Foundations of Functional Programming

A C Norman, Easter Term 2006

Part IB

1 Introduction

This course is organised into several sections. It is intended to act as a bridge between the Part IA Foundations of Computer Science course (which covers the language ML and its use) and more advanced coverage in the Part II Types course. In some ways it takes a view related either parts of Computer Design or to Computation Theory: in both of those courses it is shown how complicated programming tasks can be achieved by building layer upon layer starting from some very primitive-seeming model (be it the machine code of a real computer or a Turing machine). Other parts of this course lie within the theory stream of the Computer Science Tripos and are concerned with careful use of notation, formal derivations and some theory which is in fact amazingly delicate (and *much* harder to get right than one would possibly have imagined).

(Pure) Functional Programming is a style that treats a rather mathematical view of a function as the essential building block for a programming language. Perhaps the most important characteristic of a function in this sense is that it has no memory and invoking it does not cause any side-effects. It is a function if the *only* thing you can get out of it is a result when you provide it with an argument, and if you give the same argument on different occasions you are guaranteed to get the same result. This is a purist view of functions. You might note that when ML introduces the keyword `ref` and all that goes with it it is going beyond this pure interpretation of functional programming. Some people would consider the ML exception-handling facilities as going beyond "functional" while others would be happy to accept it. This course starts from the more fundamentalist position!

(Extended) functional programming has been a serious thread within our field for a long time. Perhaps Lisp, created by John McCarthy in the late 1950s is the first big landmark. Lisp provided a collection of imperative (ie non-functional) facilities, but some of its users found that it was not just possible but convenient and helpful to de-emphasise their use. It is possible to use today's common-or-garden programming languages (such as C and C++, Java and even Perl, Python and all the rest, as well as the older fashioned or not-taught-here Fortran, PL/I, Ada, Visual Basic etc) can be used in more-or-less functional ways, but to date I see functional programming in the pure sense as having been something that has influenced language facilities and programmer style but which has not fully taken over the main-stream. There are a variety of languages where the functional ideals have been central to the language design. These languages can be and have been used for the construction of serious applications. I will just mention Haskell and Miranda here alongside the core parts of ML. "Functional" is generally seen as an alternative to "imperative" and the type systems that go with it are from a different family from those associated with object oriented languages.

Study of what could be done with functions started before computers existed.

If asked to think of a function it is possible that the sort of example that might spring to mind would be along the lines of writing (in ML syntax)

```
fun f x = x + 1;
```

and then saying that `f` was a function. The purist view dislikes this for two reasons:

1. The example given involves things like "+" and "verb.1." as well as the function we were trying to concentrate on. If you are *really* intent on concentrating on functions perhaps you should not allow yourself to talk about anything else at all! This line of thought leads to an investigation of whether things we normally find built into programming languages (such as numbers and arithmetic) are really needed or whether functions in the purest possible sense can fill in for them somehow;

2. The special ML syntax using the word `fun` seems close to a cheat[1]. Two complaints arise. One is that making the function definition that way suggests you are going to use it later. The process of making a definition and later retrieving it is being relied upon somehow. This goes outside the world of pure functions and so is to be avoided if possible! The second issue is that the function defined above has been given a name (`f`) and that can both be seen as inelegant and as an extra complication if you want to ask when two functions are the same. For instance after the following:

   ```
   fun g y = 1 + y;
   ```

   are the functions `f` and `g` the same in any useful sense? And if so what proof rules or techniques cane be used to show it?

To some extent both of these points are taken care of by concentrating on the notation for anonymous functions, as in:

```
fn x => x;
```

but while in most ML programs such notation is an occasional feature, pure functional programming views these lambda-expressions as central. Having deemed them central it is perhaps then permissible to introduce syntactic sugar that lets programmers go back to writing something much closer to what they used to be used to!

The theory behind functional programming has several branches. It is obviously necessary to defined the precise meaning of all notations used, giving rules

[1] Maybe theorists do not like fun? Maybe they like to make their own!

for just what transformations and proofs are valid. In this case two central issues involve the circumstances in which a computation is deemed to have completed and in deciding just when the results of two different computations should be counted as equal, and indeed when one might be able to predict in advance that they must end up equal.

The computational power of pure functions turns out to be quite high. This course discusses several ways of organising functions into a model of computation that is also practical and convenient, and explains how features you are used to having built into ordinary programming languages fit in or do not. A slight concern for practicalities will intrude at this stage!

Those concerned with modelling computation through functions fairly rapidly discovered that un-constrained use of functions as building blocks gave them too much expressive power. From the perspective of writing a program to perform a task excess power may not be too bad, but if you also want to analyse what is going on and prove properties it is a menace. Type schemes were brought in as the answer: an ideal type scheme would satisfy four objectives:

1. All programs that you legitimately want to write should be expressible within the constraints of the type-checking;

2. All programs that are wild or that cheat or are too complicated for their own good should be rejected as violating type constraints;

3. The process of type-checking should be reasonably easy to explain and should appear to users to be consistent. Type-violations should be reported to the user in a clear and obvious way;

4. It should be possible to type-check any program (correct or faulty) in a reasonable amount of time.

The type-checker in ML gives a pretty good illusion of meeting these ideals. However the seemingly effortless way in which it works conceals the fact that the ML scheme represents an amazingly delicate balance between capability and feasibility. This course will explain the ML type-checking algorithm and thus provide some motivation for next year's coverage of some of the alternative schemes that have been investigated.

So overall the aims of this course are:

- To show how lambda-calculus and related theories can provide a foundation for a large part of practical programming.

- To present students with one particular type analysis algorithms so that they will be better able to appreciate the Part II Types course.

- To provide a bridge between the Part IA Foundations of Computer Science course and the theory options in Part II.

Its official syllabus, which gives a model for the order in which topic will be covered, shows it split into three parts. The first two of these will use around 5 lectures each while the final part is around two lectures:

Part A. The theory

A.1 Introduction. Combinators. Constants and Free Variables. Reduction. Equality. the Church-Rosser theorem. Normal forms.

A.2 The Lambda calculus. Lambda-terms, alpha and beta conversions. Free and bound variables. Abbreviations in the notation. Pure and applied lambda calculi. Relationship between combinators, lambda calculus and typical programming languages.

A.3 Encoding of data: booleans, tuples, lists and trees, numbers. The treatment of recursion: the Y combinator and its use.

A.4 Modelling imperative programming styles: handling state information and the continuation-passing style.

A.5 Relationship between this and Turing computability, the halting problem, recursive functions etc.

Part B. Implementation techniques

B.1 Combinator reduction as tree-rewrites.

B.2 Conversion from lambda-calculus to combinators.

B.3 The treatment of lambda-bindings in an interpreter: the environment.

B.4 Closures. ML implementation of lambda-calculus. SECD machine.

B.5 Brief survey of performance issues.

Part C. Type Reconstruction

C.1 Let-polymorphism reviewed following the Part IA coverage of ML.

C.2 Unification. A type-reconstruction algorithm.

C.3 Decidability and potential costs.

and finally the objectives[2] at the end of the course students should

[2]These items are all listed here in this tedious and pedantic manner because current guidelines seem to insist on being almost unbearably explicit about what is going on, possibly to the extent that it conflicts with actually getting down to the teaching!

- Understand the rules for the construction and processing of terms in the lambda calculus and of Combinators;
- Know how to model all major aspects of general-purpose computation in terms of these primitives;
- Be able to derive ML-style type judgements for languages based upon the lambda-calculus .

When I took over this course I found it unexpectedly hard to produce obvious suggestions for relevant background reading. The two main suggestions from previous years, Hindley and Seldin[1] and Revesz[2] have now both gone out of print. Various much older books I investigated are again either out of print or much too detailed (to say nothing of expensive!) to be useful for a 12 lecture course in the Easter Term. Thus I find I have to fall back on the above two suggestions, which can at least be found in libraries. It will sometimes also help if you look back at Larry Paulson's book on ML from last year. If anybody identifies a book or web-resource that is especially useful (and for choice cheap) I would really like to know.

2 Combinators

Historically the first carefully developed and worked-though study of what could be done if all you had was functions introduced the idea of a *combinator*. A combinator is really nothing more than one of some agreed set of functions that one takes as given. Expressions are built up out of combinators using *application*, so if you have any two expressions E_1 and E_2 then $(E_1 E_2)$ will also be considered a valid expression, denoting E_1 applied to an argument E_2. The primitive combinators that are used introduce various reduction rules. For instance it will be common to declare that there is a combinator **K** that denotes a function such that one can perform a reduction

$$(\mathbf{K}\ x)\ y \rightarrow x$$

for any expressions x and y.

The combinators used here have simple unconditional re-write rules of this style as their *only* properties. A pure combinator theory looks as what can be done using just such combinators and nothing else. An applied theory also allows for the introduction of *constants* in expressions. For instance the numbers **1**, **2**, ...might be deemed to exist as constants and then combinatory forms such as $(\mathbf{K}\ \mathbf{1})$ can be written. Some people will then add further constants such as + with whole rafts of new valid reductions such as

$$(+\ \mathbf{2})\ \mathbf{3} \rightarrow \mathbf{5}$$

A reasonable way of understanding when something is called a combinator and when it is called a constant (even if it is something like +) is that combinator reduction rules are always valid and what happens does not depend on the identify and nature of the arguments are all. In the study of functions via combinators it will generally be considered that the theory or proofs about how constants work is beyond our concern.

An amazing result about combinators (that you may have seem at least brief mention of in the Part IA course?) is that if you have the combinator **K** as defined above together with another one **S** such that

$$((\mathbf{S}\ f)\ g)\ x \rightarrow (fx)(gx)$$

then all possible computations can be expressed using expressions built up out of applications involving just **S** and **K**. I will show some details of this later.

In my explanation of combinator I have used some variables, x, y, f and g. These are purely part of the explanation and my descriptive notation. Within a combinator expression you do not have any variables at all. As you will see rather soon this is a rather good thing since the proper and careful understanding of variables introduces rather a lot of extra complication.

The only thing explained about combinators so far is that each basic combinator comes with a reduction rule, notated with "$\rightarrow$". The challenge is now to imagine that some (perhaps large) tree of applications has been built (all the leaf elements will be combinators) and now the question is "what can be achieved by performing some of these reductions?". The notation "$\Rightarrow$" is used to talk about the effect of making a sequence of the primitive reductions, so we say $E_1 \Rightarrow E_2$ if there is some way of applying the combinator rules to E_1 that eventually converts it into $E2$. For instance $\mathbf{S\ K\ K\ S} \Rightarrow \mathbf{S}$ since if I subscript the combinators[3] so you can see more readily how they move I can do two primitive reductions:

$$\mathbf{S}_1\mathbf{K}_1\mathbf{K}_2\mathbf{S}_2 \rightarrow \mathbf{K}_1\mathbf{S}_2(\mathbf{K}_2\mathbf{S}_2) \rightarrow \mathbf{S}_2$$

When you do a series of reductions and find that no more are possible you have a *normal form*. There are two especially important variations on this idea. The first is to continue reductions for so long as there is any sub-expression in your formula that could be reduced. If, when you keep doing this, you eventually find that there is nothing more that can be done then you have a normal form. An alternative is to agree that you can stop when you can not perform a *head* reduction: this would be one involving the leftmost combinator in the expression. Stopping at this stage gives you a head-normal form. If you have a sub-expression

[3] And from now on I will be assuming left association of function application, so f a b is to be interpreted as (f a) b.

X that blows up and reduction on it never terminates then $\mathbf{K}\ X$ does not have a full normal form (because you would be keeping on trying to reduce X), but it is already in head normal form. This area of just what a normal form is and when you can obtain one is slightly more delicate than you might have imagined so I will need to say a little more about it later.

Now given two expressions (perhaps both in normal form) in what circumstance should one declare that they are equal? Somebody coming to this from a computation theory background might reasonably say that combinators and expressions built up using them are all functions, and so two functions are equal if they behave the same way whatever arguments they are given. This is *not* the traditional view taken by functional programming people. They take the view that equality of behaviour is undecidable and that they want to work with things that are as definite as possible. Thus the most basic view is that two expressions are the same if their printed representations are identical[4].

This interpretation of equality means one has to be pretty careful at times. In particular the fact that you "know" that two expressions mean the same is not enough to let you declare them equal! The next variation on equality is to define two expressions to be equivalent if you can transform one to the other by a sequence of reductions and inverse reductions. In terms of Part IA Discrete Mathematics, let two expressions M and N be related if there is a single direct reduction $M \rightarrow N$. Then the transitive closure of this relation shows when an expression can be reduced to another in perhaps many steps. The symmetric & reflexive closure of *this* is then the smallest equivalence relation that contains the reduction step, and is the one we use to stand for equality.

It would be nice to have a set of rules to discover what sequence of reductions to perform and what the intermediate expressions were!

The term *extensional* equality is used to describe the case where two expressions become equal if applied to sample arguments. For instance $\mathbf{K}\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})$ and $\mathbf{S}\ \mathbf{K}$ are both in normal form and they are visibly different. I assert that they are not equal. However if you consider arbitrary values x and y then

$$\mathbf{K}\ (\mathbf{S}\ \mathbf{K}\ \mathbf{K})\ x\ y \rightarrow \mathbf{S}\ \mathbf{K}\ \mathbf{K}\ y \rightarrow \mathbf{K}\ y\ (\mathbf{K}\ y) \rightarrow y$$

and

$$\mathbf{S}\ \mathbf{K}\ x\ y \rightarrow \mathbf{K}\ y\ (x\ y) \rightarrow y$$

and so the expressions are extensionally equal.

There are perhaps three fundamental results about combinators:

[4]Note a useful feature of combinators – there are no names for formal parameters to worry about. In a more messy world one would have to add lots of special treatment to say that `fun f x = x` and `fun f y = y` were defining the same function because the choice of name for the formal parameter was not supposed to matter

Universality: In the same sense that a Turing Machine or a Register Machine is universal, reduction of combinator expressions based on just **S** and **K** represents a universal model of computation. later in this course I will be showing how to perform mappings from a convenient and sensible notation to achieve this. Part of what is implied by this is that some combinator expressions will not have normal forms – any attempt to reduce them will loop or explode;

Normal Order Reduction: Given a big combinator expression it will usually be the case that there are many different reductions that could be done within it. When trying to reach a normal form you could start with any one of these. In some cases there will be ways of selecting reductions that just introduce more possible reductions and if you persist in making such selections you may *never* reach a normal form. However if you always select the leftmost outermost of all possible reductions (this strategy is known as *normal order reduction* then if there is any way to reach a normal form at all you will reach it;

Church-Rosser: If a combinator expression can be reduced to a normal form at all then any two different ways of achieving this will end up giving you the same result. This result (the Church-Rosser property for combinator forms) may sound obvious and you might expect its proof to be easy, but it turns out to be slightly slippery!

3 The Lambda calculus

A while after the study of combinators started Alonzo Church invented the lambda-calculus. In this a function is written using the notation $\lambda\ x\ .\ A$ where x is now used as a name for the argument of the function and A is its body – an expression usually involving x. This is the notation for (anonymous) functions that ML notates as `fn x => A`. A lambda-expression is now a constant (as for the constants used with combinators, and pure lambda terms will be ones that do not use any of these), a variable name (generally written as a lower case letter), a lambda expression $\lambda\ x\ .\ E$ or an application $E_1 E_2$.

A variable mentioned just after the symbol λ is *bound* throughout the body of the function concerned. Any use of a variable that is not bound is described as reference to a *free* variable. Thus in the expression

$$\lambda\ f\ .\ f\ x$$

f is a bound variable and x is a free variable.

There are two things one can do with a lambda-expression:

α conversion: This just renames a bound variable. If you had $\lambda\ x\ .\ x$ you could α convert it into $\lambda\ y\ .\ y$;

β reduction: Given an application where the function part is a lambda expression you can perform a β reduction. The expression $(\lambda\ x\ .\ A)\ B$ turns into whatever you get by replacing every x present in A by B.

Actually beta-reduction is distinctly more delicate than that because now there are names for variables there is a nasty possibility of name clashes. So the simple explanation of β reduction given above is proper provided there are no clashes or conflicts. And if necessary α conversion can be used to rename variables before trying the reduction to make sure that this is so. Let me give an example of an expression where naive substitution could mess up scope and get bindings wrong:

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

here the outside argument y (free in this part of the expression) is moved in to be within the scope of the inner lambda. In such cases it is important to use α conversion to rename variables to avoid clashes, for instance by adjusting things to read

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

where no ambiguity can possibly arise.

A lambda-expression is in normal form if no sub-expression can be reduced. It is in head-normal form if the outermost application can not be reduced. This means that a head-normal lambda expression has the form

$$\lambda\ x_1\ x_2\ \ldots\ x_n\ .\ v\ E_1\ \ldots\ E_m$$

where v is some variable.

Lambda calculus has the Church-Rosser property in that again if two different ways of performing reductions both lead to normal forms then the two results will differ by at worst alpha-conversion.

The example

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

shows that performing sequences of beta reductions may not manage to reduce the size of an expression: in this case reducing it turns it back into itself. The notation Ω is sometimes used for expressions that do not have a normal form because their reduction never terminates. Lambda calculus seems to be one of the most awkward and un-natural notations ever invented! It is thus *very* useful to introduce various short-hand forms. It is normal to write $\lambda\ a\ b\ c\ .\ A$ for $\lambda\ a\ .\ (\lambda\ b\ .\ (\lambda\ c\ .\ A))$. As usual the application $f\ a\ b$ means $(f\ a)\ b$. Even with these sorts of short-hand

the ordering of the items you have to write when composing lambda expressions often seems awkward and confusing. However one can draw a parallel between pure lambda calculus and the notation present in quite comfortable programming languages and base on this have a nicer way of writing things.

Consider the introduction of a local variable in a language akin to ML[5]. One writes something like

```
let v = <some value>
in <some expression using v> end
```

I will interpret this as standing for the lambda expression

$$(\lambda\, v \text{ . some expression using v})\ \text{some value}$$

and looking at β-reduction you can see that the intent in each case is much the same!

Now consider a simple[6] function definition

```
let f x = body
in ... f(arg) ... end
```

which in the same style I can view as just a syntactic sugaring for

$$(\lambda\, f \text{ . } (f\ \text{arg}))\ (\lambda\, x \text{ . body})$$

in which on lambda is used to describe the function itself while the other captures the idea that the name f should be associated with the function.

The key to functional *programming* is contained in these two little translations. The programming-language-like syntax used is really tolerably close to ML, and as the Part IA Foundations of Computer Science course showed that provides a quite convenient basis for writing real programs. But the notation is also just a re-arrangement of lambda calculus, which is spartan and (one hopes!) susceptible to mathematical study and analysis, so looking at programming this way can perhaps provide a really solid basis for understanding what programs mean and do.

When using the programming-language-like notation for lambda calculus it is important to avoid unwittingly including things that lambda calculus does not do directly. For instance a pure functional programming language would not have numbers or any other sort of data in it. The only thing it can work with will be lambda expressions. Thus the results of evaluating a fragment of program will

[5] I will not use the exact syntax of ML here in part to remind you that I am really talking about lambda calculus!

[6] Here I am not going to permit the definition of recursive functions, and I insist that the definition has a limited scope.

be a lambda expression (there is nothing else it could be!). Issues of possible evaluation orders, of normal forms and of the interpretation of "equality" arise in very much the same way as they do for combinators. Except that dealing with the fact that the names of bound variables ought to be considered unimportant but that name clashes have the potential to foul things up adds a fair amount of extra pain.

4 Encoding of data

From the point of view of programming pure lambda calculus starts off looking very weak and weedy. The purpose of this section is to demonstrate that it is in fact astonishingly powerful and that pretty well all the things you are used to in ML can be modelled in it rather easily and with only a few lines of code! You may have seen some of these examples already, but I collect them together here so you see them all at once.

4.1 Boolean values and tests

I will model the values **true** and **false** by the lambda expressions $\lambda\ a\ b\ .\ a$ and $\lambda\ a\ b\ .\ b$. Then I can map the programming language construct

```
if x then y else z
```

onto just $x\ y\ z$

Suppose x is **true** then this returns y, while if x is **false** is returns z. So for now on think of all programs as being prefixed with the text

```
let true a b = a
in let false a b = b
in let if x y z = x y z
in ...
```

and agree that a proper number of `end` marks will be stuck on the end. Note that I have (slightly) extended the syntax that I allow in my "programming language" to permit function definitions to have many arguments, but this just maps onto the lambda-calculus shorthand for the same thing and it does not count as cheating.

4.2 Tuples

```
let pair a b f = f a b;
in let left p = p true
in let right p = p false
in ...
```

Now `pair A B` will create a lambda expression such that the functions `left` and `right` will retrieve `A` and `B` from it. You might note that the main trick being used here is just like the Curried functions you saw in ML.

The data-type *list* must give you the possibility to have an empty list or a non-empty one, and non-empty lists have two components. I can model this by using nested pairs with an explicit boolean value to indicate whether I have an empty or non-empty list:

```
let nil = pair true <anything>
in let isempty list = left list
in let cons hd tl = pair false (pair hd tl)
in let first list = left (right list)
in let rest list = right (right list)
in ...
```

gives you a constant `nil` to use as an empty list, a test `isempty` to identify it, a function `cons` to build non-empty lists and selectors `head` and `tail` to extract their components. The text `<anything>` above can be replaced by an arbitrary expression (maybe `true`) since nobody should ever access it.

Trees and the like can be build in a way similar to that used to construct lists.

At this stage it is perhaps proper to introduce a remark about evaluation order. In the ML course you were introduced to lazy lists as a special programming technique. In functional programming one can consider two philosophies. One chooses to evaluate function applications by reducing the argument first and then performing the β reduction that substitutes it into the body of the function it is being passed to. This evaluation order is essentially the one that ML uses. Another possibility is to use normal order reduction (leftmost outermost reduction first). This latter policy is perhaps less convenient to implement efficiently, but it causes programs to run and yield a result if any imaginable evaluation order will. If you have a functional programming language that uses normal order reduction (which I will typically be supposing here) then examples such as

```
if true 1 <something that blows up>
```

will compute to completion in a nice way (and such examples are pretty common in quite ordinary programs) and all lists will act as if "lazy" in the sense of the Part IA course without any need for any special action.

The next thing to consider is the introduction of recursive function definitions. Simple lambda expressions represent simple functions, but the notation

```
let f x = ... f x' ...
```

which seems so natural in the programming-language notation needs some special treatment. The key issues is that in the body of the function the function's name (`f`) must be available as a bound variable. Thus the first stage is to try writing

$$\lambda\, f\, .\, \lambda\, x\, .\, \ldots\ f\, x'\ \ldots$$

to go as the main part of the function. The $\lambda\, x$ indicates that we have a function of one argument (x) and the outer lambda make the inner f into a bound variable. Overall this gives us a function that would become just what we wanted em if it could be provided with what we wanted as its argument! As a magician[7], finding a rabbit in one of my hats, I introduce the fix-point operator **Y** which I want to satisfy the identity $\mathbf{Y}\ f\ =\ f\ (\mathbf{Y}\ f)$. Suppose for a short while that such a function **Y** can exist. Then the transformation to

```
let f = Y (lambda f . (lambda x . ... f x' ...))
```

turns out to be exactly what is needed to model our recursive function.

Let me give a more concrete example (a rather traditional one):

```
let fact n =
    if n = 0 then 1 else n * fact(n-1)
```

becomes

```
let fact = Y (lambda fact . (lambda n .
    if n = 0 then 1 else n * fact(n-1)))
```

Now imagine expanding out the use of **Y**. The effect is to replace the inner call to `fact` with what used to be the whole function definition. After a couple of expansions one would see

```
let fact = lambda n .
   if n = 0 then 1 else n * (
      lambda n .
         if n = 0 then 1 else n * (
            Y (lambda fact . (lambda n .
            if n = 0 then 1 else
             n * fact(n-1))))(n-1))(n-1)
```

One reasonable interpretation is that the recursion is being copied with by simply copying the function's own definition to within itself. The result is a notionally infinite function definition in much the same way that lazy lists can be notionally

[7] Or as one who had seen this topic covered under the heading of Compiler Construction earlier in the year!

infinite. What happens in practise is that the **Y** operator unwinds things as much as is needed for a particular call to the function concerned but hopefully never has to go on for an infinite way.

Now the question is can **Y** be expressed in terms of pure lambda calculus. Consider

```
let Y f =
   let g h = f (h h)
   in g g end
in ...
```

and first observe that this definition is *not* recursive: it expands into

$$\mathbf{Y} = \lambda\, f\, .\, (\lambda\, g\, .\, g\, g)\, (\lambda\, h\, .\, f\, (h\, h))$$

quite harmlessly. But now `Y f` expands to `g g`, which in turn becomes `f (g g)` which is equal to `f (Y f)`. It also expands further, if you like, to an infinitely nested set of calls: `f (f (f (f (...))))`. This tricky little lambda expression is enough to let you define arbitrary recursive functions!

Finally for this section I will show how it is possible to model the integers.

Let me first introduce a little extra notation. I will write $f \circ g$ for the functional composition of f and g. To show that this is not cheating I will write

```
let compose f g x = f (g x)
in ...
```

Now I will choose to represent the natural numbers as by lambda expressions as follows:

```
let identity x = x;
in let zero f  = identity
in let one f   = f
in let two f   = f o f
in let three f = f o f o f
in let four f  = f o f o f o f
in ...
```

With this representation of whole numbers addition and multiplication are astonishingly easy, and as a joke exponentiation is also trivial! Note that if n is a number than $n\ f$ represents the function f composed with itself n times. The proper thing to represent zero is then related to the identity function.

```
let add n m f = (n f) o (m f)
in let multiply m n f = m (n f)
in let expt m n = n m
in ...
```

The behaviour of the exponentiation function may be unexpected but there have been previous Part IA questions based on the above so I hope you can work through what is going on and understand it for yourselves.

The next thing needed is a way to test if a number is zero.

```
let K x y = x
in let ifzero n a b = n (K b) a
in ...
```

where if n is non-zero the result will be at least one application of **K** and the result will be b, while if $n = 0$ the result will be a.

At this stage you might like to stand back and observe the *amazing* power and expressiveness of lambda calculus and functional programming. Starting from almost nothing we have built up models of booleans, tuples, lists, conditionals and now integers and each new definition has only bee around a (short) line long.

Subtraction of natural numbers is a curious operation because sometimes the proper result would be negative, and negative numbers are not natural. Thus it should not be too amazing that the functional code for subtraction is distinctly curious. The neatest version I have come across is based on defining a predecessor function. This is the *concept*:

```
let inc n f = f o (n f)
in let predecessor n = n inc <minus-one>
in ...
```

where the predecessor of a value n is obtained by incrementing "-1" a suitable number of times. The problem here is that we can not have a proper lambda-representation for -1 that is really consistent with our understanding of numbers as things that compose a function with itself[8].

So consider

```
let specialinc n = ifzero n (inc n) (inc n)
in let predecessor n =
    n specialinc ?
in ...
```

where I have left "?" where the "-1" will go. What I then want to do is to select a value to insert in place of the question mark so that `specialinc ?` turns into `zero`. Try it!

```
   specialinc ?
-> ifzero ? (inc ?) (inc ?)
-> ? (K (inc ?)) (inc ?)
```

[8] -1 would find the inverse of a function and that is too much to ask for in general.

and **hah Bingo!** I can insert $\lambda\ p\ .\ \lambda\ q\ .$ zero in place of the question mark. It just ignores its two arguments and returns `zero`. Thus I end up with just

```
let specialinc n = ifzero n (inc n) (inc n)
in let predecessor n =
    n specialinc (lambda p . lambda q . zero)
in ...
```

This may be pretty devious but it works! Now given that we have a predecessor function and we can also write recursive functions it is not too hard to implement subtraction, magnitude comparisons, division, remainder and all the rest. If you need integers (rather than just natural numbers) you might model them as tuples with a boolean value for the sign and a natural number for the magnitude. If you want floating point than you just need a tuple to hold an exponent and a mantissa. Even there the code will not be too lengthy.

A further issue that needs making but which I intend to gloss over somewhat is the treatment of mutually recursive sets of functions. This can be handled by using the `pair` function to make a tuple out of all the functions that are to be defined together, and then using the **Y** operator to fix this. The details often become rather grubby but it can be done.

One can worry about the performance expectations of all this simulation, but I will return to that topic later on. For now my concern is to get across the point that lambda calculus is not just a foundation for programming but it seems to be a quite natural and convenient one. Perhaps you can see how having seen the way in which data can be modelled functionally a lot of people got quite enchanted with the idea that all their programming should be in a pure functional style. And a different idea also grew: perhaps the best way of building a really solid theoretical basis for practical programming languages would be to relate their properties to something as spartan and tractable as lambda calculus through these layers of modelling and simulation.

Of course

5 Modelling imperative programming

When somebody sets up an agenda saying that they will write all their code in a functional style it is natural for others to come up with challenges. An important instance of such a challenge is that of programs where the order of evaluation matters and where this shows up because executing code fragments may have side-effects. Related to this is to treatment of control structures: loops, goto statements and exception handling.

Sorting all this out leads to something that is pretty messy at times, but in a course on the foundations of functional programming it seems proper to explain how it can be done. I think this is not very useful for hand-written code but there are two ways in which it can be of practical relevance. The first is to transform imperative code to a pure functional form if that is thought liable to help you perform formal transformations or verification on it. The other is as part of a compiler: very competent compilers have been written using in effect pure lambda calculus as the internal representation for programs and using the translations and mappings given here to convert arbitrary code into that form.

I will first look at code where you want to specify the order of evaluation. Well in fact with pure functional programming it may be impossible to force much in that respect, but it *is* possible to re-structure code so that an intended order of evaluation is explicit and visible. The technique used is known as *continuation passing*.

One interpretation of this is to look at the way in which function calls are compiled for a traditional computer. The arguments are prepared and put somewhere (typically in registers or on a stack), a return address is established and then control is transferred to the function. When it has finished its work it prepares a return value and jumps back to the location indicated by the return address. With a slightly different viewpoint this can become a lot more symmetric. Think of returning from the function as calling the return address as a new function, passing your return value as its argument. And then thing of the return address as a perfectly ordinary argument to the original function, not as something special.

It now becomes possible to take ordinary function calls and re-write them in a functional language so you can see all this happening. The code tends to look a bit messy and contorted, but once you get used to it is is not too bad and it is certainly no worse than looking at how one maps high level languages down onto machine code. Here is an example, with first the original version of a function:

```
let f a b = g(h a, k b)
in ...
```

and now a version expanded to use continuation passing:

```
let f a b cont =
   h a (lambda r1 .
         k b (lambda r2 .
            g r1 r2 cont))
in ...
```

You may note that each function defined now has an extra final argument which will be its “continuation” (ie what to do with its result), and the code I have shown

visibly starts by calling `h` to compute `h a`, which it passes to a continuation that accepts the result and calls it `r1`. Next `k` is called and the result it produces ends up as `r2`. Finally `g` is called, and the result from g is passed to whatever continuation `f` had been given. Here is the same function translated so that it calls `k b` before `h a`:

```
let f a b cont =
   k b (lambda r2 .
         h a (lambda r1 .
            g r1 r2 cont))
in ...
```

The continuation passing style makes order of evaluation and the need for temporary storage very clearly explicit (which is why it can be useful within a compiler). It also makes it easy to talk about recursive and iterative (sometimes known as *tail recursive*) calls, in that the stack space that is implicit in direct use of a programming language becomes explicit here each time a new lambda-expression has to be set up to be an intermediate continuation.

The real excitement with continuation passing is that you can model quite complicated control structures very readily. A location in your code that might ever be reached has to be represented by a function, and passing that as a continuation allows you to direct control to there. In a few cases it can be useful to pass a function several alternative continuations, for instance one to activate in normal circumstances and another to use as an exception exit. This scheme makes it possible to model `goto`, `try` and `throw` in terms of the purest lambda-calculus!

Now what about side-effects? Well the view here is that the effects must be to change something, so a copy of that something can be passed along as an extra argument and when it is changed an updated version will be passed at the next function call. This is very much what is happening with accumulator arguments in a typical iterative style of ML programming. When you combine this with continuation passing you see that every function ends up with two extra (implicit?) arguments: one its continuation and another a data-structure that represents the current state of all updateable state that your program may need. If this state is very large and messy this can be painful, but for many programs it will not be.

There have been a number of proposals for modelling input and output (which tend to count as side-effects) into functional programming. many of these cheat somewhat! One view is to say that a functional program can be given a list of all characters it will ever ask to read as its argument, and it will generate as its output a list consisting of printing and other directives. A surrounding framework can then look at the output it produces and cause things to happen.

6 Computability

Maybe by now it will be 100% obvious that pure functional programming is computationally universal. However just to make things explicit consider a Register Machine. Suppose it has two registers, a and b, and its states are labelled 0, 1, ... I can produce a lambda-calculus program that models it very easily. If in some state n the transition is to increment the a register and go to state m I will define a function

```
let fn a b = fm (inc a) b
in ...
```

while if the node tests a, decrements if non-zero and goes to p or q my definition will be

```
let fn a b = ifzero a (p a b)
                      (q (predecessor a) b)
in ...
```

Obviously similar code applies if it is the b register that changes. Starting and stopping the register machine does not introduce anything more that is at all complicated.

I will later show how anything expressed in lambda-calculus can be expanded out in terms of the combinators **S** and **K** and that will establish that they are computationally universal too. This of course carries with it the result that there is no computable way of telling if a given lambda expression has a normal form.

One might like to see how to encode lambda expressions or combinatory forms in such a way that a Turing Machine could reduce them. But first I should talk about implementation techniques for when you have an ordinary computer to use!

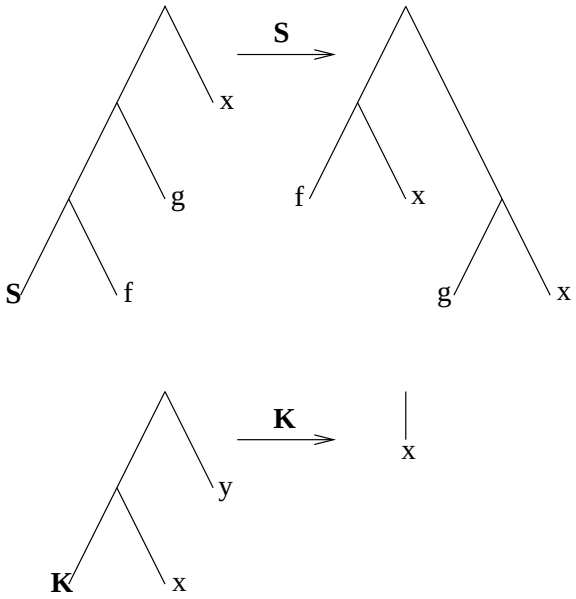
7 Implementing Combinator reduction

The basic combinator re-write rules are just

$$\begin{array}{rcl} \mathbf{K}\ x\ y & \rightarrow & x \\ \mathbf{S}\ f\ g\ x & \rightarrow & f\ x\ (g\ x) \end{array}$$

A natural way to represent combinator expressions within a computer will be as trees[9], and then the re-writes become graph transformations which can be shown as follows:

[9] Actually even if things start off as trees they are liable to end up with shared sub-structure and possibly with loops, so really they are directed graphs.



To perform reduction on a graph of combinators one scans down the leftmost chain of edges. If it does not end then your reduction does not terminate! If it does then what is at its end must be either **S** or **K**, since no other leaf elements exist! In each case you check to see if there were enough branches above it to make one of the above transformations possible, and if so you re-write the tree. Note that for **S** this involves allocating a couple of new nodes of store for the new branches. Then scan again to find the new leftmost combinator. When the head combinator does not have enough arguments to reduce the graph has been brought to head normal form. Anything done to if beyond this will not alter the leading combinator. But if desired the sub-trees can be reduced to get the whole expression into normal (not just head-normal) form.

It might be noted that locating the head combinator that is to be activated and keeping track of the chain of links leading to it perhaps seems to call for recursion. When people write real combinator reducers there is a high probability that they will avoid need for this extra memory by reversing the chain of pointers as they scan down the graph and then putting it back into its original configuration as they unwind.

Here seems a good place to comment again about evaluation orders and options. Applicative order evaluation arranges to reduce arguments to all functions to their normal form before passing them into the body of the function. The combinator reduction code given above does not do that naturally, but simple direct lambda-calculus reducers can find this the easiest scheme to implement. Normal order reduction is when at each stage the leftmost outermost reduction is performed. Even if it is harder to implement it has the benefit that it guarantees that a program will not loop if there is *any* evaluation order that avoids getting

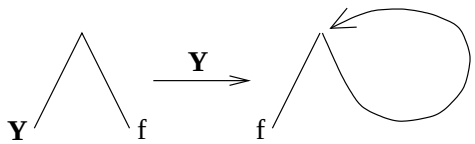
stuck. However here is an example of normal order evaluation being obviously inefficient:

$(\lambda\, x\, .\ +\ x\, x)$ messy expression $\rightarrow +$ messy expression messy expression

and now the messy expression will be evaluated twice. Applicative order would (of course) have evaluated it just once before substituting its result into the function body.

A third scheme (which has a more pragmatic flavour) is *lazy evaluation* which is just like normal order evaluation except that it keeps track of where arguments get substituted into a body. Then if such an expression is evaluated once (for any reason) it is arranged that all the other references to the expression end up pointing to the resulting value. The effect is that an arguments is never evaluated more than once. In terms of the graph re-write for a reduction triggered by **S** as shown above it is very easy to arrange this! All you do is to make the re-written graph over-write the top node in the original one. Thus is several parts of the entire program share references to the sub-expression with the **S** reduction in it the first one to call for it to be reduced gets that job done on behalf of all the rest. One should note that over-writing the input graph here seems to be an important side-effect and suggests that the combinator reducer itself will want to be written in an imperative language not a (pure) functional one.

When using graph-rewrites and combinators some cases can be dealt with especially neatly. Notably the **Y** operator can be built in as a primitive combinator using the re-write



which creates a looped-up structure but feels a very simple operation to have to perform to implement something that at first appeared to be rather messy and hard to understand. This combinator implementation suggests you can think of **Y** as the basic operator for creating loops in graphs!

8 Conversion from lambda-calculus

Basic conversion from lambda-expressions to combinators is very easy, but when done in the simplest possible way the result is a very serious expansion in bulk. It is still useful to explain the basic method. First and as a matter of convenience introduce a third combinator **I** with re-write rule

$$\mathbf{I}\, x \quad \rightarrow \quad x$$

and note that (**S K K**) could have been used in place of **I** if one were feeling really bonded to the full primitive experience. Now take any lambda expression (and suppose that all the variable in it are bound). Use the transformations

$$\begin{array}{rcl} \lambda\, x\, .\, x & \rightarrow & \mathbf{I} \\ \lambda\, x\, .\, y & \rightarrow & \mathbf{K}\, y \\ \lambda\, x\, .\, (f\ g) & \rightarrow & \mathbf{S}\ (\lambda\, x\, .\, f)\ (\lambda\, x\, .\, g) \end{array}$$

where the first rule that applies in any case is the one that is to be used. The final rule reduces the number of applications forming the body of any lambda expression, and must eventually reduce things until all that is left is a simple variable as the body. Then one of the first two rules will kick in and the lambda disappears. These rules should be applied first to the innermost lambda expressions present in the input (and will then remove this lambda in favour of a big mess of combinators).

It is jolly easy to write a program to apply the above three rules, and when you have you can try it out. What you will find is that even quite short fragments of lambda-calculus expand into horrible large strings of combinators. But this problem is not an essential part of the process. But adding just a few more combinators and a few simple optimisation rules to the translation it is possible to reach a stage where the bulk of combinators generated is directly proportional[10] to the size of the lambda expression you started with.

The translation between lambda-calculus and combinators is not without some worry. In general the translation process will preserve extensional equality but it will not guarantee to preserve the sort of equality that lambda-calculus experts are most keen on. Furthermore small variations and optimisations (and optimisation is very much needed to make this conversion useful!) all affect this. In general if you take a lambda expression M and another N such that $M \rightarrow N$ and convert M into a combinator form M' and N into N' you can not be quite certain that $M' = N'$ in the combinator world (until you allow for extensional equality). Hmmmm.

There are a whole raft of slightly different combinator conversion procedures, and the issue of *exactly* how the combinator reductions you end up with relate to what happened in pure lambda calculus is again one of those issues that is more delicate than you would like it to be. However here is a set that behaves reasonably well!

[10] There are some delicacies here! Sufficiently messy lambda expressions will have to use a large number of distinct names for the variables that they introduce, and proper measurement of their "bulk" must account for this. For instance if there are n variables then you need $\log_2(n)$ bits to specify which you are referring to. To achieve linear grown the combinator translation process also has to take special action on some ill-balanced inputs, and I am not describing the details here.

$$
\begin{array}{rcl}
\mathbf{K}\ x\ y & \rightarrow & x \\
\mathbf{S}\ f\ g\ x & \rightarrow & f\ x\ (g\ x) \\
\mathbf{I}\ x & \rightarrow & x \\
\mathbf{B}\ f\ g\ x & \rightarrow & f\ (g\ x) \\
\mathbf{C}\ f\ x\ y & \rightarrow & f\ y\ x \\
\mathbf{S'}\ z\ f\ g\ x & \rightarrow & z\ (f\ x)\ (g\ x) \\
\mathbf{B'}\ z\ f\ g\ x & \rightarrow & z\ f\ (g\ x) \\
\mathbf{C'}\ z\ f\ x\ y & \rightarrow & z\ (f\ y)\ x
\end{array}
$$

And then the compilation rules get elaborated with

$$
\begin{array}{rcl}
\mathbf{S}\ (\mathbf{K}\ x)\ (\mathbf{K}\ y) & \rightarrow & \mathbf{K}\ (x\ y) \\
\mathbf{S}\ (\mathbf{K}\ x)\ y & \rightarrow & \mathbf{B}\ x\ y \\
\mathbf{S}\ (\mathbf{K}\ x)\ \mathbf{I} & \rightarrow & x \\
\mathbf{S}\ x\ (\mathbf{K}\ y) & \rightarrow & \mathbf{C}\ x\ y \\
\mathbf{S}\ (\mathbf{B}\ x\ y)\ z & \rightarrow & \mathbf{B'}\ x\ y\ z \\
\mathbf{Y}\ (\mathbf{K}\ x) & \rightarrow & x \\
\mathbf{I}\ x & \rightarrow & x \\
\mathbf{B}\ \mathbf{I}\ x & \rightarrow & x \\
\mathbf{B}\ x\ \mathbf{I} & \rightarrow & x \\
\mathbf{B}\ (\mathbf{B}\ x\ y)\ z & \rightarrow & \mathbf{B'}\ x\ y\ z \\
\mathbf{C}\ (\mathbf{B}\ x\ y)\ z & \rightarrow & \mathbf{C'}\ x\ y\ z \\
\mathbf{C}\ + & \rightarrow & + \text{ (etc)} \\
\lambda\ x\ .\ x & \rightarrow & \mathbf{I} \\
\lambda\ x\ .\ y & \rightarrow & \mathbf{K}\ y \\
\lambda\ x\ .\ (f\ g) & \rightarrow & \mathbf{S}\ (\lambda\ x\ .\ f)\ (\lambda\ x\ .\ g)
\end{array}
$$

Here the idea is that the last few rules are exactly as before, but the various earlier ones are used as simple rewrites (performed as the tree of combinators is built up) that spot special cases of the use of the more general combinators and convert them into uses of more specialised ones. If these rules are used the factorial function gets converted as follows:

```
fun fact n = if n = 1 then 1 else n * fact(n-1);
=> Y (B (S (C' if (= 1) 1)) (B (S *) (C B (C - 1))))
```

where this is using an applied combinator system with built-in numbers, “if” test and arithmetic.

9 Lambda-bindings in an interpreter

The delight about using combinators is that the implementation does not have to worry about the names of variables. However one should also understand how

to evaluate lambda-expressions directly. The most obvious approach would be to implement code that performed beta-reductions directly: a substitution process. The big pain with this is avoiding trouble with unwanted name-capture due to clashes in the names of bound variables. The normal way out of this trouble is to keep an *environment* that records the values associated with all currently-bound variables. The details of how this can be implemented can be found in notes from the Compiler Construction course, and so will not be repeated here.

10 Closures

When you pass a complete lambda-expression around there may not be any special problems. However within a lambda-reducer you will be working with fragments of lambda expression and very many of those will have free variables. These will of course normally be variables that are bound by some enclosing construct. When such a sub-expression is passed around it is important that extra information be included with it to carry this context information. Such extra information is generally called the *environment* and the combination of an expression and its associated environment is a *closure*. A particular case where closures are needed is when functions are returned as results, for instance as in the ML code

```
fun f x =
   fn y => x + y;
```

where the lambda expression `fn y => x + y` must be returned together with an environment that shows the value for `x` that it is supposed to be using.

An ML implementation of lambda-calculus was shown to you earlier in the year. The ML implementation tends to be a recursive function. Sometimes it is useful to have an explanation of how to reduce lambda expressions that shows all memory use quite explicitly. One such scheme is to use what is known as an SECD machine. SECD stands for State, Environment, Control and Dump. It represents a style of abstract machine with those four components. The state is an expression that is in the process of being reduced. The environment is a list of pairs: the first component of each pair is the name of a variable and the second component is a value that the variable has. The environment is always searched in such a way that the topmost binding of a variable is the one retrieved. Control and Dump are used as stacks to track the context in which an expression is being evaluated: popping items off these stacks reveals information about what is to be done with the result of evaluating a sub-expression.

All actions taken by an SECD machine are triggered by recognising the symbol or simple pattern in the State and Control, and each step makes a simple change moving information between the parts of the machine. If you are ever

going to implement an interpreter for some sort of (mostly) functional language, and especially if your implementation will need to support tail-recursion, closures and the like it is probable that you should think in SECD terms about what you will be doing.

11 Performance

If you have a programming language that looks somewhat like ML then *most* of the code that people write will not be very tricky, and even though it may be expressed in different ways much of it will be asking for just the same sorts of computation that an equivalent program in C, C++ or Java would have. Specifically (iterative) ML code to compute factorials and the ordinary ML code to build and use binary trees and lists is really not very different from the corresponding code in any other language. The same will be true for other functional languages, and in consequence a *sufficiently clever compiler*[11] can generate excellent code for them. If the functional language is, by its specification, lazy then this too need not hurt too badly, since there are techniques (discussed in the Part II course on optimising compilers) that can spot the large number of places where the administrative overhead of supporting laziness is not needed. So the party line is that even the purest functional language could deliver realistic performance. In fact users of such languages are often much more concerned with correctness and elegance than they are with absolute speed, but the literature contains a reasonable trail or work where people have striven to extract the largest grain chunks from functional code that they can such that the chunks can be subjected to all the tools of modern compiler optimisation.

12 Let-polymorphism

One thing we have seen is that pure lambda-calculus (or combinators) can be used to model almost anything else in programming that we might want to use. Once (for instance) we have shown that numbers and arithmetic can be simulated this way it seems fair to put them into our programming system as primitive operations, to avoid the cost an inconvenience of having to run the simulation. After all we have proved that including them does not make the programming model any more complicated to analyse, since all the complication was already available via the simulation! But when we do that it seems *very* desirable to encapsulate the native implementation of the data-type. When everything was being modelled

[11] A mythical entity, often invoked by those who are championing languages that in reality seem to go a bit slowly!

as a function you could try applying any object to any other one and you would get some sort of defined effect. If you make numbers primitive items you would really like to insist that attempts to use numbers as functions or to apply "+" to non-numbers should not be permitted. The fact that pure lambda-calculus also allows you to synthesise your own pretty wild control structures (specifically the **Y** operator and its friends) renders it too powerful. Perhaps we would like a more restricted system where the user is explicitly permitted to make recursive function definitions but defining private variants on **Y** is prohibited.

The mechanism that has been developed to impose this sort of discipline is type-checking. In some sense it tends to be a purely destructive add-on. The user writes a functional program, and passes it through a type-checker. Sometimes the type-checker moans and rejects the code as "invalid". If the type-checker does not moan you just execute the program be doing ordinary lambda-reduction. So type-checking just flags some programs (that you could have tried before) as dodgy. A good type system will object to programs that in your heart of hearts you can agree are really a bit dodgy, and it will always endorse nice tidy code.

The simplest forms of type system assign a fixed, definite type to each value that you work with. Thus each variable or sub-expression in your program will be marked as being an integer, or a function from integers to lists or whatever. Such schemes are not too hard to set up but tend to feel excessively restrictive in use.

The style of type-checking most commonly used with functional languages is *polymorphic*. This means that the type ascribed to a function or expression can involve *type variables*. The key problem in polymorphic type checking is that of working out when to introduce these variables. There are higher order styles of type-checker where the type-variables can themselves be constrained or typed. I will not consider such cases here!

The type-judgement scheme provided in ML mostly seems effortless and natural when you first use it. Treated informally it is usually possible to work out what type a function should be given, and the procedure you use will be a close relative of the algorithm that ML in fact uses. However there are some fine points that generally do not show up when you are just a user of the type-checker.

For this course I will note that ML gives special treatment to arithmetic and its type analysis does some distinctly funny things with integers and real numbers. An effect of its curiosity is that code such as

```
fun double x = x + 1;
```

needs extra annotation to help resolve whether the addition is of integers or reals. I will ignore this problem here by thinking in terms of programs where all numbers are integers. The ad-hoc polymorphism that deals with arithmetic is really not especially interesting!

What is interesting is the way of generating the types that contain type variables. For instance after making the ML definition

```
fun S f g x = f x (g x);
```

you will be told that **S** has the type

$$(\alpha \to (\beta \to \gamma)) \to ((\alpha \to \beta) \to (\alpha \to \gamma))$$

This type-expression is a general one: any of the type variables in it can be replaces by any more restrictive type to get something more concrete, and in general when **S** is used it will act as in one of the more concrete ways. A property of ML type-checking is that every expression that has a type at all can be given an unique most-general type, and all other types it can ever be viewed as having can be obtained by substituting for type-variables in the general version. This apparently obvious property may not be present in all possible type reconstruction schemes!

One curiosity about ML-style type reconstruction is that it that transformations on expressions that ought not to change their value at all can change their type-checking status. For instance start with the lambda expression (here written in ML syntax)

```
(fn i => (i 3, i "s")) (fn x => x);
```

and perform a β-reduction on it to obtain

```
((fn x => x) 3, (fn x => x) "s");
```

In ML the first of these will refuse to type-check while the second will be entirely happy! Perhaps this fact will make it clear why people are interested in looking at alternative type schemes but you should be aware that striking a balance between type reconstruction power and the difficulty of finding types is astonishingly hard. The ML scheme seems to be a clear cut examples of a “Which” type “best buy”.

13 Unification and type reconstruction

The ML type-reconstruction method is explained here with more care than its coverage in the Part IA course, but still somewhat informally. What is perhaps more critical is that here I will show what has to be done (well enough that you could possibly implement it) but I will not even start to provide a proof that the algorithm that I describe extracts exactly the type expressions that are wanted. In general type reconstruction is a delicate enough process that until somebody has defined the algorithm as a carefully and formally specified set of rules and worked

through proper proofs of its properties one ought to be very suspicious. What I hope is that an sketch of the method here will make its more precise coverage next year easier to approach.

Type reconstruction uses one significant sub-algorithm: unification. You should already be familiar with this from its use in Prolog. In this case we want a variant on unification that takes two trees, each of which is starting to represent a type. Within these trees the nodes will denote constructors such as "$\rightarrow$" that indicates that the type is the type of a function and some of the leaves will be definite fixed types such as `int` and `string`. Other leaves (in each tree) may be logical variables. Unification attempts to match the two trees against one another, instantiating logical variables where necessary in the process. When a variable is instantiated the value it is given must not refer back to itself. In the context of Prolog you may have heard this called the *occurs check* and ignored it. In type-checking you want to make sure this condition is checked for. When unification succeeds it will often have had a side-effect of instantiating several more logical variables. If any of the unifications attempted during type reconstruction fail (either because of a gross miss-match or because of occurs-check trouble) then you declare that the expression does not have a valid type.

If you are not especially worried about efficiency it is really quite easy to implement unification. With quite large amounts of care unification can be performed in an amount of time linear in the bulk of the two inputs. When I come to talk about the cost of ML-style type reconstruction I will suppose that you have gone to the (significant) trouble of implementing linear-cost unification: if you want to code something to try I suggest you just write the obvious code – it will run quite fast enough!

The eventual object of type reconstruction is to ascribe a type to a *complete* functional program. But the strategy used is to work from the bottom up giving types to various little fragments of code first. These types will often need to be refined later on when the context in which the fragment exists is examined. The scheme that supports this is the use of logical variables. So let me first indicate the (initial) type judgement you make for leaf elements in an expression.

If the leaf is a constant then the nature of the constant defines its type instantly. Thus **true** has type `boolean`, **17** has type `integer` and `+` has[12] type `int->int->int`.

When a variable is seen it is associated with a type expression that is a new uninstantiated logical variable. The issue of treating the binding of variables will be discussed later under "lambda" and "let". Specifically the behaviour when there are several references to the same name within one scope depends on whether the variable concerned was bound using `lambda` or `let`. Getting a

[12]Remember that I am ignoring the issue of `int` vs. `real` arithmetic

precise explanation of just how to avoid mix-ups with names and scopes in lambda calculus is (as usual) a much more messy business than it deserves to be. Since this is a Part IB course not a Part II one I will not write out formal rules for how to do it and I will not show exactly where you need to perform re-naming substitutions to untangle things. I count these as "mere" technicalities, even though in an implementation it is important to get them all right!

There re are four remaining basic constructions that can appear in a parse tree:

if: When an expression `if` b `then` E_1 `else` E_2 you form the type expressions for each of the sub-trees. You unify the type of b with `boolean` and unify the types of E_1 and E_2 with each other. The result of this last unification is then the type of the whole expression;

application: If the application is A B and treating A and B as separate values leads to type-expressions τ_A and τ_B then create two new logical variables p and q. Unify τ_A with $p \rightarrow q$. Unify τ_B with p. Return q as the constructed type for the application;

lambda: If you have `fn` v `=>` E then you ensure that the logical variable for the type of v is the same as all the ones used for v throughout E. This can be done by unifying all these! Suppose it is τ_v and also suppose that the type you construct for E is τ_E. Then the type for the lambda expression is just $\tau_v \rightarrow \tau_E$. Actually a natural way to implement scope rules will be to build up a map of bound variables and their associated type (logical) variables as you recurse down into an expression, and then to do the unifications that I am describing as you return from analysing each sub-expression. That way as you start to analyse a sub-expression you will have available to you information about all bound variables;

let: The rule for `let` v `=` E_1 `in` E_2 is the key one in ML type reconstruction. Note that top-level variable and function definitions are treated as if they are introduced with `let` clauses. You reconstruct a type for E_1. Now if there are any uninstantiated logical variables in this type you leave them as type-variables and obtain a polymorphic type for v, τ_v. Now at each place in E_2 where there is reference to v you make its type into a copy of τ_v but with a fresh set of logical variables in each place that v occurs.

The rule for conditionals is pretty dull and obvious. The one for applications is the main one that actually does work. But the key thing that makes this particular type derivation special is the difference it makes between

```
(fn v => B) A;
```

and

```
let v = A in B;
```

In the first case the complete expression is analysed as a whole so all uses of v must be consistent throughout B, and the eventual type that gets unified into existence for v depends on all of A and on all of the contexts within B that v is used in. In the second case the type given to v depends *only* on the expression A, and type variables in this expression are turned into fresh logical variables before use. Consider once again the example

```
let fun i x = x
in (i 3 , i "s") end
```

which I will interpret as meaning (in a cruder syntax)

```
let i = fn x => x
in (i 3 , i "s")
```

Here in the `let` clause `i` will get give the type $\alpha \rightarrow \alpha$ where α is a type variable. After the word `in` the variable α will have to be turned into two new logical variables for the two uses of `i` And one of these logical variables will end up unified to `int` while the other ends up as `string`.

14 Decidability and costs

A good thing about ML type-checking is that it is decidable. As with so many observations in functional programming this seems a pretty silly thing to say – you are likely to think that *of course* it is decidable. Well various attempts at just slightly more general or powerful type-checking schemes lead to type reconstruction where you can write down a lot of equations to be solved (like the equations we solved in the last section through the use of unification) but where there can not be a general algorithm that guarantees to find a solution.

Experience with the ML system also shows that type-checking of all real programs is pretty cheap. It therefore came as a slightly nasty surprise that the algorithm (even when very carefully implemented, using data structures that share as much as possible) has exponential worst case. As a result there are quite short ML programs that would type-check if you had unlimited resources but where the type reconstruction will not complete on any real machine. The bad constructions are relatives of the following:

```
let f1 y = (y,y)
in let f2 y = f1 (f1 y)
in let f3 y = f2 (f2 y)
in let f4 y = f3 (f3 y)
...x
```

where you should note that `f1` makes a tree with two leaves, and each subsequent line squares the number of leaf nodes present. This leads to types which if written out linearly would, at the nth line have around 2^{2^n} symbols! Because the types concerned have a rather regular shape it is possible to represent them in a way that shares a huge number of sub-trees, but things can be set up so that one can show that even with maximal structure sharing the type you end up with is of size 2^n. If the type is that large then any algorithm that produces it must take exponential time.

If you do not like my use of tuples as in `(y,y)` above or would rather restrict yourself to lambda expressions not visible function definitions then it is easy to expand out the above example in terms of the more pure styles of functional programming and the same bad results apply. Indeed there is much to be said in this example for making the tuple using

```
let pair x y z = z x y
```

since that leads to the ML types of the functions `f1` etc involving ridiculously increasing numbers of distinct type variables!

15 Practical back-up for this course

The section of the lab's teaching web pages that relate to this course contains a file `parser.jar` that I have prepared. It is not a fully stable and fully debugged program (apologies) but may be fun or useful. The source that I used to build it is also present on the web page as a `.zip` archive. I built and have tested it using JDK 1.2.2 on Windows NT. If any of you take a copy and make any interesting improvements it would be nice if you passed thgem back to me giving me full permission to do whatever I like with your corrections and enhancements. The code is available for any use you may need to make of it in association with the Computer Science courses here, but I am retaining copyright and anybody who sees a way to use it commercially is invited to talk to me before going too far.

To use it go `java -jar parser.jar` and then type in input to its prompt. The syntax that you use is similar to that of ML but the language uses lazy evaluation and it does not impose any type-checking.

The code may well have bugs. No: it *will* have bugs! Any piece of program that large is bounnd to have some. The syntax that it accepts is only documented in the file `small.cup`, but it is much like ML. The code works by converting things to combinators and then doing graph reduction on what results. It gives lazy evaluation. At the time of writing these notes it does not do any type-checking at all, but I intend to add ML-type polymorphic checking, but then if an expression fails to type-check I will still allow you to evaluate it! I hope it is interesting and

useful! Reading the Java code may show you something about practical aspects of (one way of) implementing functional languages, while just trying it out lets you see the sample code that models numbers etc working.

The following output has been edited in a few places to help things fit on lines of the page.

```
% java -jar parser.jar

2 + 3;
Input expression + 2 3
=> 5


y f;
Input expression y f
=> f (f (f (f (f (f (f (f (f (f (f (f
(f (f (f (f (f (f (f (f (f (f (f ...


fun double x = x + x;
Statement FUN double x = + x x
Combinator form: S + I


double 7;
Input expression double 7
=> 14


fun fact n = if n = 1 then 1 else n * fact(n-1);
Statement FUN fact n =
          IF (= n 1) 1 (* n (fact (- n 1)))
Combinator form: Y (B (S (C' IF (= 1) 1))
                 (B (S *) (C B (C - 1))))


fact 7;
Input expression fact 7
=> 5040
```

```
fun compose f g x = f (g x);
Statement FUN compose f g x = f (g x)
Combinator form: B


fun zero f x = x;
Statement FUN zero f x = x
Combinator form: K I


fun one f = f;
Statement FUN one f = f
Combinator form: I


fun add n m f = compose (n f) (m f);
Statement FUN add n m f = compose (n f) (m f)
Combinator form: S' B


fun multiply m n f = m (n f);
Statement FUN multiply m n f = m (n f)
Combinator form: B


fun expt m n = n m;
Statement FUN expt m n = n m
Combinator form: C I


val two = add one one;
Combinator form: S' B I I


val four = multiply two two;
Combinator form: B (S' B I I) (S' B I I)


fun display n = n (fn x => x + 1) 0;
Statement FUN display n = n (+ 1) 0
Combinator form: C (C I (+ 1)) 0
```

```
display one;
Input expression display one
=> 1


display two;
Input expression display two
=> 2


display four;
Input expression display four
=> 4


display (expt four two);
Input expression display
=> 16


fun ifzero n f g = n (k g) f;
Statement FUN ifzero n f g = n (kk g) f
Combinator form: C C’ K


fun inc n f = compose f (n f);
Statement FUN inc n f = compose f (n f)
Combinator form: S B


fun specialinc n = ifzero n (inc n) (inc n);
Statement FUN specialinc n =
          ifzero n (inc n) (inc n)
Combinator form: S (S (C C’ K) (S B)) (S B)


fun pred n = n specialinc (fn f => fn g => zero);
Statement FUN pred n = n specialinc (K (K zero))
Combinator form: C (C I (S (S (C C’ K) (S B))
                 (S B))) (K (K (K I)))
```

```
display (pred four);
Input expression display (pred four)
=> 3


fun ff n = ifzero n one (multiply n (ff (pred n)));
Statement FUN ff n =
          ifzero n one (multiply n (ff (pred n)))
Combinator form: Y (B (S (C (C C' K) I)) (B (S B)
                 (C B (C (C I (S (S (C C' K)
                 (S B)) (S B))) (K (K (K I)))))))


display (ff four);
Input expression display (ff four)
=> 24
```

You might like to observe how the combinator forms for some things are exceptionlly compact: multiplication is just **B** while increment is just **S B** for the Church numerals. The final example computes factorials in terms of pure functions.

References

[1] J R Hindley and J P Seldin. *Introduction to Combinators and Lambda-Calculus*. Cambridge University Press, 1986.

[2] G E Revesz. *Lambda Calculus, Combinators and functional programming*. Cambridge University Press, 1988.