

Department of Computer Science
Australian National University

COMP3610
Principles of Programming Languages

An Introduction to the Lambda Calculus

Clem Baker-Finch
August 8, 2007

Contents

1	Motivation	1
2	The Untyped Lambda Calculus	1
2.1	Introduction	1
2.1.1	Scope	3
2.2	β Conversion	3
2.3	β Reduction	4
2.4	Substitution	6
3	Lambda Calculus as a Model of Computation	7
3.1	Multiple Arguments	8
3.2	Booleans	8
3.3	Church Numerals	9
3.4	Combinators	12
3.5	Recursion	12
4	The Typed Lambda Calculus	14
4.1	Extending the Typed Lambda Calculus	14
4.2	The Typing Relation	15
4.3	The Curry-Howard Correspondence	16
4.4	Conditional Expressions	17
4.5	Type Variables and Polymorphism	18
4.6	Recursion	19

1 Motivation

The lambda calculus is a system (calculus) of pure functions. That is, *everything* is a function. It was invented in the 1940s by logician Alonzo Church as a study of the foundations of mathematics and computation. The lambda calculus is one of the most important cornerstones of Computer Science. Of particular interest for us, the λ -calculus is:

- one of the standard *models of computation*
- a foundation for *denotational semantics*
- the core of all *functional programming languages*
- yet another view of *natural deduction*

Much of programming language research has been built on the lambda calculus. For example, the foundations of object-oriented languages may be studied in terms of the lambda calculus extended with records. The experimental language *PCF* (Programming language of Computable Functions) is a small extension of the lambda calculus that has been widely studied by programming language researchers as the simplest language to demonstrate a number of interesting semantic features.

LISP, one of the first programming languages to be invented, is fundamentally based on the lambda calculus.

The type systems in languages like Haskell and ML are extensions of the simply typed lambda calculus (and *PCF*).

This section of the COMP3610 lecture notes provide an introduction to the λ -calculus, mainly focusing on the dot-points listed above. Other parts of the course may make reference to, or be based on these notes.

2 The Untyped Lambda Calculus

There are in fact many variants of λ -calculi: untyped, λI , λK , simply typed (Church or Curry style), second-order polymorphic λ -calculus, System F_ω etcetera, etcetera. We will only look at two, beginning with the *untyped* λ -calculus followed by the *typed* λ -calculus extended with some features reminiscent of the functional programming languages with which we are familiar.

2.1 Introduction

The λ -calculus is a calculus of “pure” functions. To see what that might mean, consider the notation we have learnt in Mathematics classes to define functions:

$$f\ x = x^2 + x + 1$$

(Mathematicians are probably more likely to write $f(x) = \dots$ but we’re computer scientists.) How does this describe a function? What is “the function” itself? We’re forced to speak indirectly in terms of a *name*, the *argument* and the *result*: “ f is the function which, when applied to any argument x , yields $x^2 + x + 1$.” The name of the function f is arbitrary — it may just as well be g or $fred$. (Of course, the parameter name x is equally arbitrary.) Can we describe and define functions without giving them names?

Lambda notation allows us to do so. First, instead of writing the formal parameter on the left of the defining equation, a different choice of notation allows us to put it on the right:

$$f = \lambda x. x^2 + x + 1$$

where the formal parameter is identified by prefixing it with λ and separating it from the function body with a period. Now f is still the same function; only the notation we have used to define it has changed. What is really interesting is that the left hand side of the equation is no longer in the form of an application — it's just a name, so the right hand side *is* the function and we thus have a way of expressing functions directly without giving them names. We sometimes refer to them as *anonymous functions*.

$$\lambda x. x^2 + x + 1$$

is the function which, when applied to any argument x , yields $x^2 + x + 1$. Applying the function to an argument, say $(f\ 3)$, is expressed by writing the function followed by the argument (with parentheses if necessary to avoid ambiguity):

$$(\lambda x. x^2 + x + 1)\ 3$$

Such an application is evaluated by replacing the parameter x by 3 in the body to give $3^2 + 3 + 1$, and arithmetic tells us the result is 13.

The sceptical reader may question how this might extend to *recursive* function definitions which it seems must rely on functions being named. Consider

$$fact\ n = \mathbf{if}\ n == 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times fact(n - 1)$$

Since the name seems to be essential to express the recursion (it appears on both sides of the equation) it is reasonable to ask whether recursive functions can be defined anonymously using lambda notation. The answer is “yes” and is the main topic of a later section of the course, but we will also deal with it briefly here in terms of the λ -calculus.

The pure λ -calculus embodies this kind of function definition and application in its purest form. In the pure λ -calculus *everything* is a function: arguments to functions are themselves functions and the result returned by a function is another function.

The syntax of lambda terms comprises only three kinds of terms. A variable x is a term; the *abstraction* of a variable from a term M is a term; and the application of one term M to another term N is a term. We also allow parentheses to express the structure of terms. Later in the course we will consider the concepts of *concrete* and *abstract* syntax, where the real purpose of the parentheses will become clearer.

$$M ::= x \mid \lambda x. M \mid MN \mid (M)$$

The term x is a *variable*, $\lambda x. M$ is an *abstraction*, and MN is an *application*.

Intuitively, abstractions represent *functions*. The variable following the λ is the parameter. Evaluation is by *substitution*, e.g.:

$$\begin{aligned} (\lambda x. \lambda y. x)MN &= (\lambda y. M)N && \text{(by substituting } M \text{ for parameter } x \text{ in the body } \lambda y. x) \\ &= M && \text{(by substituting } N \text{ for parameter } y \text{ in the body } M) \end{aligned}$$

Parentheses are minimised by convention:

- application associates to the left (as is familiar from Haskell):
 $MNP = (MN)P$
- application has precedence over abstraction:
 $\lambda x.MN = \lambda x.(MN)$
 so the body of the λ -abstraction continues as far to the right as possible;
- sequences of λ s may be collapsed:
 $\lambda xyz.M = \lambda x.\lambda y.\lambda z.M$

For example, $\lambda xy.xyx$ is taken to be the same term as $\lambda x.(\lambda y.((xy)x))$

2.1.1 Scope

An occurrence of a variable x is said to be *bound* when it occurs in the body M of an abstraction $\lambda x.M$. We say that λx is a *binder* whose scope is M . An occurrence of x is *free* if it appears in a position where it is not bound by an enclosing abstraction on x .

For example, the occurrences of x in xy and $\lambda y.xy$ are free while the occurrences of x in $\lambda x.x$ and $\lambda z.\lambda x.\lambda y.x(yz)$ are bound. In $(\lambda x.x)x$ the first occurrence of x is bound and the second is free. You might like to think of bound variables as *local* and free variables as *global*.

We can inductively define the set of bound variables in a term as follows:

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

Similarly free variables:

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

A term with no free variables is *closed*. We often call closed terms *combinators*. The simplest combinator is the identity function:

$$\text{id} = \lambda x.x$$

2.2 β Conversion

The original studies of λ -calculus focused on the *equational theory* produced by evaluation of applications. for reasons I have yet to discover, this was called *beta conversion*.

$$(\lambda x.M)N = M[x := N] \tag{\beta}$$

Where the notation $M[x := N]$ means to substitute term N for all occurrences of x in term M . We will have more to say about substitution later in these notes.

The intuition is that

- $\lambda x.M$ is a *function* with *parameter* x .
- $(\lambda x.M)N$ is an *application* and N is the *argument*.
- The application is *evaluated* by *substituting* the argument for the parameter in the body of the function.

We call a term of the form $(\lambda x.M)N$ a *reducible expression* or a *redex* for short.

β -conversion is the basis of a theory of equality between λ -terms. To complete the theory we need a few structural rules.

Conversion applies to any subterms:

$$\frac{M = N}{MP = NP}$$

$$\frac{M = N}{PM = PN}$$

$$\frac{M = N}{\lambda x.M = \lambda x.N}$$

(=) is an equivalence relation:

$$M = M \quad (\textit{reflexive})$$

$$\frac{M = N}{N = M} \quad (\textit{symmetric})$$

$$\frac{M = L \quad L = N}{M = N} \quad (\textit{transitive})$$

2.3 β Reduction

Conversion is all very interesting, but as computer scientists we're typically more interested in *computation*, which essentially implies a *direction* on the beta rule. In its pure form, the λ -calculus has no built-in constants or primitive operators and the sole means by which terms "compute" is the application of functions to arguments (which are themselves functions).

Instead of β -conversion we have β -reduction:

$$(\lambda x.M)N \rightarrow M[x := N] \quad (\beta)$$

Since a lambda term may have several redexes, the question remains as to which redex to choose to evaluate next. Different *evaluation strategies* lead to different outcomes.

Full β -reduction. Under this strategy, *any* redex may be chosen as the next one to reduce. For example, the term

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

which we can write more readably as $\text{id}(\text{id}(\lambda z.\text{id}z))$, contains three redexes:

$$\begin{array}{c} \text{id}(\text{id}(\lambda z.\text{id}z)) \\ \text{id}(\underline{\text{id}(\lambda z.\text{id}z)}) \\ \text{id}(\text{id}(\lambda z.\underline{\text{id}z})) \end{array}$$

If a term contains no redexes, we say it is in *normal form*. For this term we end up with the same normal form $\lambda z.z$ no matter which order we choose the next redex. In fact this is a general result:

Under full β -reduction, if a term has a normal form that normal form is *unique*. This is a consequence of a stronger result known as the **Church-Rosser** property which says that if we can reach two distinct terms M and M' by β -reduction from some term N , then there is a term P to which both M and M' β -reduce.

Normal Order. Under this strategy, the *leftmost outermost redex* is always reduced first. The term above would be reduced as follows

$$\begin{array}{l} \text{id}(\text{id}(\lambda z.\text{id}z)) \\ \rightarrow \underline{\text{id}(\lambda z.\text{id}z)} \\ \rightarrow \lambda z.\underline{\text{id}z} \\ \rightarrow \lambda z.z \end{array}$$

Normal order is a *normalising strategy* in the sense that if a term has a normal form, this strategy will find it.

Applicative Order. Under this strategy, the *leftmost innermost redex* is always reduced first. The term above would be reduced as follows:

$$\begin{array}{l} \text{id}(\text{id}(\lambda z.\text{id}z)) \\ \rightarrow \underline{\text{id}(\lambda z.\text{id}z)} \\ \rightarrow \lambda z.\underline{\text{id}z} \\ \rightarrow \lambda z.z \end{array}$$

In contrast with normal order, under this strategy arguments are evaluated before being passed to functions. It turns out that applicative order is not a normalising strategy. For example, suppose that M does not have a normal form (that is, reducing M does not terminate). In that case, applicative order evaluation of $(\lambda x.\lambda y.y)M$ will never find the normal form of the term $(\lambda y.y)$.

Call-by-Name. This is a restriction of normal order, where no reductions are allowed *inside* abstractions. For our example term, the reduction is the same but stops earlier:

$$\begin{aligned} & \underline{\text{id}(\text{id}(\lambda z.\text{id}z))} \\ \rightarrow & \underline{\text{id}(\lambda z.\text{id}z)} \\ \rightarrow & \lambda z.\text{id}z \end{aligned}$$

Algol-60 introduced call-by-name parameter passing, and Haskell’s semantics is a more efficient variant known as *call-by-need* where instead of re-evaluating an argument each time it is used, all occurrences of the argument are overwritten with its value the first time it is evaluated. This requires that we have a run-time representation of terms that allows sharing of subexpressions, leading to a terms becoming *graphs* rather than trees.

Call-by-Value. Most languages, including Java and ML use this strategy, where again no reductions are allowed inside abstractions *and* a redex is only reduced when its argument part has already been reduced to a *value* — a term which cannot be reduced any further.

$$\begin{aligned} & \text{id}(\underline{\text{id}(\lambda z.\text{id}z)}) \\ \rightarrow & \underline{\text{id}(\lambda z.\text{id}z)} \\ \rightarrow & \lambda z.\text{id}z \end{aligned}$$

Call-by-value is related to applicative order, in exactly the same way that call-by-name relates to normal order. The call-by-value strategy is *strict* in the sense that the arguments to a function are evaluated whether or not they are used by the function. In contrast *non-strict* (or *lazy*) strategies such as call-by-name and call-by-need only evaluate the arguments that are actually used. The difference between strict and non-strict becomes clearer when we consider the possibility of non-termination. We will see a more formal definition of these concepts later in the course.

2.4 Substitution

The idea of substituting a term for all occurrences of a variable, as in the definition of β -reduction is intuitively appealing but it turns out that a formal definition is quite a delicate matter. A naïve (and ultimately incorrect) attempt might be:

$$\begin{aligned} x[x := N] &= N \\ y[x := N] &= y \\ (\lambda y.M)[x := N] &= \lambda y.(M[x := N]) \\ (MP)[x := N] &= (M[x := N])(P[x := N]) \end{aligned}$$

For most examples this seems to work. For example

$$(\lambda y.x)[x := (\lambda z.zw)] = \lambda y.\lambda z.zw$$

which matches our intuitions about how substitution should behave. However

$$(\lambda x.x)[x := y] = \lambda x.y$$

conflicts with a basic understanding that the names of bound variables (i.e. parameters) don't matter. The identity function is the same whether we write it as $\lambda x.x$ or $\lambda z.z$ or $\lambda \text{fred.fred}$. If these don't behave the same way under substitution they won't behave the same way under evaluation and that seems wrong. The mistake is that the substitution should only apply to *free* variables and not *bound* ones. In the example, x is bound in the term so we should not substitute it. That seems to give us what we want:

$$(\lambda x.x)[x := y] = \lambda x.x$$

But that's still not quite enough, as the following example demonstrates:

$$(\lambda z.x)[x := z] = \lambda z.z$$

This has changed a constant function into the identity function — in some sense this is the dual of the problem identified above. Once again, the choice of z as the binder in $\lambda z.x$ should be completely arbitrary. This phenomenon is known as *free variable capture* as the z being substituted is free, but in the result it is bound. Our solution will be to allow renaming of bound variables (a process traditionally known as α -conversion). We are now in a position to give a formal definition of substitution which accounts for the issues explored here.

$$\begin{aligned} x[x := N] &= N \\ y[x := N] &= y \\ (\lambda x.M)[x := N] &= \lambda x.M && (\dagger) \\ (\lambda y.M)[x := N] &= \lambda y.M[x := N] \quad \text{if } y \notin FV(N) \\ (\lambda y.M)[x := N] &= \lambda z.M[y := z][x := N] \quad \text{if } y \in FV(N), z \text{ a fresh variable} && (\ddagger) \\ (MP)[x := N] &= (M[x := N])(P[x := N]) \end{aligned}$$

assuming that x and y are different variables.

Notice in (\dagger) that the substitution does not apply to bound occurrences of x . In (\ddagger) notice the replacement of bound variables to avoid capture of free variables.

That all seems quite complicated — and it is — so it is common to abide by a convention rather than keep dealing with this machinery. Typically we work with lambda terms “up to renaming of bound variables” or “up to α -conversion” and don't allow variable capture. In fact many compilers, including `ghc`, rename *all* variables to be unique, so as to circumvent such problems.

Exercise 2.1

The λ -calculus reduction workbench, available through the course web site, automatically generates fresh variables to avoid free variable capture. Try evaluating some term on the workbench, such as `(\ x . \ y .x) y` and `(\ x . \ x . x) y` to see it in action.

3 Lambda Calculus as a Model of Computation

So far this may all seem uncomfortably vacuous — it all looks like symbol pushing and nothing more. There are none of the *data values* which we are used to having at the basis of computation, such as numbers and booleans. Without them, what can we *do* with the λ -calculus? In this section we will see how to *represent* such data types in the λ -calculus. This is not such an unfamiliar idea: we are used to the idea of interpreting sequences of bits as integers, floats,

characters, instructions and so on. Here we will interpret certain lambda terms as representing boolean and numeric values, in a manner that can be extended to other classes of values and data structures. What arises from defining a sufficient set of representations is a model of computation equivalent in power to Turing machines, recursive function theory and every other standard model of computation.

3.1 Multiple Arguments

First we will deal with a straightforward matter. Many functions of interest (addition, for example) take more than one argument, yet λ -calculus abstractions only have a single binder. There are two solutions: either pass a single argument consisting of a *tuple* of values, or *curry* the functions. The first solution requires that we find a way to represent tuples in the λ -calculus, which turns out to be not too difficult. The second approach, which we will adopt here, treats a function of several arguments as taking each argument in turn, one at a time, and should be familiar from your Haskell programming experience.¹

3.2 Booleans

It may be that we naturally consider boolean values to somehow be primitive or atomic — they are what they are. In fact, what we are interested in (and this is a rather “algebraic” attitude that extends to *all* data types and structures) is how they *behave*. At the heart of our understanding of boolean values is that they guide a *choice*: if something is true choose this, otherwise choose that. So what we want is to come up with a way of *representing* true, false and choice (i.e. conditional) as lambda terms, such that they *work together* as we wish — that is, their evaluation leads to results that correspond to (representations of) our intuitive computational expectations.

$$\begin{aligned} \text{tt} &= \lambda xy.x \\ \text{ff} &= \lambda xy.y \\ \text{cond} &= \lambda abc.abc \end{aligned}$$

where `tt` represents *true*, `ff` represents *false* and `cond` represents *if-then-else*.

Notice that `tt` is a function that takes two arguments and returns the first, while `ff` also takes two arguments and returns the second. The `cond` combinator takes three arguments and applies the first to the second and the third. The key observation is that the first argument to `cond` will be either `tt` or `ff`, thus choosing either the second or the third argument respectively.

So for example, **if true then M else N** is represented by the term:

$$\text{cond tt } MN = (\lambda abc.abc)(\lambda xy.x)MN$$

¹The expression *curry* is after Haskell Brooks Curry, a student of Church and one of the λ -calculus pioneers.

which evaluates as follows:

$$\begin{aligned}
 & (\lambda abc.abc)(\lambda xy.x)MN \\
 \rightarrow & (\lambda bc.(\lambda xy.x)bc)MN \\
 \rightarrow & (\lambda c.(\lambda xy.x)Mc)N \\
 \rightarrow & (\lambda xy.x)MN \\
 \rightarrow & (\lambda y.M)N \\
 \rightarrow & M
 \end{aligned}$$

as we would wish.

Exercise 3.1

Complete a similar evaluation of `cond ff MN`

A little thought should convince you that other logical operators such as conjunction and disjunction can be defined in terms of these three notions, so we have constructed three combinators that form the basis of a representation of booleans in the λ -calculus.

Exercise 3.2

Using logical equivalences like

$$x \text{ and } y = \text{if } x \text{ then } y \text{ else false}$$

define combinators `and`, `or` and `not` representing logical conjunction, disjunction and negation.

3.3 Church Numerals

One way of representing the natural numbers in the λ -calculus is as *Church numerals*, which have a similar flavour to *Peano arithmetic*. The natural numbers can be defined inductively as follows:

- **zero** is a natural number;
- if n is a natural number, then **succ**(n) is also a natural number.

So for example

$$3 = \text{succ}(\text{succ}(\text{succ}(\text{zero})))$$

The Church numerals $c_0, c_1, c_2, c_3, \dots$ are defined as follows:

$$\begin{aligned}
 c_0 &= \lambda sz.z \\
 c_1 &= \lambda sz.sz \\
 c_2 &= \lambda sz.s(sz) \\
 c_3 &= \lambda sz.s(s(sz)) \\
 &\dots
 \end{aligned}$$

We have used s and z as suggestive names for the bound variables, but of course they are quite arbitrary. The idea is that c_n takes two arguments s and z (for successor and zero) and applies s ,

n times to z . Once again, we are not so interested in *representation* but rather in the *interaction* with the combinators representing primitive operations on numbers, such as **succ**.

A successor function on Church numerals needs to take the c_n term apart to get the body, add another s application, then rewrap it in the binders:

$$\text{succ} = \lambda n. \lambda s z. s(nsz)$$

For example:

$$\begin{aligned} \text{succ } c_2 &= (\lambda n. \lambda s z. s(nsz))(\lambda ab. a(ab)) \\ &\rightarrow \lambda s z. s((\lambda ab. a(ab))sz) \\ &\rightarrow \lambda s z. s((\lambda b. s(sb))z) \\ &\rightarrow \lambda s z. s(s(sz)) \\ &= c_3 \end{aligned}$$

Addition can be performed by a term **plus** that takes two Church numerals, m and n , and yields another Church numeral (i.e. a function) that accepts arguments s and z , applies s iterated n times to z (by passing arguments s and z to n), and then applies s iterated m more times to the result:

$$\text{plus} = \lambda m. \lambda n. \lambda s z. ms(nsz)$$

Exercise 3.3

Convince yourself that this definition is correct by evaluating $\text{plus } c_2 c_3$. Do it by hand and try some examples using the workbench.

Exercise 3.4

Define combinators for multiplying and exponentiating Church numerals. A natural approach to defining multiplication is to use the **plus** combinator, but you may also find another way. Subtraction and predecessor combinators are probably too difficult for you to invent, but they are among the predefined combinators that come with the λ -calculus workbench. Look at their definitions and try to understand and describe how they work.

Example 3.5

Here is an example reduction to normal form, with redexes in **red** (the function) and **blue** (the argument) and their residuals in **green**, with the substitutions of the argument still in **blue**. The

expression is $\text{cond tt (succ } c_2) c_0$:

$$\begin{aligned}
& (\lambda bxy.bxy)(\lambda tf.t)((\lambda nfx.f(nfx))(\lambda sz.s(sz)))(\lambda sz.z) \\
\rightarrow & (\lambda xy.(\lambda tf.t)xy)((\lambda nfx.f(nfx))(\lambda sz.s(sz)))(\lambda sz.z) \\
& (\lambda xy.(\lambda tf.t)xy)((\lambda nfx.f(nfx))(\lambda sz.s(sz)))(\lambda sz.z) \\
\rightarrow & (\lambda y.(\lambda tf.t)((\lambda nfx.f(nfx))(\lambda sz.s(sz)))y)(\lambda sz.z) \\
& (\lambda y.(\lambda tf.t)((\lambda nfx.f(nfx))(\lambda sz.s(sz)))y)(\lambda sz.z) \\
\rightarrow & (\lambda tf.t)((\lambda nfx.f(nfx))(\lambda sz.s(sz)))(\lambda sz.z) \\
& (\lambda tf.t)((\lambda nfx.f(nfx))(\lambda sz.s(sz)))(\lambda sz.z) \\
\rightarrow & (\lambda f.(\lambda nfx.f(nfx))(\lambda sz.s(sz)))(\lambda sz.z) \\
& (\lambda f.(\lambda nfx.f(nfx))(\lambda sz.s(sz)))(\lambda sz.z) \\
\rightarrow & (\lambda nfx.f(nfx))(\lambda sz.s(sz)) \\
& (\lambda nfx.f(nfx))(\lambda sz.s(sz)) \\
\rightarrow & \lambda fx.f((\lambda sz.s(sz))fx) \\
& \lambda fx.f((\lambda sz.s(sz))fx) \\
\rightarrow & \lambda fx.f((\lambda z.f(fz))x) \\
& \lambda fx.f((\lambda z.f(fz))x) \\
\rightarrow & \lambda fx.f(f(fx))
\end{aligned}$$

Which is c_3 , as expected.

Another class of operators on numbers are the relational operators, bringing the representation of booleans and numbers together. Consider testing whether a Church numeral is zero. To achieve this we must find some appropriate pair of arguments that will give us back this information. Specifically we want to apply our numeral to a pair of terms ss and zz (so ss is substituted for s and zz is substituted for z) such that applying ss to zz one or more times yields ff while not applying it at all yields tt . The constant function $\lambda x.ff$ serves as ss and tt as zz , leading to:

$$\text{iszero} = \lambda m.m(\lambda x.ff)tt$$

For example

$$\begin{aligned}
\text{iszero } c_1 &= (\lambda m.m(\lambda x.ff)tt)c_1 \\
&\rightarrow c_1(\lambda x.ff)tt \\
&= (\lambda sz.sz)(\lambda x.ff)tt \\
&\rightarrow (\lambda z.(\lambda x.ff)z)tt \\
&\rightarrow (\lambda x.ff)tt \\
&\rightarrow ff
\end{aligned}$$

and

$$\begin{aligned}
\text{iszero } c_0 &= (\lambda m.m(\lambda x.ff)tt)c_0 \\
&\rightarrow c_0(\lambda x.ff)tt \\
&= (\lambda sz.z)(\lambda x.ff)tt \\
&\rightarrow (\lambda z.z)tt \\
&\rightarrow tt
\end{aligned}$$

3.4 Combinators

In the two sample evaluations above, we worked with a mixture of combinators and pure terms, expanding the combinators as necessary before performing β reduction. This was important because our intention was to informally verify that our choice of combinators to represent the various functions and values was correct. However, taking a more abstract approach, we can work simply with “higher-level” rules dealing directly with the combinators. For example, the following rules can be derived from their definitions as lambda terms:

$$\begin{aligned} \text{iszero } c_0 &= \text{tt} \\ \text{iszero } c_n &= \text{ff} \quad \text{for all } n \neq 0 \\ \text{succ } c_n &= c_{n+1} \\ \text{plus } c_m \ c_n &= c_{m+n} \end{aligned}$$

and so on. This kind of conceptual abstraction is very familiar to computer scientists. For example, even when coding in assembler language we think in terms of data values passing between registers and memory locations, rather than the bit pattern of the instruction and the micro-code running on the processor. Taking that idea one step further, notice that the lambda term for **plus** only behaves as addition under a particular interpretation involved with Church numerals — in another context it’s just another lambda term. The same idea of different interpretation of the same data object is familiar in computer science: the same bit pattern may represent a character, a number, an instruction and so on.

3.5 Recursion

Every reasonable model of computation must allow for some form of *repetition*, but so far we have only talked about simple values and primitive operations. If we consider the definition of the factorial function mentioned in section 2.1:

$$\text{fact } n = \mathbf{if } n == 0 \mathbf{ then } 1 \mathbf{ else } n \times \text{fact}(n - 1)$$

We have combinators for most components of this definition, e.g.

$$\text{fact} = \lambda n. \text{cond} (\text{iszero } n) \ c_1 \ (\text{mul } n \ (\text{fact}(\text{pred } n)))$$

To complete the job we want to find a lambda term that represents the *fact* function. At the moment we still rely on naming the function and giving an equation for which we have learned a way to interpret computationally. As mentioned in section 2.1 we will look at this in more depth in a later section of the course.

Recall that a term that cannot take a step under the evaluation relation is called a *normal form*. It is interesting that some terms do not have a normal form, for example the *divergent* combinator:

$$\text{omega} = (\lambda x.xx)(\lambda x.xx)$$

contains exactly one redex (the whole term) and reducing it yields exactly the same term **omega** again.

There are various combinators, related to **omega**, known as *fixpoint operators*. A well-known one is the **Y** combinator:²

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

²Why not?

Just looking at the term isn't very helpful. To get some intuition of its behaviour we will explore a specific example. When we write recursive function definitions like

$$f = \langle \textit{body containing } f \rangle$$

the intention is that the definition should be “unfolded” inside the body. For example, the intuition about the factorial function definition in section 2.1 is an infinite unfolding:

```

if n == 0 then 1
else n * (if n-1 == 0 then 1
          else (n-1) * (if n-2 == 0 then 1
                       else (n-2) * (if n-3 == 0 then 1
                                       else (n-3) * ...

```

Or, using Church numerals:

```

cond (iszero n) c1
  (mul n (cond (iszero (pred n)) c1
              (mul (pred n) (cond (iszero (pred(pred n))) c1
                                  (mul (pred(pred n)) (cond (iszero (pred(pred(pred n)))) c1
                                                              (mul (pred(pred(pred n)))) ...

```

The Y combinator can give us this unfolding effect. First we define:

$$g = \lambda h. \langle \textit{body containing } h \rangle$$

and then:

$$f = Y g$$

Doing this to the factorial function gives:

$$\begin{aligned}
g &= \lambda f. \lambda n. \text{cond (iszero } n) \text{ } c_1 \text{ (mul } n \text{ (} f \text{ (pred } n)))} \\
\textit{fact} &= Y g
\end{aligned}$$

Let's see what happens as we begin evaluating (*fact* *c*₃). Rather than writing out the whole term in λ -notation, we will leave in the combinator names, expanding them as necessary.

$$\begin{aligned}
\textit{fact } c_3 &= (Y g) c_3 \\
&= ((\lambda f. (\lambda x. f(xx)))(\lambda x. f(xx))) g) c_3 \\
&\rightarrow (\lambda x. g(xx))(\lambda x. g(xx)) c_3 \\
&\rightarrow g ((\lambda x. g(xx))(\lambda x. g(xx))) c_3
\end{aligned}$$

Now notice that the first argument to *g* (the subterm in blue) is just (*Y g*). Observe that *fact* has a “self-replicating” aspect, so that when it is applied to an argument (say *n*), it also supplies *itself* to *g*. Let's continue, expanding *g* to the term it names:

$$\begin{aligned}
&g ((\lambda x. g(xx))(\lambda x. g(xx))) c_3 \\
&= g (Y g) c_3 \\
&= (\lambda f. \lambda n. \text{cond (iszero } n) \text{ } c_1 \text{ (mul } n \text{ (} f \text{ (pred } n)))}) (Y g) c_3 \\
&\rightarrow (\lambda n. \text{cond (iszero } n) \text{ } c_1 \text{ (mul } n \text{ ((} Y g \text{) (pred } n))}) c_3 \\
&\rightarrow \text{cond (iszero } c_3) \text{ } c_1 \text{ (mul } c_3 \text{ ((} Y g \text{) (pred } c_3))} \\
&\rightarrow \text{cond ff } c_1 \text{ (mul } c_3 \text{ ((} Y g \text{) } c_2)) \\
&\rightarrow \text{mul } c_3 \text{ ((} Y g \text{) } c_2)
\end{aligned}$$

Now notice that the subterm $\text{redex } (Y\ g)\ c_2$ is just $(\text{fact } c_2)$, so we have calculated that

$$\text{fact } c_3 = \text{mul } c_3 (\text{fact } c_2)$$

which is what we were aiming for.

Exercise 3.6

Complete the calculation of $(\text{fact } c_3)$, both by hand and on the lambda calculus workbench.

We will talk more about fixpoints later in the *Fixpoint Theory of Recursive Function Definitions* component of the course. For now just be convinced that we *can* cope with repetition or recursion in the λ -calculus.

4 The Typed Lambda Calculus

In Mathematics and Computer Science, especially in programming languages, we usually find it helpful to classify values and expressions according to some notion of having a particular *type*. The pure λ -calculus as we have seen it so far could be considered to have a degenerate type system where every term has the same type, say D , and we could write

$$M : D$$

Alone that isn't very interesting, but considering that since every term is a function (and every argument and result is too) we can also write

$$M : D \rightarrow D$$

to indicate M 's functional characteristic. An immediate consequence is that D and $(D \rightarrow D)$ need to be equal (or at least isomorphic), but unfortunately they can't be — they don't even have the same cardinality. This caused some concern: what could a model of the λ -calculus be and in particular is there a “function space” model? The affirmative answer was provided by Dana Scott who took $(D \rightarrow D)$ to be the *continuous* functions from D to D .

4.1 Extending the Typed Lambda Calculus

Let's begin by attempting to classify the *boolean* values as having type **Bool**. In the previous section we represented the boolean value *false* as the combinator $\text{ff} = \lambda xy.y$. It is tempting to say ff has type **Bool** but that just doesn't work. We may be able to *interpret* the behaviour of ff as being like *false* in certain contexts, but in another context the *same term* is the Church numeral c_0 which we would like to classify as being of type **Int**. Furthermore, in general $\lambda xy.y$ is just a function that takes two arguments and returns the second. A similar point can be made regarding *all* the combinators we introduced in our discussion of the pure λ -calculus as a model of computation.

Instead of trying to classify the combinators, what we will do is to introduce a type **Bool** and *new values* to the calculus:

$$\text{true, false} : \mathbf{Bool}$$

We will also need to add (at least) a primitive conditional operation as the cond combinator no longer serves that purpose. Similarly we can introduce a type **Int** along with primitive values and operators to further extend the λ -calculus.

$$0, 1, 2, 3, \dots : \mathbf{Int}$$

4.2 The Typing Relation

We know the type of the primitive **Bool** and **Int** values, but our aim is to construct a type system for the λ -calculus syntactic categories variable, abstraction and application that is well-behaved: for example, if $M : t$ and M reduces to N then $N : t$. The system should also not be too conservative, in the sense that most useful terms will have a type.³

Consider how to go about assigning a type to $\lambda x.M$. We want it to have a *function type* which we will write as

$$\lambda x.M : t_1 \rightarrow t_2$$

where t_2 is the result type and t_1 is the argument type. The result type is therefore the type assigned to M , but to work out that type, there is a dependency on the type assumed for the binder x . An obvious example is the identity function:

$$\lambda x.x : t \rightarrow t$$

In general, the collection of type assumptions for all the free variables is needed as part of the type assignment process. The typing relation is thus a triple:

$$\Gamma \vdash M : t$$

where M is a term, t is the type being assigned to M and Γ is a collection of type assumptions (or *type bindings*) of the form $x : t$. We call Γ a *context* or an *environment*. For simplicity we will gloss over details and require that a context contains at most one binding for any variable. We will write $\Gamma, x : t$ to indicate a context containing the binding $x : t$.

The type assignment system is as follows:

$$\Gamma, x : t \vdash x : t \quad (\text{Var})$$

$$\frac{\Gamma, x : t_1 \vdash M : t_2}{\Gamma \vdash \lambda x.M : t_1 \rightarrow t_2} \quad (\text{Abstr})$$

$$\frac{\Gamma \vdash M : t_1 \rightarrow t_2 \quad \Gamma \vdash N : t_1}{\Gamma \vdash MN : t_2} \quad (\text{App})$$

The *Var* rule says that the type of a free variable x is determined by the corresponding context assumption. It is important to keep in mind that in an abstraction $\lambda x.M$ the *bound* occurrences of the binder x are the *free* occurrences of x in term M . The *Var* rule is an *axiom* and the other two are *inference rules*. As in your earlier studies of natural deduction, the components above the line are *premises* and below the line is the *consequence*. In other words, if we can deduce the premises then the rule allows us to deduce the consequence. As with natural deduction we can present type assignments as derivation trees.

The *Abstr* rule can be read as follows: if, under the assumption that x has type t_1 we can determine the type of M to be t_2 , then the abstraction $\lambda x.M$ has functional type $t_1 \rightarrow t_2$. Notice that the assumption about the type of x is expressed by the context $\Gamma, x : t_1$.

The *App* rule says that if we can determine that M has a functional type $t_1 \rightarrow t_2$ and that N 's type agrees with the argument type of M then the application MN has type t_2 , as

³We could completely evaluate a term and then classify its result, but typing is intended to be a *static* analysis — we want to classify their behaviour *prior to* evaluation.

given by the result type of function M . Explicit in this rule is a requirement that type of the argument provided agrees with the type the function is expecting. If not, we may reject the term as invalid. Essentially, this is the *point* of static typing. Another thing to notice about the *App* rule is that the contexts for typing M and N are both the same Γ . If that were not the case we would be able to have *different type assumptions* for the same free variable. If that were permitted — for example $x : t$ in M and $x : t'$ in N , what should be the type assumption for x in the context for MN ? This is an interesting question which goes beyond the aims of this course. Our focus is on the *simply typed* λ -calculus so we make the straightforward, conservative choice of requiring the contexts to be the same.

As an example of the system in action, here is a derivation of the type of $(\lambda x.x)\mathbf{true}$:

$$\frac{\frac{x : \mathbf{Bool} \vdash x : \mathbf{Bool}}{\vdash \lambda x.x : \mathbf{Bool} \rightarrow \mathbf{Bool}} \text{Abs} \quad \vdash \mathbf{true} : \mathbf{Bool}}{\vdash (\lambda x.x) \mathbf{true} : \mathbf{Bool}} \text{App}$$

As another example, here is a derivation of a type for $\lambda xy.x$:

$$\frac{\frac{x : \mathbf{Bool}, y : \mathbf{Int} \vdash x : \mathbf{Bool}}{x : \mathbf{Bool} \vdash \lambda y.x : \mathbf{Int} \rightarrow \mathbf{Bool}} \text{Abs}}{\vdash \lambda xy.x : \mathbf{Bool} \rightarrow (\mathbf{Int} \rightarrow \mathbf{Bool})} \text{Abs}$$

You may be wondering why we chose **Bool** and **Int** as the types for x and y . It's a good question and one we will consider later.

There are well known algorithms for carrying out these type assignments (credited to Roger Hindley and Robin Milner) which underly the type checking phases of typed functional languages such as ML and Haskell, but we don't have time to go into that here.

4.3 The Curry-Howard Correspondence

As an interesting aside, there is a noteworthy correspondence between simple types and propositional logic. A sequent-style presentation of natural deduction would include rules like:

$$\begin{array}{l} \Gamma, P \vdash P \qquad \qquad \qquad (Ass) \\ \frac{\Gamma, P \vdash Q}{\Gamma \vdash P \supset Q} \qquad \qquad \qquad (\supset-I) \\ \frac{\Gamma \vdash P \quad \Gamma \vdash P \supset Q}{\Gamma \vdash Q} \qquad \qquad \qquad (\supset-E) \end{array}$$

By mapping \supset to \rightarrow , (and eliding the λ -terms) the correspondence with the typing rules is striking: the *Ass* and *Var* rules are the same, as are \supset -I and *Abs*, and \supset -E and *App* respectively. If we also had product and sum types, they would correspond to conjunction and disjunction respectively, and so on.

Furthermore, in *constructive* logics, a proof of a proposition P will consist of positive *evidence* for P . In constructive logics proof by contradiction, or the law of excluded middle — that $Q \vee \neg Q$ is axiomatic — do not hold, so for example, a proof of $P \supset Q$ is a “mechanical procedure” that,

given a proof of P , constructs a proof of Q . In that case, as well as types corresponding to propositions, we may consider λ -terms to correspond to proofs.

The relationship often called the Curry-Howard *Isomorphism* goes even further, with β -reduction corresponding to *proof normalisation*. These observations (and others) are usually attributed to William Howard, Haskell Curry and William Tait.

4.4 Conditional Expressions

We observed in section 4.1 that when adding primitive values and types to the λ -calculus we also need to add some primitive operations on those values — at least a conditional on the booleans and a successor on the numbers. Typing the successor function is obvious:

$$\emptyset \vdash \text{succ} : \mathbf{Int} \rightarrow \mathbf{Int}$$

Typing conditional expressions is a little more interesting. We will add a new syntactic form **if** *_* **then** *_* **else** *_* with evaluation rules:

$$\mathbf{if\ true\ then\ } M \mathbf{\ else\ } N \rightarrow M$$

$$\mathbf{if\ false\ then\ } M \mathbf{\ else\ } N \rightarrow N$$

$$\frac{P \rightarrow Q}{\mathbf{if\ } P \mathbf{\ then\ } M \mathbf{\ else\ } N \rightarrow \mathbf{if\ } Q \mathbf{\ then\ } M \mathbf{\ else\ } N}$$

The third rule puts in place an evaluation strategy that is strict in the boolean expression and non-strict in the two result arms of the conditional. The corresponding typing rule is

$$\frac{\Gamma \vdash B : \mathbf{Bool} \quad \Gamma \vdash M : t \quad \Gamma \vdash N : t}{\Gamma \vdash \mathbf{if\ } B \mathbf{\ then\ } M \mathbf{\ else\ } N : t} \quad (If)$$

The first component B obviously must be a **Bool**, but also notice that both the then and else parts M and N must have the same type. This may seem unduly conservative — it is clear that the expression

$$\mathbf{if\ true\ then\ } 42 \mathbf{\ else\ id}$$

always returns an **Int** so it always has that type. But by the *If* rule, the expression cannot be typed because $42 : \mathbf{Int}$ and $\text{id} : t \rightarrow t$ which must be different types. On the other hand, consider a similar term:

$$\mathbf{if\ } \langle \text{complicated expression} \rangle \mathbf{\ then\ } 42 \mathbf{\ else\ id}$$

Now it may be that the complicated expression always evaluates to **true** or always to **false**, but as suggested in section 4.2 we want type assignment to be a static analysis, not involving evaluation. In that case typing conditional expressions is necessarily conservative, ensuring consistency by requiring both arms of the conditional expression to have the same type.⁴

Exercise 4.1

As a simple exploration of how we might deal with data structures, consider adding *pairs* to the λ -calculus. Use a simple notation like (M, N) and, following Haskell as usual, use the same notation for the types: (t_1, t_2) . You will need rules to type pairs and you will also need projection functions like **fst** and **snd**.

⁴More esoteric type systems, such as *intersection types* allow us a little more flexibility than the simple system we are exploring.

4.5 Type Variables and Polymorphism

In one of the examples in section 4.2 where we deduced the type of $(\lambda x.x)\mathbf{true}$ to be **Bool**, one of the intermediate stages was to assign type **Bool** \rightarrow **Bool** to the identity function. But there is nothing about the identity function itself which restricts its argument and result types to be **Bool** — intuitively, *id* will work for arguments of *any* type. This is the fundamental idea behind *parametric polymorphism*, a concept which you should find familiar from Haskell.

In Haskell, we would assign the identity function the type $a \rightarrow a$ where a is a *type variable* with the natural interpretation that *any type* may be substituted for the variable. The information contained in this type is therefore that *id*'s argument and result are of the same (arbitrary) type.

Being more precise, we can define the syntax of type expressions as follows:

$$t ::= a \mid \mathbf{Bool} \mid \mathbf{Int} \mid t_1 \rightarrow t_2 \mid (t)$$

where a is a type variable, **Bool** and **Int** are two primitive types (we could add others in a similar fashion) and $t_1 \rightarrow t_2$ is a function type, indicating an argument of type t_1 and a result of type t_2 . Parentheses are minimised by (\rightarrow) associating to the right.

In another example in section 4.2 we derived a type **Bool** \rightarrow (**Int** \rightarrow **Bool**) for the term $\lambda xy.x$. Clearly, this function need not be restricted to those argument types — here is another type assignment for $\lambda xy.x$ with type variables instead of **Bool** and **Int**:

$$\frac{x : a, y : b \vdash x : a}{x : a \vdash \lambda y.x : b \rightarrow a} \text{Abs}$$

$$\frac{x : a \vdash \lambda y.x : b \rightarrow a}{\vdash \lambda xy.x : a \rightarrow (b \rightarrow a)} \text{Abs}$$

To treat parametric polymorphism properly we would need to embark on a more detailed discussion, beginning with type substitution and probably going as far as the second-order (polymorphic) λ -calculus, where the type system itself has abstraction and application, parallel to the term language. We don't have time for that but to give you an idea of the issues involved, suppose we define a function

$$\mathbf{cond} = \lambda xyz. \mathbf{if} \ x \ \mathbf{then} \ y \ \mathbf{else} \ z$$

which could be assigned the type

$$\mathbf{cond} : \mathbf{Bool} \rightarrow a \rightarrow a \rightarrow a$$

and then elsewhere use it in an expression

$$\mathbf{cond} \ \mathbf{true} \ 42 \ 0$$

for which we would obviously like to derive type **Int**. To do so would require variable a to be instantiated to **Int**, so a suitable mechanism must be provided. This path would lead us into extending the term language with *let-bindings* and notions of type instantiation and generalisation. If you're interested in exploring this further, I can suggest some starting point readings.

4.6 Recursion

To wrap up our discussion of the λ -calculus we will revisit the implementation of recursion, but this time in the context of the simply-typed λ -calculus.

Unfortunately, the fixpoint combinator Y that we introduced in section 3.5

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

does not exist in the simply-typed λ -calculus because it cannot be assigned a type. Why not? Somewhere in the derivation we would need to deal with the *self-application* sub-term (xx) with the (*App*) rule:

$$\frac{\Gamma \vdash x : t_1 \rightarrow t_2 \quad \Gamma \vdash x : t_1}{\Gamma \vdash x : t_2} \text{App}$$

Since x is a variable, and all three judgements have the same context Γ , it is clear that x can only have one type, so

$$t_1 = t_2 = (t_1 \rightarrow t_2)$$

That is:

$$t = (t \rightarrow t)$$

which only has an infinite solution and is outside our simple type language. In fact *no* fixpoint combinator can be typed. The upshot is that we need to add a *primitive* fixpoint operator in much the same fashion as we did with boolean and integer operators. Without a more thorough treatment of polymorphism it is better to do this by adding a new syntactic form, as we did with **if** _ **then** _ **else** _ in section 4.4. The form will be **fix** M with evaluation rules

$$\begin{aligned} \mathbf{fix} (\lambda x.M) &\rightarrow M[x := \mathbf{fix} (\lambda x.M)] \\ \frac{P \rightarrow Q}{\mathbf{fix} P \rightarrow \mathbf{fix} Q} \end{aligned}$$

A variant on this form that occurs in some of the literature is $\mu x.M$ which is equivalent to $\mathbf{fix}(\lambda x.M)$. The typing rule is

$$\frac{\Gamma \vdash M : t \rightarrow t}{\Gamma \vdash \mathbf{fix} M : t} \text{(Fix)}$$

This may be easier to grasp with an example, so we will revisit the factorial function. The recursive definition we begin with is

$$fact = \lambda n. \mathbf{if} n == 0 \mathbf{then} 1 \mathbf{else} n \times fact(n - 1)$$

which we rewrite as before to

$$\begin{aligned} g &= \lambda f. \lambda n. \mathbf{if} n == 0 \mathbf{then} 1 \mathbf{else} n \times f(n - 1) \\ fact &= \mathbf{fix} g \end{aligned}$$

Observe the types $g : (\mathbf{Int} \rightarrow \mathbf{Int}) \rightarrow (\mathbf{Int} \rightarrow \mathbf{Int})$ and $\mathbf{fix} g : \mathbf{Int} \rightarrow \mathbf{Int}$

Exercise 4.2

Check that these types are correct by building the derivation trees for their type assignments.

The evaluation of $(fact\ 3)$ proceeds as follows:

$$\begin{aligned}
 fact\ 3 &= (\mathbf{fix}\ g)\ 3 \\
 &= \mathbf{fix}\ (\lambda f.\lambda n.\ \mathbf{if}\ n == 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times f(n - 1))\ 3 \\
 &\rightarrow (\lambda n.\ \mathbf{if}\ n == 0\ \mathbf{then}\ 1\ \mathbf{else}\ n \times (\mathbf{fix}\ g)(n - 1))\ 3 \\
 &\rightarrow \mathbf{if}\ 3 == 0\ \mathbf{then}\ 1\ \mathbf{else}\ 3 \times (\mathbf{fix}\ g)(3 - 1) \\
 &= 3 \times (\mathbf{fix}\ g)\ 2
 \end{aligned}$$

and so on. Note that we have left in the function names $fact$ and g for readability only. The whole process could (should) be written without naming any terms. Effectively, applying the \mathbf{fix} operator to g is self-replicating in just the same way as the application of combinator Y in section 3.5.

Finally, it is worth noting that if we had gone through the full details of polymorphism and let-bindings we would be able to define \mathbf{fix} as an operator with type $\mathbf{fix} : (a \rightarrow a) \rightarrow a$.

Exercise 4.3

We can define a recursive function to test whether natural numbers are even as follows:

$$\begin{aligned}
 even &= \lambda n.\mathbf{if}\ (n == 0)\ \mathbf{then}\ \mathbf{true} \\
 &\quad \mathbf{else}\ \mathbf{if}\ (n == 1)\ \mathbf{then}\ \mathbf{false} \\
 &\quad \mathbf{else}\ even\ (n - 2)
 \end{aligned}$$

Express this function using \mathbf{fix} and evaluate an expression such as $(even\ 3)$.