A short introduction to the Lambda Calculus

Achim Jung*

March 18, 2004

Abstract

The lambda calculus can appear arcane on first encounter. Viewed purely as a "naming device", however, it is a straighforward extension of ordinary mathematical notation. This is the point of view taken in these notes.

1. A brief history of mathematical notation. Our notation for **numbers** was introduced in the Western World in the Renaissance (around 1200) by people like Fibonacci. It is characterised by a small fixed set of digits, whose value varies with their position in a number. This place-value system was adopted from the Arabs who themselves credit the Indians. We don't know when and where in India it was invented.

A notation for **expressions** and **equations** was not available until the 17th century, when Francois Viète started to make systematic use of placeholders for parameters and abbreviations for the arithmetic operations. Until then, a simple expression such as $3x^2$ had to be described by spelling out the actual computations which are necessary to obtain $3x^2$ from a value for x.

It took another 250 years before Alonzo Church developed a notation for arbitrary **functions**. His notation is called λ-calculus ("lambda calculus"). Church introduced his formalism to give a functional foundation for Mathematics but in the end mathematicians preferred (axiomatic) set theory. The λ-calculus was re-discovered as a versatile tool in Computer Science by people like McCarthy, Strachey, Landin, and Scott in the 1960s.

Alonzo Church, 14.6.1903–11.8.1995

Incidentally, the history of programming languages mirrors that of mathematical notation, albeit in a time-condensed fashion: In the early days (1936-1950), computer engineers

*School of Computer Science, The University of Birmingham, Edgbaston, Birmingham, B15 2TT, A.Jung@cs.bham.ac.uk

struggled with number representation and tried many different schemes, before the modern standard of 2-complement for integers and floating point for reals was generally adopted. Viète's notation for expressions was the main innovation in FORTRAN, the world's first high-level programming language (Backus 1953), thus liberating the programmer from writing out tedious sequences of assembly instructions. Not too long after this, 1960, McCarthy came out with his list processing language Lisp. McCarthy knew of the λ-calculus, and his language closely resembles it.

Today, not many languages offer the powerful descriptive facilities of the λ-calculus, in particular, the mainstream languages Java and C++ make a strict distinction between primitive datatypes and objects, on the one hand, and functions (= methods), on the other hand. Likewise, the line of development started with Lisp, although it led to some truly remarkable languages such as ML and Haskell, has found it difficult to incorporate object oriented features. The OCaml dialect of ML is one of the few attempts to combine the two paradigms.

2. Expressions in the λ-calculus. The λ-calculus is a notation for functions. It is extremely economical but at first sight perhaps somewhat cryptic, which stems from its origins in mathematical logic. Expressions in the λ-calculus are written in strict *prefix* form, that is, there are no infix or postfix operators (such as $+, -, (\_)^2$, etc.). Furthermore, function and argument are simply written next to each other, without brackets around the argument. So where the mathematician and the computer programmer would write "$\sin(x)$", in the λ-calculus we simply write "$\sin\ x$". If a function takes more than one argument, then these are simply lined up after the function. Thus "$x + 3$" becomes "$+\ x\ 3$", and "x^2" becomes "$*\ x\ x$". Brackets are employed only to enforce a special grouping. For example, where we would normally write "$\sin(x) + 4$", the λ-calculus formulation is "$+\ (\sin\ x)\ 4$".

3. Functions in the λ-calculus. If an expression contains a variable — say, x — then one can form the function which obtains by considering the relationship between concrete values for x and the resulting value of the expression. In mathematics, function formation is sometimes written as an equation, $f(x) = 3x$, sometimes as a mapping $x \mapsto 3x$. In the λ-calculus a special notation is available which dispenses with the need to give a name to the function (as in $f(x) = 3x$) and which easily scales up to more complicated function definitions. In the given example we would re-write the expression "$3x$" into "$*\ 3\ x$" and then turn it into a function by preceding it with "$\lambda x.$". We get: "$\lambda x.*\ 3\ x$". The Greek letter λ ("lambda") has a role similar to the keyword "`function`" in some programming languages. It alerts the reader that the variable which follows is not part of an expression but the *formal parameter* of the function declaration. The dot after the formal parameter introduces the function body. Let's look more closely at the similarity with programming languages, say Pascal:

```
function  f(  x   :  int) :  int  begin  f :=  3 * x   end;
   |          |                      |           |
   λ          x                      .          * 3 x
```

You may be interested to see the same in Lisp:

```
(lambda (x) (* 3 x))
```

4. And on and on... A function which has been written in λ-notation can itself be used in an expression. For example, the application of the function from above to the value 4 is written as $(\lambda x.*\ 3\ x)\ 4$. Remember, application is simply juxtaposition; but why the brackets around the function? They are there to make clear where the definition of the function ends. If we wrote $\lambda x.*\ 3\ x\ 4$ then 4 would become part of the function body and we would get the function which assigns to x the value $3 * x * 4$ (assuming that $*$ is interpreted as a 3-ary function; otherwise the λ-term is nonsensical, see below.). So again,

brackets are used for delineating parts of a λ-term, they do not have an intrinsic meaning of their own.

Although it is not strictly necessary, it will be convenient to introduce abbreviations for λ-terms. We write them in the same way as we always to in mathematics, employing the equality symbol. So if we abbreviate our function term to F:

$$F \stackrel{\text{def}}{=\!=} \lambda x.* \; 3 \; x$$

then we can write $F \; 4$ instead of $(\lambda x.* \; 3 \; x) \; 4$.

With this our description of the λ-calculus as a notational device is almost complete; there is just one more case to consider. Suppose the body of a function consists of another function, as here

$$N \stackrel{\text{def}}{=\!=} \lambda y.(\lambda x.* \; y \; x)$$

If we apply this function to the value 3 then we get back our old friend $\lambda x.* \; 3 \; x$, in other words, N is a function, which when applied to a number, returns another function (i.e., $N \; 3$ behaves like F). However, we could also consider it as a function of *two* arguments, where we get a number back if we supply N with *two* numerical arguments ($N \; 3 \; 4$ should evaluate to 12). Both views are legitimate and perfectly consistent with each other. If we want to stress the first interpretation we may write the term with brackets as above, if we want to see it as a function of two arguments then we can leave out the brackets:

$$\lambda y.\lambda x.* \; y \; x$$

or, as we will lazily do sometimes, even elide the second lambda:

$$\lambda y \; x.* \; y \; x$$

but note that this is really just an abbreviation of the official term.

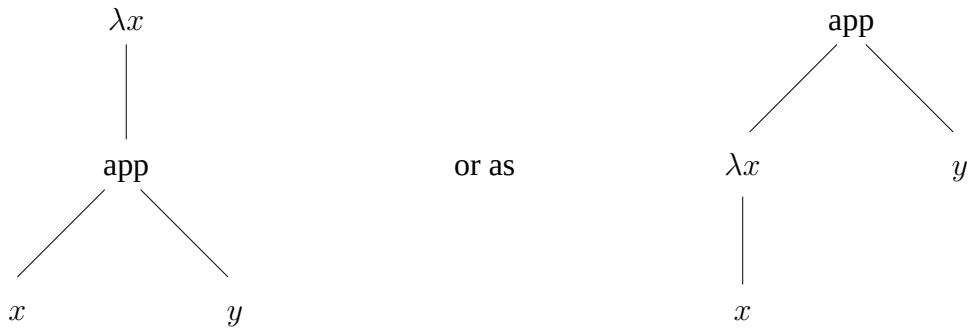
Likewise, in the application of N to arguments 3 and 4 we can use brackets to stress that 3 is to be used first: $(N \; 3) \; 4$ or we can suggest simultaneous application: $N \; 3 \; 4$. Whatever our intuition about N, the result will be the same (namely, 12).

5. The official definition. Function formation and function application are all that there is. They can be mixed freely and used as often as desired or needed, which is another way of saying that λ-terms are constructed according to the grammar

$$M \;\; ::= \;\; c \;\mid\; x \;\mid\; M \; M \;\mid\; \lambda x.M$$

Here the placeholder c represents any **constant** we might wish to use in a λ-term, such as numbers 1, 2, 3,... or arithmetic operators $+$, $*$, etc. A term without constants is called **pure**. Similarly, the letter x represents any of infinitely many possible variables.

The grammar is ambiguous; the term $\lambda x.x \; y$ could be parsed as



(where we use "app" to indicate use of the clause "$M\ M$" in the derivation). With auxiliary brackets, the two possible interpretations can be indicated by writing $\lambda x.(x\ y)$ and $(\lambda x.x)\ y$, respectively. According to our convention from above, only the first interpretation should be possible. Using additional non-terminals and productions the conventional interpretation can be enforced:

$$\begin{array}{rcl} \text{<term>} & ::= & \text{<atom>} \mid \text{<app>} \mid \text{<fun>} \\ \text{<atom>} & ::= & \text{<head}-\text{atom>} \mid (\ \text{<app>}\ ) \\ \text{<head}-\text{atom>} & ::= & x \mid c \mid (\ \text{<fun>}\ ) \\ \text{<app>} & ::= & \text{<head}-\text{atom> <atom>} \mid \text{<app> <atom>} \\ \text{<fun>} & ::= & \lambda x.\text{<term>} \end{array}$$

but only a compiler would be interested in so much detail.

6. Reduction. λ-terms on their own would be a bit boring if we didn't know how to *compute* with them as well. There is only one rule of computation, called **reduction** (or β-reduction, as it is known by the aficionados), and it concerns the replacement of a formal parameter by an actual one. It can only occur if a functional term has been applied to some other term. Two examples:

$$(\lambda x.*\ 3\ x)\ 4 \quad \longrightarrow_\beta \quad *\ 3\ 4$$

$$(\lambda y.y\ 5)(\lambda x.*\ 3\ x) \quad \longrightarrow_\beta \quad (\lambda x.*\ 3\ x)\ 5 \quad \longrightarrow_\beta \quad *\ 3\ 5$$

We see that reduction is nothing other than the textual replacement of a formal parameter in the body of a function by the actual parameter supplied.

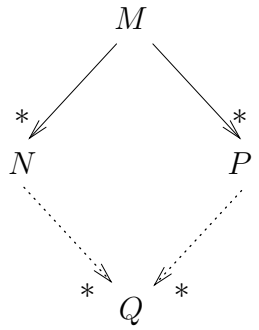
One would expect a term after a number of reductions to reach a form where no further reductions are possible. Surprisingly, this is not always the case. The following is the smallest counterexample:

$$\Omega \quad \stackrel{\text{def}}{=\!=} \quad (\lambda x.x\ x)(\lambda x.x\ x)$$

The term Ω always reduces to itself. If a sequence of reductions has come to an end where no further reductions are possible, we say that the term has been reduced to **normal form**. As Ω illustrates, not every term has a normal form.

7. Confluence. It may be that a λ-term offers many opportunities for reduction at the same time. In order for the whole calculus to make sense, it is necessary that the result of a computation is independent from the order of reduction. We would like to express this property for all terms, not just for those which have a normal form. This is indeed possible:

Theorem 1 (Church-Rosser) *If a term M can be reduced (in several steps) to terms N and P, then there exists a term Q to which both N and P can be reduced (in several steps). As a picture:*

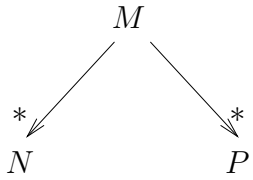


(The little $*$ next to the arrows indicates several instead of just a single reduction. "Several" can also mean "none at all".)

For obvious graphical reasons, the property expressed in the Theorem of Church and Rosser is also called **confluence**. We say that β-reduction is **confluent**. The following is now an easy consequence:

Corollary 2 *Every λ-term has at most one normal form.*

Proof. For the sake of contradiction, assume that there are normal forms N and P to which a certain term M reduces:



By the theorem of Church and Rosser there is a term Q to which both N and P can be reduced. However, N and P are assumed to be in normal form, so they don't allow for any further reductions. The only possible interpretation is that $N = P = Q$. ∎

8. Exercises

1. Translate the following Java expressions into λ-calculus notation:
   (a) `sin(x+3)`
   (b) `length(y)+z`
   (c)
   ```
   public static int quot(double x, double n)
             { return (int)(x/n); }
   ```
2. Draw the syntax trees for the following λ-terms:
   (a) $\lambda xy.x$
   (b) $\lambda xyz.xyz$
   (c) $(\lambda x.xx)(\lambda x.xx)$
3. Reduce to normal form:
   (a) $(\lambda x.+\ x\ 3)4$
   (b) $(\lambda fx.f(fx))\ (\lambda y.*\ y\ 2)\ 5$
4. Let T be the λ-term $\lambda x.xxx$. Perform some β-reductions on TT. What do you observe?
5. Let S be the term $\lambda xyz.(xz)(yz)$ and K the term $\lambda xy.x$. Reduce SKK to normal form. (Hint: This can be messy if you are not careful. Keep the abbreviations S and K around as long as you can and replace them with their corresponding λ-terms only if you need to. It becomes very easy then.)
6. Let S be the term $\lambda xyz.(xz)(yz)$ and K the term $\lambda xy.x$. Reduce SKK to normal form. (Hint: This can be messy if you are not careful. Keep the abbreviations S and K around as long as you can and replace them with their corresponding λ-terms only if you need to. It becomes very easy then.)
7. Let Z be the λ-term $\lambda zx.x(zzx)$ and let Y be ZZ. By performing a few β-reductions, show that YM will be a fixpoint of M for any term M, i.e., we have $YM =_\beta M(YM)$.
8. Suppose a symbol of the λ-calculus alphabet is always 5mm wide. Write down a pure λ-term (i.e., without constants) with length less than 20cm having a normal form with length at least $10^{10^{10}}$ light-years. (A light-year is about 10^{13} kilometers.)

9. 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.

10. 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:

$$\begin{array}{lcll} \text{zero?} \; 0 \; x \; y & \longrightarrow & x & \\ \text{zero?} \; n \; x \; y & \longrightarrow & y & (n \neq 0) \end{array}$$

In Java, we would write this as

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

We also assume constants "pred" and "succ" for predecessor and successor function on natural numbers.

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:

$$I \; n \; f \; x \;\; = \;\; f \; (f \; (f \; \ldots \; (f \; x) \; \ldots))$$

If $n = 0$ then $I \; 0 \; f \; x$ should simply return x, without applying f at all. 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 iterate $f \quad (n-1)$-times on the argument $(f \; x)$; if successful, this will return f applied to x n-times.

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 you do it $(n-1)$-times for me first...". 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:

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

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":

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

This definition is no longer circular, so S is a proper term. What we now seek is a term I which satisfies

$$I \;=\; S\ I$$

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

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 $I \stackrel{\text{def}}{=} Y\ S$.

We call such a Y a **fixpoint combinator**. Here is Turing's fixpoint combinator:

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

To check that $Y\ M$ reduces to $M\ (Y\ M)$ takes just two reduction steps. It was an exercise on the last exercise sheet.

Fixpoints are also used to create `while`-loop like behaviour. Consider, for example, the problem of finding the smallest number for which a given function 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 M:

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

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 $Y\ L\ f\ 0$.

11. 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 $\mathbb{N}$ 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. For the other direction, we have several choices for the precise version of the λ-calculus we want to consider. As I have developed the calculus in this handout, the first one should be the following:

Theorem 3 *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, for example the following will do:

$$0 \stackrel{\text{def}}{=\!=} \lambda f\ x.x$$
$$n \stackrel{\text{def}}{=\!=} \lambda f\ x.f\ (f\ \ldots\ (f\ x)\ \ldots) \qquad (n\text{-fold application of } f \text{ to } x)$$

With this representation in mind, the following is true:

Theorem 4 *The pure λ-calculus is Turing-complete.*

12. 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 "int" for integers and "real" for 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**):

$$\tau \ ::= \ c \mid \tau \to \tau$$

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.

With such a system, the type of the sine function can be denoted by "$\mathrm{real} \to \mathrm{real}$" and it is obvious that it cannot accept the logarithm function as an argument because the latter also has type "$\mathrm{real} \to \mathrm{real}$" and not "$\mathrm{real}$" 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:

Definition 5 (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 $\sigma \to \tau$.*

Application. *If M is well-typed of type $\sigma \to \tau$ and N is well-typed of type σ then $M\ N$ 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 $\sigma \to \sigma$ no matter what σ stands for. The term $\lambda x{:}\sigma.\lambda y{:}\tau.x{:}\sigma$ is well-typed of type $\sigma \to (\tau \to \sigma)$. On the other hand, the term $\sin\ \log$ is not well-typed. Furthermore, any term of the shape $M\ M$ cannot be annotated with simple types (Exercise 4).

13. 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** $A, B, C, \ldots$ and which we refine as we go along. Consider, for example, the term $\lambda f\ x.f\ x$: 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 $A \to C$, 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 $A \to C$. Likewise, for the abstraction $\lambda f.\lambda x.f\ x$ we remember that f should have type $A \to C$. According to the function formation rule, then, the complete term should have type $(A \to C) \to (A \to C)$. 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 $D \to D$, with D a fresh type variable. On the other hand, we have type $(A \to C) \to (A \to C)$ 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 is $D \to D$. 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:

$$(\lambda f{:}\mathrm{int} \to \mathrm{int}\ \lambda x{:}\mathrm{int}.f\ x)\ (\lambda y{:}\mathrm{int}.y)\ 3$$

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

Theorem 6 *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 "int", and the following constants:

Numerals. A constant $\overline{\mathrm{n}}$ of type int for every natural number n.

Conditional. Constants zero?$_\sigma$ of type $\mathrm{int} \to (\sigma \to (\sigma \to \sigma))$ for every type σ.

Successor function. Constants succ and pred of type $\mathrm{int} \to \mathrm{int}$.

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

We have:

Theorem 7 *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.

15. Exercises

1. Leaving f as an unspecified variable, try to evaluate the term $T\ T\ f\ x$ from Section 1 to normal form.

2. Find type annotations which show that T is well-typed.

3. Show that $I\ 3$ (Section 10) evaluates to T (as one would hope).

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

16. Learn more...

1. G. Michaelson. *An Introduction to Functional Programming through Lambda Calculus*. Addison-Wesley, 1989.

   This is a nice gentle introduction to the λ-calculus, leading towards functional programming. The early chapters are very accessible.

2. N. D. Jones. *Computability and Complexity: From a Programming Perspective*. MIT Press, 1997.

   This is one of the very few books which subscribe to the view that undecidability can be explained without Turing machines. However, the underlying formalism is that of functional programming with which you'd better be familiar.

3. J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ-Calculus*. Cambridge University Press, 1986.

   A formal introduction to the Lambda Calculus. Fairly mathematical.

4. H. P. Barendregt. *The Lambda Calculus: Its Syntax and Semantics*. North-Holland, revised edition, 1984.

   The standard reference on the subject.

5. D. S. Scott. A type-theoretical alternative to ISWIM, CUCH, OWHY. *Theoretical Computer Science*, 121:411–440, 1993. Reprint of a manuscript written in 1969.

   The typed lambda calculus with the constants of LCF/PCF is proposed as a standard description language for computable functions.