

Handout 11

More on the Lambda Calculus

1. Higher-order functions. The λ -calculus is a purely syntactic device; it does not make any distinctions between simple entities, such as numbers and more complicated ones, such as functions of functions. Whatever can be described as a λ -term is available for manipulation by other λ -terms.

Let us look at an example. A term for squaring integers is given by

$$Q \stackrel{\text{def}}{=} \lambda x. * x x$$

If we want to compute x^8 then this can be achieved by squaring x three times: $x^8 = ((x^2)^2)^2$. In λ -calculus notation, we would write for the “power-8”-function:

$$P_8 \stackrel{\text{def}}{=} \lambda x. Q (Q (Q x))$$

We see that taking a number to power 8 amounts to applying the squaring function Q three times. It is now a simple step to write out a λ -term which applies *any function* three times:

$$T \stackrel{\text{def}}{=} \lambda f. (\lambda x. f (f (f x)))$$

(Observe the — unnecessary — brackets around the inner function; I wanted to stress that T takes as argument a function f and returns another function with argument x .) The term P_8 can now be written as $T Q$, and 5^8 comes out as $T Q 5$.

There is nothing to stop us from applying the tripling operator T to itself, $T T$. What we get is an operator which will triple any function we pass to it three times, so it is in fact a 27-fold operator, that is, $T T f x$ will compute the result of applying f 27 times to x .

Operators such as T are called **higher order** because they operate on functions rather than numbers.

2. Iteration and recursion in the λ -calculus. As we have seen with the terms T and $T T$, a short combination of λ -terms can express repeated application of a function. How can we generalise this to get the behaviour of a `for`-loop, where the number of repetitions is controlled by a counter? This requires a wholly new idea which we will now develop step by step. First of all, we have to use a constant which allows us to distinguish between 0 and positive numbers. Let us call this constant “zero?”. Its behaviour is like an `if-then-else` clause depending on the value of a number:

$$\begin{aligned} \text{zero? } 0 x y &\longrightarrow x \\ \text{zero? } n x y &\longrightarrow y \quad (n \neq 0) \end{aligned}$$

In Java, we would write this as

$$(n==0) ? x : y$$

We also assume constants “pred” and “succ” for predecessor and successor functions on natural numbers.¹ `succ` n is $n + 1$. `pred` n is $n - 1$ if $n > 0$; `pred` 0 is 0.

Let us now construct a term I (for “Iteration”) which takes as arguments a number n , a function f , and a value x , and computes the n -fold application of f to x :

$$I n f x = f (f (f \dots (f x) \dots))$$

Mathematically, we might write this as $f^n(x)$, so $I n f$ has the effect of f^n – the function got by applying f , n times in succession. If $n = 0$ then $I 0 f x$ should simply return x , without applying f at all – $I 0 f$ is just the identity function. Here is a first attempt at defining I :

$$I = \lambda n f x. \text{zero? } n x (I (\text{pred } n) f (f x))$$

Here is the rationale: If $n = 0$ then `zero? $n x M$` will evaluate to x , no matter what M is. If $n > 0$ then we apply f^{n-1} to the argument $(f x)$; if successful, this will return f^n applied to x .

There is only one snag; our definition of I uses I itself in the body (which is why I left out the “def” from the equality symbol). It follows the usual idea of recursion: “I can do it n times if I have subcontractors who can do it $(n - 1)$ times for me...”. In other words, the term as written above does nothing else but add one further iteration to an assumed $(n - 1)$ -fold iteration.

How can we overcome the circularity? Have a look at the definition again:

$$I = \lambda n f x. \text{zero? } n x (I (\text{pred } n) f (f x))$$

¹The natural numbers are the non-negative integers 0, 1, 2, Sometimes you see a definition where the natural numbers start at 1, but for us, as for most computer scientists, the natural numbers will always include 0.

Another way of reading this is to say that I (if it ever can be found) would be a fixpoint of the term on the right. Let's make this view more explicit. We change the term on the right into a function which turns “ $(n - 1)$ -iterators” into “ n -iterators”:

$$M \stackrel{\text{def}}{=} \lambda I'. (\lambda n f x. \text{zero? } n \ x \ (I' \ (\text{pred } n) \ f \ (f x)))$$

(I have changed the I there into an I' , to avoid any name clash between the formal parameter I' and the I I am trying to define.) This definition is no longer circular, so M is a proper term. What we now seek is a term I which satisfies

$$I = M I$$

that is, a term which is a **fixpoint** for M .

Amazingly, such a fixpoint can always be found. In fact, there are terms Y which construct a fixpoint for *any* term M , that is, they satisfy

$$Y M = M (Y M)$$

Once we have such a Y , we have solved our iterator problem because we can set $I \stackrel{\text{def}}{=} Y M$ for our M as defined above. We call such a Y a **fixpoint combinator**. Here is Turing's fixpoint combinator:

$$Y \stackrel{\text{def}}{=} (\lambda x y. y(xxy))(\lambda x y. y(xxy))$$

Let us write $T \stackrel{\text{def}}{=} \lambda x y. y(xxy)$, so that $Y \stackrel{\text{def}}{=} T T$. We then see that $Y M$ reduces to $M (Y M)$:

$$\begin{aligned} Y M &= T T M \\ &= (\lambda x y. y(xxy)) T M \\ &\rightarrow_{\beta} M(T T M) \\ &= M(Y M) \end{aligned}$$

Fixpoints are also used to create `while`-loop like behaviour. Consider, for example, the problem of finding the smallest number that is at least n and for which a given function f returns zero. We implement this as a fixpoint equation as follows:

$$Z = \lambda f n. \text{zero? } (f n) \ n \ (Z f \ (\text{succ } n))$$

(“If $f(n)$ yields 0 return n , else continue the search at $n + 1$.”) Transform this into a function in the unknown Z' :

$$L \stackrel{\text{def}}{=} \lambda Z' f n. \text{zero? } (f n) \ n \ (Z' f \ (\text{succ } n))$$

and the desired root-finder comes out as

$$Z \stackrel{\text{def}}{=} Y L$$

The smallest root of a function f (if there is one at all) is calculated by $Y L f 0$.

3. The λ -calculus as a model of computation. There are a number of variants of the λ -calculus which one can consider for a comparison with Turing machines. For this purpose, we call a calculus **Turing-complete** if it allows one to define all *computable* functions from \mathbb{N} (the set of natural numbers) to \mathbb{N} . In order to avoid pathological calculi, we have to require also that calculations in the calculus can be performed effectively (for example, by a machine). This latter requirement is no problem for the λ -calculus; the operation of β -reduction is well-defined and can be performed by a computer program. To get to Turing completeness, the first step is obviously to decide how natural numbers are to be represented in the λ -calculus. One idea, which we have been implicitly using all along so far, is to use constants 0, 1, 2, ... for the natural numbers and special symbols for arithmetic operators. We also need ways to write algorithmic constructs, and we have seen that special symbol `zero?` gives us conditionals and the fixpoint combinator Y (which is a pure λ -term) then gives us recursion and iteration. Once we have that ability to construct algorithms, it turns out that the only operators we need are successor and predecessor – all other computable operators on \mathbb{N} can be constructed algorithmically from them. In short –

Theorem 1 *The λ -calculus enriched with `zero?`, `pred`, `succ` and constants for all numbers is Turing-complete.*

Surprisingly, we can say the same about the *pure* λ -calculus, without any constants at all. In order for this to make sense, one has to agree on a representation of natural numbers as certain λ -terms. There are several possibilities for this; we shall look at the *Church numerals*. Recall the iterator I defined above, with $In f$ having the effect of f^n . In that definition, it was assumed that we had natural numbers and arithmetic defined using constants: n would be one of the natural number constants, and I was defined using `pred` and `zero` (as well as the fixpoint combinator). The idea of Church numerals is that there are *pure λ -terms* that have the same effect as the iterator terms In .

$$\begin{aligned} 0 &\stackrel{\text{def}}{=} \lambda f x. x \\ 1 &\stackrel{\text{def}}{=} \lambda f x. f x \\ n &\stackrel{\text{def}}{=} \lambda f x. f(f \dots (f x) \dots) \quad (n\text{-fold application of } f \text{ to } x). \end{aligned}$$

Thus with a Church numeral n , $n f$ has the effect of f^n . With this representation in mind, the following is true:

Theorem 2 *The pure λ -calculus is Turing-complete.*

An important part of this is to define pure λ -terms $\text{zero}?$, pred and succ that have the desired effect when applied to Church numerals.

For successor, we use the fact that $f^{n+1}(x) = f(f^n(x))$:

$$\text{succ} \stackrel{\text{def}}{=} \lambda n f x. f(n f x)$$

Now any Church numeral can be got by applying succ enough times to 0. In fact, for any Church numeral n we find $n \text{ succ } 0$ reduces to n . The trick² is to note that if it works for n then it also works for $\text{succ } n$: for

$$\begin{aligned} \text{succ } n \text{ succ } 0 &= (\lambda n f x. f(n f x)) n \text{ succ } 0 \\ &\longrightarrow_{\beta} \text{succ } (n \text{ succ } 0) \\ &\longrightarrow_{\beta} \text{succ } n. \end{aligned}$$

Obviously it works for 0,³ for

$$0 \text{ succ } 0 = (\lambda f x. x) \text{ succ } 0 \longrightarrow_{\beta} 0$$

and so, repeating, we see it works for all numerals n .

For $\text{zero}?$, we use a function $\lambda u. y$ that always returns the same value y . This function has the property that if you apply it one or more times to an actual parameter x then the result is y , but, as for any function, if you apply it zero times to x then the result is just x .

$$\text{zero?} \stackrel{\text{def}}{=} \lambda n x y. n (\lambda u. y) x$$

Defining pred is harder.

$$\text{pred} \stackrel{\text{def}}{=} \lambda n f x. n (\lambda g h. h(g f)) (\lambda u. x) (\lambda u. u)$$

Straightaway we get

$$\begin{aligned} \text{pred } 0 &\longrightarrow_{\beta} \lambda f x. 0 (\lambda g h. h(g f)) (\lambda u. x) (\lambda u. u) \\ &\longrightarrow_{\beta} \lambda f x. (\lambda u. x) (\lambda u. u) \\ &\longrightarrow_{\beta} \lambda f x. x \\ &= 0 \end{aligned}$$

For higher numerals $n > 0$, let us write T for $\lambda g h. h(g f)$. Then it can be proved that – using informal notation –

$$T^n(\lambda u. x) = \lambda h. h(f^{n-1}(x))$$

so

$$T^n(\lambda u. x)(\lambda u. u) = (\lambda u. u)(f^{n-1}(x)) = f^{n-1}(x)$$

and so $\text{pred } n f$ acts as f^{n-1} .

4. Banning bad terms with types. As the previous section showed, there is good use in the λ -calculus for slightly strange terms without normal form such as

$$Y \stackrel{\text{def}}{=} (\lambda x y. y(x x y))(\lambda x y. y(x x y))$$

However, there is nothing in the grammar which stops us from forming truly awful terms, such as “sin log”, where the sine function is applied not to a number but to the logarithm function. Such terms do not make any sense at all, and any sensible programming language compiler would reject them as ill-formed. What is missing in the calculus is a notion of **type**. The type of a term should tell us what kind of arguments the term would accept and what kind of result it will produce. For example, the type of the sine function should be “accepts real numbers and produces real numbers”.

A language for expressing these properties (i.e., types) is easily defined. We start with some base types such as “nat”, “int” or “real” for natural numbers, integers or real numbers, and then form **function types** on top of them. The grammar for this idea is extremely simple (which is why it is called the **system of simple types**):

$$\tau ::= c \mid \tau \rightarrow \tau$$

The placeholder c represents all the base types we might wish to include. Apart from this, all one can do is form a function type from given types. $\sigma \rightarrow \tau$ is the type for functions for which the parameter is of type σ and the result of type τ .

Note that the grammar is ambiguous because it doesn’t include brackets. In practice, we shall need to add brackets. However there is a useful bracketing convention that $\sigma \rightarrow \tau \rightarrow \nu$ is implicitly bracketed as $\sigma \rightarrow (\tau \rightarrow \nu)$. Think back to the way a function with two parameters, of types σ and τ , would be represented in λ -calculus application: it takes one parameter, of type σ , and delivers a result that is another function, which takes parameter of type τ and gives a result of

²induction

³base case

type v . This makes it of type $\sigma \rightarrow (\tau \rightarrow v)$. Hence the bracketing convention for $\sigma \rightarrow \tau \rightarrow v$ gives the type that is natural for functions with two parameters. In general, if you see a type without any brackets, such as

$$\sigma_1 \rightarrow \dots \rightarrow \sigma_n \rightarrow \tau,$$

then you can think of it as the type for functions with n parameters, of types σ_i , and a result of type τ .

It is for *higher order* functions that the brackets become essential. For example, $(\sigma \rightarrow \tau) \rightarrow v$ is the type of a function whose parameter is itself a function.

With such a system, the type of the sine function can be denoted by “real \rightarrow real” and it is obvious that it cannot accept the logarithm function as an argument because the latter also has type “real \rightarrow real” and not “real” as required.

On the basis of a type system such as the simple one exhibited here, we can formulate restrictions on what kind of terms are valid (or **well-typed**). We do so by employing an inductive definition that starts by annotating variables with types.

Definition 3 (Well-typed λ -terms)

Base case. For every type σ and every variable x , the term $x:\sigma$ is well-typed and has type σ .

Function formation. For every term M of type τ , every variable x , and every type σ , the term $\lambda x:\sigma. M$ is well-typed and has type $\sigma \rightarrow \tau$.

Application. If M is well-typed of type $\sigma \rightarrow \tau$ and N is well-typed of type σ then MN is well-typed and has type τ .

(Convention: Within a single term we will not use the same variable name with two different type annotations.)

Some examples:

- $\lambda x:\sigma. x:\sigma$ is well-typed of type $\sigma \rightarrow \sigma$ no matter what σ stands for.
- The term $\lambda x:\sigma. \lambda y:\tau. x:\sigma$ is well-typed of type $\sigma \rightarrow (\tau \rightarrow \sigma)$, i.e. $\sigma \rightarrow \tau \rightarrow \sigma$.
- The term $\sin \log$ is not well-typed.

Furthermore, any term of the shape $M M$ cannot be annotated with simple types (Exercise 4).

5. Calculating simple types. It is quite easy to find out whether a term can be typed or not by following the steps in which the term was constructed. What we do is to annotate subterms with type expressions which still contain **type variables** A, B, C, \dots and which we refine as we go along. Consider, for example, the term $\lambda f x. f x$. $f x$: We give x the type A (a type variable) and give f the type B . Because the subterm $f x$ needs to be well typed according to the application rule in Definition 3, we refine B to the shape $A \rightarrow C$, with C another type variable. The application $f x$ is then possible and gets type C . The abstraction $\lambda x. f x$ is always possible, and because of our assumption about x , will have type $A \rightarrow C$. Likewise, for the abstraction $\lambda f. \lambda x. f x$ we remember that f should have type $A \rightarrow C$. According to the function formation rule, then, the complete term should have type $(A \rightarrow C) \rightarrow (A \rightarrow C)$. At this stage the type variables can be instantiated with something more concrete (such as “int” or “real”) but we only wanted to establish typability and so we can stop here. Further refinement is required if we extend the term to $(\lambda f x. f x) (\lambda y. y) 3$. Taken on its own, the subterm $\lambda y. y$ will have type $D \rightarrow D$, with D a fresh type variable. On the other hand, we have type $(A \rightarrow C) \rightarrow (A \rightarrow C)$ for $\lambda f x. f x$. In order for the application $(\lambda f x. f x) (\lambda y. y)$ to make sense, we must refine A to D and also C to D . The resulting type of $\lambda f x. f x$ is $(D \rightarrow D) \rightarrow (D \rightarrow D)$. Finally, 3 should have type “int” and in order for the last application to become well typed we refine D to “int”. The complete term then gets type “int” as well. If we spell out the types in the term we get:

$$(\lambda f:\text{int} \rightarrow \text{int}. \lambda x:\text{int}. f x) (\lambda y:\text{int}. y) 3$$

6. Regaining Turing completeness. Well-typed λ -terms are always well-behaved with respect to reduction:

Theorem 4 Every well-typed λ -term has a normal form.

Perhaps we have gone a bit too far now because it follows that the fixpoint combinator Y is not typable and hence does not belong to the **simply typed λ -calculus**. Because of its absence you can probably believe that the simply typed λ -calculus is *not* Turing-complete. In order to restore completeness, one has to explicitly enrich the calculus with fixpoint combinator *constants*. One such system is known under the name **PCF** (“programming computable functions”), introduced by Scott and Plotkin. It consists of λ -terms for a simple type system with base type “nat”, and the following constants:

Numerals. A constant \bar{n} of type **nat** for every natural number n .

Conditionals. Constants zero?_σ of type **nat** \rightarrow $\sigma \rightarrow \sigma \rightarrow \sigma$ for every type σ .

Successor function. Constants succ and pred of type **nat** \rightarrow **nat**.

Fixpoint combinators. Constants Y_σ of type $(\sigma \rightarrow \sigma) \rightarrow \sigma$ for every type σ .

We have:

Theorem 5 PCF is Turing-complete.

With PCF, then, we have a language which is expressive and well-typed at the same time. In fact, conceptually (and, would you believe it, historically), there is only a small step from PCF to the functional programming language ML.

Exercise Sheet 11

Quickies (I suggest that you try to do these *before* the Exercise Class.)

1. This question uses the definition of pred for Church numerals, and the definition $T \stackrel{\text{def}}{=} \lambda gh. h(gf)$ that went with it. Let us further define

$$S \stackrel{\text{def}}{=} \lambda mh. h(mfx).$$

- (a) Reduce $S\ n\ (\lambda u. u)$ to normal form.
(b) Show that $T(\lambda u. x)$ and $S0$ both reduce to $\lambda h. hx$.

Classroom exercises (Hand in to your tutor at the end of the exercise class.)

2. Suppose σ is a type and the variable x is given the type $x:\sigma$.

- (a) In $0 \stackrel{\text{def}}{=} \lambda f\ x:\sigma. x:\sigma$, what is the most general possible type of f , and then what is the type of 0 ?
(b) Similarly, in $1 \stackrel{\text{def}}{=} \lambda f\ x:\sigma. f\ x:\sigma$, what is the most general possible type of f , and then what is the type of 1 ?
(c) What is the most general way to give a type to f so that 0 and 1 have the same type? Show that then $2 \stackrel{\text{def}}{=} \lambda f\ x:\sigma. f(f\ x:\sigma)$ also has that type. 3

3. Define the addition operator $+$ as a λ -term

- (a) in the pure λ -calculus using numerals, and
(b) in PCF using succ and pred , and zero_σ and Y_σ at appropriate types. (To define $+$ $m\ n$ use recursion on m .) 1+2

Homework exercises

There are no homework exercises this week.

Practice questions

4. Argue that no term of shape $M\ M$ (application of a term to itself) can be typed with simple types.

Review questions

5. In which ways does Java's way of defining functions (i.e. methods) differ from that of the λ -calculus?
6. Why is the Church-Rosser Theorem important for the theory and applicability of the λ -calculus?
7. How are iteration and recursion implemented in the λ -calculus?
8. What is the "system of simple types"? What are the advantages of simply typed terms over untyped ones, and what are the drawbacks?