P:

- If P is a variable, it must be x, but this contradicts inversion, which would require $a = b$.
- If P is an abstraction $\lambda y.Q$, but this contradicts Inversion which would require b to be an arrow.
- If P is an application, $Q_1 Q_2$ then, by Inversion and the fact that Q_1 must be in normal form, we would have that there is a unique B such that $x : a \vdash Q_1 : B \to b$, but this contradicts the Subformula Property.

□

Now I would like you to consider the following formulas of propositional logic. Which of these formulas are valid, i.e. true for all possible instantiations of a and b?

(F1) $a \Rightarrow a$

(F2) $(a \Rightarrow a \Rightarrow b) \Rightarrow a \Rightarrow b$

(F3) $((a \Rightarrow a) \Rightarrow b) \Rightarrow b$

(F4) $a \Rightarrow b$ (with $a \neq b$)

(F5) $(a \Rightarrow b) \Rightarrow b$

Obviously a implies a is true, no matter what a is. Formulas (F2) and (F3) are also valid and can be argued as follows.

Lemma 13.2. *Formulas* $(a \Rightarrow a \Rightarrow b) \Rightarrow a \Rightarrow b$ *and* $((a \Rightarrow a) \Rightarrow b) \Rightarrow b$ *are valid.*

Proof. For the first, assume $a \Rightarrow a \Rightarrow b$, call it (x) and then assume a (y). Applying (x) to (y) we have $a \Rightarrow b$ and applying this to (y) we obtain the truth of b.

For the second, assume $((a \Rightarrow a) \Rightarrow b)$, call it (x). We have $a \Rightarrow a$ already because if we assume a, then clearly a follows. Therefore, using (x) we can obtain b, as required. □

On the other hand, the final two formulas are not valid. For example, $a \Rightarrow b$ is not true for $a := \top$ and $b := \bot$. Likewise, $(a \Rightarrow b) \Rightarrow b$ is not true for $a := \bot$ and $b := \top$.

If you look at the syntax, the only difference between (T1)–(T5) and (F1)–(F5) is the way I wrote the arrow, single-bodied in the former and double-bodied in the latter. Moreover, the types that were inhabited turned out to correspond to the formulas that were valid. Is this a coincidence?

Let's consider *why* we know that (T1)–(T3) are inhabited and *why* we know that (F1)–(F3) are valid: what is the evidence we are using to justify our conclusions in each case. For each type A, the evidence is that there is a closed term M and a type derivation for $\vdash M : A$. For each formula A, the evidence is that there is a proof of A. Let's put these side-by-side in the case of (T2) and (F2), using Γ as a short hand for $\{x : a \to a \to b,\ y : a\}$:

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : a \to a \to b} \quad \dfrac{}{\Gamma \vdash y : a}}{\Gamma \vdash xy : a \to b} \quad \dfrac{}{\Gamma \vdash y : a}}{\Gamma \vdash xyy : b}}{x : a \to a \to b \vdash \lambda y.xyy : a \to b}}{\vdash \lambda xy.xyy : (a \to a \to b) \to a \to b}$$

Proof. Assume $a \Rightarrow a \Rightarrow b$, call it (x), and then assume a (y). Applying (x) to (y) we have $a \Rightarrow b$ and applying this to (y) we obtain b. □

If try to reconcile the two sides of this example, you will soon realise that the backwards rule for implication in the proof corresponds quite exactly to the (TAbs) rule. The (TAbs) rule says, reading bottom to top, that to show that a term $\lambda x.M$ has type $A \to B$ under hypotheses Γ, you should assume that x has type A and go on to try to type M with B. The backwards rule for implication also says that, to prove $A \Rightarrow B$, you should assume A and try to prove B.

Similarly, the forwards rule for implication corresponds to the (TApp) rule. The (TApp) rule says, reading *top to bottom*, that if I already know that M has type $A \to B$ under some hypotheses Γ and I already know that N has type A under some hypotheses Γ, then I can conclude that MN has type B under the same hypotheses. The forwards rule for implication says as much: if I know $B \Rightarrow A$ and I know A, then I know B.

The (TVar) just corresponds to where we are recalling an assumption that we already made by using the name we gave it at the time (in this case, x or y).

Take a look at (T3) vs (F3), this time letting $\Gamma = \{x : (a \to a) \to b\}$:

$$\dfrac{\dfrac{\dfrac{}{\Gamma \vdash x : (a \to a) \to b} \quad \dfrac{\dfrac{}{\Gamma, y : a \vdash y : a}}{\Gamma \vdash \lambda y.y : a \to a}}{\Gamma \vdash x(\lambda y.y) : b}}{\vdash \lambda x.x(\lambda y.y) : ((a \to a) \to b) \to b}$$

Proof. Assume $((a \Rightarrow a) \Rightarrow b)$, call it (x). We have $a \Rightarrow a$ already because if we assume a, then clearly a follows. Therefore, using (x) we can obtain b, as required. □

Try to match up the backwards reasoning with uses of the (TAbs) rule and forwards reasoning with uses of the (TApp) rule.

13.2 C-H for the implicational fragment

I deliberately named all the assumptions in the forgoing proofs and I deliberately chose names that came from our set of term variables e.g. $x, y \in \mathbb{V}$. Let us say that a proof is *properly labelled* if all

starting assumptions and those introduced by backwards rules are given a name from $\mathbb{V}$ and names are given sensibly – i.e. we don't give the same name to two different formulas in the same proof. Obviously, it is not a real restriction to require proofs to be properly labelled, given a proof I can always do the labelling afterwards, but it is necessary to state a strong correspondence:

Theorem 13.1 (Curry-Howard)**.** *If we consider properly labelled proofs in the implicational fragment of propositional logic (i.e. the only connective is implication), then the following is true:*

- *From a constructive proof of A from starting assumptions* Γ*, one can extract a term M such that* $\Gamma \vdash M : A$*.*
- *From each term M such that* $\Gamma \vdash M : A$*, we can extract a constructive proof that A follows from assumptions* Γ*.*

(This requires we accept that $\to$ *is an equally good way to write implication.)*

We will come to the *constructive* requirement in a moment. First, let's consider another example. Suppose you give me a proof of the transitivity of implication:

$$(a \Rightarrow b) \Rightarrow (b \Rightarrow c) \Rightarrow a \Rightarrow c$$

Proof. Suppose $a \Rightarrow b$ (x) and $b \Rightarrow c$ (y). Then assume a (z). It follows from (x) and (z) that b holds, and it follows from that and (y) that c holds, as required. □

I can translate each use of the backwards rule for implication into a use of (TAbs), each use of the forwards rule for implication into a use of (TApp) and each recall of a labelled assumption into (TVar). This gives me a skeleton of a type derivation, in which I know which rules are used to conclude which types, but we don't yet have any of the terms in the interior:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash y : b \to c}\,(\text{TVar}) \quad \dfrac{\dfrac{}{\Gamma \vdash x : a \to b}\,(\text{TVar}) \quad \dfrac{}{\Gamma \vdash z : a}\,(\text{TVar})}{\Gamma \vdash ? : b}\,(\text{TApp})}{\Gamma \vdash ? : c}\,(\text{TApp})}{x : a \to b, y : b \to c \vdash ? : a \to c}\,(\text{TAbs})}{x : a \to b \vdash ? : (b \to c) \to a \to c}\,(\text{TAbs})}{\vdash ? : (a \to b) \to (b \to c) \to a \to c}\,(\text{TAbs})$$

But, if I know all the types and which rules were used, then the terms are completely determined, I can just fill them in starting from the leaves and working my way down:

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{\Gamma \vdash y : b \to c}\,(\text{TVar}) \quad \dfrac{\dfrac{}{\Gamma \vdash x : a \to b}\,(\text{TVar}) \quad \dfrac{}{\Gamma \vdash z : a}\,(\text{TVar})}{\Gamma \vdash xz : b}\,(\text{TApp})}{\Gamma \vdash y(xz) : c}\,(\text{TApp})}{x : a \to b, y : b \to c \vdash \lambda z.\, y(xz) : a \to c}\,(\text{TAbs})}{x : a \to b \vdash \lambda yz.\, y(xz)(b \to c) \to a \to c}\,(\text{TAbs})}{\vdash \lambda xyz.\, y(xz) : (a \to b) \to (b \to c) \to a \to c}\,(\text{TAbs})$$

The term that I obtain in this way is sometimes called the *proof term* of the formula. In this case, the proof term for the transitivity of implication is none other than the program that implements function composition:

$$N \circ M \coloneqq (\lambda x\, y\, z.\, y(xz))\, N\, M$$

There is a sense in which, if it is in normal form, a term of type A *is* a proof of A. For example, let's revisit (T2). The term here is $\lambda x.\, x(\lambda y.\, y)$ and the type is $(a \to a) \to b \to b$. Since this term is in normal form, the judgement $\vdash \lambda x.\, x(\lambda y.\, y) : (a \to a) \to b \to b$ completely determines the derivation tree:

$$\dfrac{\dfrac{}{\Gamma \vdash x : (a \to a) \to b} \quad \dfrac{\dfrac{}{\Gamma,\, y : a \vdash y : a}}{\Gamma \vdash \lambda y.\, y : a \to a}}{\dfrac{\Gamma \vdash x(\lambda y.\, y) : b}{\vdash \lambda x.\, x(\lambda y.\, y) : ((a \to a) \to b) \to b}}$$

There are no choices that can be made in how we construct this tree. But then the derivation tree determines a proof, by mapping each use of (TVar) to the recall of an assumption, each use of (TApp) to the forwards rule for implication and each use of (TAbs) to the backwards rule for implication. We obtain:

Proof. Assume $(a \Rightarrow a) \Rightarrow b$, call it (x). We have $a \Rightarrow a$ by the following reasoning: assume a and call it (y), then a follows immediately from (y). From $a \Rightarrow a$ and (x) we obtain b. □

Notice how the subterm $\lambda y.\, y$ corresponds to a subproof of $a \Rightarrow a$. In a sense, $\lambda y.\, y$ *is* a proof of $a \Rightarrow a$.

Of course, the implicational fragment of propositional logic is not very interesting: we can hardly carry out many proofs in that system. Remarkably, however, the Curry-Howard correspondence carries over from this tiny λ-calculus and implicational logic, all the way up to first- and even higher-order logic, i.e. all the propositional connectives and all the quantifiers. In other words, all (constructive) mathematics corresponds to programming up λ-terms in such a way that they have the right type (of course, just as the first-order logic is larger than and more complicated than its implicational fragment, so the corresponding λ-calculi are larger and more complicated too). This is the so-called "proofs as programs" paradigm, which has given rise to new kinds of programming languages and new kinds of logic.

Types	$\leftrightarrow$	Formulas
Programs	$\leftrightarrow$	Proofs
Inhabitation	$\leftrightarrow$	Provability

14. Intuitionism

14.1 Proof from Truth

The idea of proof is that, if we have a valid proof of a proposition A, then we should be able to conclude that A is *true*. Do you believe in this? Do you believe in your proofs? Why should you believe that if you can write one of these little paragraphs of text with a scattering of mathematical formulas that therefore A is actually true? In other words, why should you believe that the rules of proof are *sound*?

Classically, we can justify the soundness of the proof rules by appealing to an independent definition of when formulas are *true*. We start from the assumption that every atomic proposition is either true or false, and then the truth of compound formulas is determined by the truth of their parts. For example, truth tables define the truth of the logical connectives in terms of their arguments. For universal quantification, $\forall x \in \mathbb{N}.\ A$ is true when $A[n/x]$ is true for all choices of natural number n. Such a definition of truth ensures that every closed formula, i.e. *sentence*, is either true or false under a given interpretation of the non-logical parts (like numbers). However, given any particular proposition C, we may not be able to tell from the definition which it is, because it may require verifying that *infinitely many* other formulas (e.g. the $A[n/x]$ when $C = \forall x \in \mathbb{N}.\ A$) are themselves true, and this is best attempted only if you are a deity.

According to this view, proof rules are a system that mere mortals can use to try to get a handle on the divine notion of *truth*. Human beings can write down proofs, which are finite objects and, as a result, conclude the truth of formulas — even universal quantifications which seem to require an infinite amount of checking.

> A proof of A from starting assumptions Γ is a certificate guaranteeing that A is true whenever Γ is true.

This view of proof as being a syntactical certificate of truth allows us to justify why this particular collection of rules for deriving certain combinations of strings from other combinations of strings makes sense as a logic. For example, under this view, the forwards proof rule for implication is sound. This rule says, roughly: "if I have proven $A \Rightarrow B$ and I have proven A, then I have proven B". If we view "proven" as a way of saying "certified to be true using finite, mortal means", then indeed, if I have certified $A \Rightarrow B$ to be true and I have certified A to be true, then by the definition of truth for

$A \Rightarrow B$, i.e.

A	B	$A \Rightarrow B$
$\bot$	$\bot$	$\top$
$\bot$	$\top$	$\top$
$\top$	$\bot$	$\bot$
$\top$	$\top$	$\top$

it can only be that B is true (i.e. we are in the last row). Similarly, the backwards proof rule for implication is sound. This rule says, roughly "to prove $A \Rightarrow B$, it suffices to assume that A is true and try to prove B". Indeed, if I want to certify that $A \Rightarrow B$ is true, then by the definition of truth, then it suffices to assume that A is true and try to certify that B is true as a consequence. This is because, if A is false, then $A \Rightarrow B$ is true by definition, so there is nothing to check.

14.2 Truth from Proof

Not all logicians are in agreement about this idea of truth at the foundation of logic and that proof is something we invent afterwards as a mortal means to discover it. For example, a problem with the foregoing account is that a proposition can be true even when we don't have a certificate of it. We can even know that some proposition A is true without *ever* proving it *directly*, for example, if we merely have a certificate that $\neg A$ is false.

By and large, *intuitionists* believe that mathematics is sum total of the mental constructions that mathematicians devise and communicate to each other — there is no ideal or platonic truth behind the scenes that we just trying to get a glimpse of. In particular, intuitionists regard a proposition as true only if there is some concrete *evidence*. This gives an alternative interpretation of the proof, which does not pressupose a definition of truth.

> A proof of A from starting assumptions Γ is a method for *constructing* evidence of A from evidence of Γ.

According to this view, proof is not about making arguments to certify truth, but instead about manipulating evidence. The idea is credited to Brouwer and Heyting and, independently, Kolmogorov. We start by explaining what kinds of evidence is needed in order to believe in any formula.

Each atomic proposition has its own notion of evidence, for example, the reading on a thermometer might be evidence for a basic proposition "the temperature is 23°C". A proof tree using the rules (Var), (Abs) and (App) is evidence for the basic proposition $\lambda x\, y.\, y\, x \in \Lambda$.

We can understand more complex formulas in terms of the evidence required for their components.

- There can be no evidence for the truth of false.
- No evidence is needed for the (self-evident) truth of true.
- Evidence for $A \wedge B$ is a pair consisting of evidence for A and evidence for B.
- Evidence for $A \vee B$ is either a piece of evidence of A or a piece of evidence of B.
- Evidence for $A \Rightarrow B$ is a *procedure* for transforming evidence of A into evidence of B.

- Evidence of $\neg A$ is a procedure for transforming any evidence of A into evidence of false.
- Evidence of $\forall x \in X . A$ is a procedure for transforming any y and evidence of $y \in X$ into evidence of $A[y/x]$.
- Evidence of $\exists x \in X . A$ is a triple consisting of a y, evidence of $y \in X$ and evidence of $A[y/x]$.

A crucial criterion for intuitionists is that evidence of an existential statement actually constructs a witness. Indeed, such logics are sometimes called *constructive* because, in general, the evidence can be seen as some construction that witnesses the truth of the formula.

This gives us an alternative way to justify our proof rules. We can justify the forwards rule for implication as follows. If we have a proof of $A \Rightarrow B$ we interpret this to mean that we have, in our hand, a procedure for transforming evidence of A to evidence of B. So, if we also have evidence of A, we can apply our procedure to it in order to obtain some evidence of B.

These can be a bit of a mouthful to write this out in English, but luckily it can be precisely summarised by the rule:

$$\frac{\Gamma \vdash M : B \to A \quad \Gamma \vdash N : B}{\Gamma \vdash MN : A} \ (\mathsf{TApp})$$

Read, from top to bottom: if I have evidence M of $B \to A$ and I have evidence N of B, then I can construct evidence of A by applying M to N, i.e. the evidence of A is MN.

Similarly, we can justify the backwards rule for implication as follows. If we have a proof of B starting from assumptions Γ including A, then this means that we have a method that can construct evidence for B from evidence for each of the propositions in Γ and evidence for A. Therefore, given evidence of each of the propositions in Γ, we have a way for converting any piece of evidence of the remaining proposition A into evidence for the proposition A, i.e. we simply supply that new piece of evidence along with our evidence for Γ to the method to obtain evidence of B. Hence we can package this up as a procedure evidencing $A \Rightarrow B$.

This is neatly summarised by the following rule:

$$x \notin \mathsf{dom}\ \Gamma \quad \frac{\Gamma \cup \{x : A\} \vdash M : B}{\Gamma \vdash \lambda x . M : A \to B} \ (\mathsf{TAbs})$$

This rule reads: if I can construct evidence of B from evidence of Γ and some evidence x of A, then from evidence of Γ only, I can construct a function that can already take any such x and return evidence of B. In other words, $\lambda x . M$ *is* the evidence by which we can believe that $A \to B$ is true.

We can justify e.g. the backwards rule for forall too. If we have a proof of A starting from assumptions Γ and that proof makes no assumption on say $x \in \mathbb{N}$, then, effectively, I have a procedure for converting arbitrary natural numbers n into evidence of $A[n/x]$: if you give me such a number, say $6 \in \mathbb{N}$, then I can substitute it everywhere into my proof of A to obtain a proof of $A[6/x]$, i.e. I have produced a new proof that is specific to the number 6.

Our type system is too simple to capture universal quantification on individuals, but more advanced type systems, such as the type system for the *Agda* programming language does so, and the terms of type $\forall x . A$ are functions.

The justification of the other *constructive* proof rules is similar. The only proof rules that do not fall into this category are the non-constructive ones.

In the classical truth-theoretic reading of logic, every proposition is either true or false, although it is quite often the case that for any particular proposition, we mortals don't know which one it happens to be. A consequence of this is that, if I know that something is not false, it must be true. So, in this truth-theoretic reading, *reductio ad absurdum* (a.k.a the double negation rule) is justified: if I know that $\neg A$ is false, A must be true — in other words, to prove A it suffices to prove $\neg\neg A$.

This is quite different from the evidence-based approach to logic. If there can be no evidence for $\neg A$, does that mean that I have evidence for A? For example, to have evidence of a disjunction $(\lambda x.\, x \in \Lambda) \vee (\vdash \mathsf{Succ} : \mathsf{Bool})$ is to actually hold evidence for one of the disjuncts, i.e. a proof tree for $\lambda x.\, x \in \Lambda$ or a typing derivation for $\vdash \mathsf{Succ} : \mathsf{Bool}$. If I have evidence for $\neg(\neg((\lambda x.\, x \in \Lambda) \vee (\vdash \mathsf{Succ} : \mathsf{Bool})))$ then I am holding a procedure that *would* transform any evidence of $\neg((\lambda x.\, x \in \Lambda) \vee (\vdash \mathsf{Succ} : \mathsf{Bool}))$ into evidence of false. That is to say, given a procedure that transforms evidence of one of the disjuncts into evidence of false as an input, I have in my hand a procedure that will produce evidence of false as output. Having such a procedure is clearly not the same as actually holding evidence of one of the disjuncts, which is a proof tree. In particular, if I had evidence of one of the disjuncts then I would know that particular disjunct was true, but if I merely have a procedure for converting evidence of their negation into evidence of false, then I can't tell you anything about which of the disjuncts is actually true, which is the whole basis for believing that a disjunction is true in the first place.

Therefore, the double negation rule is rejected by intuitionists. They don't consider proofs that use this rule to be convincing.