Introduction to Functional Programming

John Harrison
jrh@cl.cam.ac.uk

3rd December 1997

Preface

These are the lecture notes accompanying the course *Introduction to Functional Programming*, which I taught at Cambridge University in the academic year 1996/7.

This course has mainly been taught in previous years by Mike Gordon. I have retained the basic structure of his course, with a blend of theory and practice, and have borrowed heavily in what follows from his own lecture notes, available in book form as Part II of (Gordon 1988). I have also been influenced by those who have taught related courses here, such as Andy Gordon and Larry Paulson and, in the chapter on types, by Andy Pitts's course on the subject.

The large chapter on examples is not directly examinable, though studying it should improve the reader's grasp of the early parts and give a better idea about how ML is actually used.

Most chapters include some exercises, either invented specially for this course or taken from various sources. They are normally intended to require a little thought, rather than just being routine drill. Those I consider fairly difficult are marked with a (*).

These notes have not yet been tested extensively and no doubt contain various errors and obscurities. I would be grateful for constructive criticism from any readers.

John Harrison (`jrh@cl.cam.ac.uk`).

Plan of the lectures

This chapter indicates roughly how the material is to be distributed over a course of twelve lectures, each of slightly less than one hour.

1. **Introduction and Overview** Functional and imperative programming: contrast, pros and cons. General structure of the course: how lambda calculus turns out to be a general programming language. Lambda notation: how it clarifies variable binding and provides a general analysis of mathematical notation. Currying. Russell's paradox.

2. **Lambda calculus as a formal system** Free and bound variables. Substitution. Conversion rules. Lambda equality. Extensionality. Reduction and reduction strategies. The Church-Rosser theorem: statement and consequences. Combinators.

3. **Lambda calculus as a programming language** Computability background; Turing completeness (no proof). Representing data and basic operations: truth values, pairs and tuples, natural numbers. The predecessor operation. Writing recursive functions: fixed point combinators. Let expressions. Lambda calculus as a declarative language.

4. **Types** Why types? Answers from programming and logic. Simply typed lambda calculus. Church and Curry typing. Let polymorphism. Most general types and Milner's algorithm. Strong normalization (no proof), and its negative consequences for Turing completeness. Adding a recursion operator.

5. **ML** ML as typed lambda calculus with eager evaluation. Details of evaluation strategy. The conditional. The ML family. Practicalities of interacting with ML. Writing functions. Bindings and declarations. Recursive and polymorphic functions. Comparison of functions.

6. **Details of ML** More about interaction with ML. Loading from files. Comments. Basic data types: unit, booleans, numbers and strings. Built-in operations. Concrete syntax and infixes. More examples. Recursive types and pattern matching. Examples: lists and recursive functions on lists.

7. **Proving programs correct** The correctness problem. Testing and verification. The limits of verification. Functional programs as mathematical objects. Examples of program proofs: exponential, GCD, append and reverse.

8. **Effective ML** Using standard combinators. List iteration and other useful combinators; examples. Tail recursion and accumulators; why tail recursion is more efficient. Forcing evaluation. Minimizing consing. More efficient reversal. Use of 'as'. Imperative features: exceptions, references, arrays and sequencing. Imperative features and types; the value restriction.

9. **ML examples I: symbolic differentiation** Symbolic computation. Data representation. Operator precedence. Association lists. Prettyprinting expressions. Installing the printer. Differentiation. Simplification. The problem of the 'right' simplification.

10. **ML examples II: recursive descent parsing** Grammars and the parsing problem. Fixing ambiguity. Recursive descent. Parsers in ML. Parser combinators; examples. Lexical analysis using the same techniques. A parser for terms. Automating precedence parsing. Avoiding backtracking. Comparison with other techniques.

11. **ML examples III: exact real arithmetic** Real numbers and finite representations. Real numbers as programs or functions. Our representation of reals. Arbitrary precision integers. Injecting integers into the reals. Negation and absolute value. Addition; the importance of rounding division. Multiplication and division by integers. General multiplication. Inverse and division. Ordering and equality. Testing. Avoiding reevaluation through memo functions.

12. **ML examples IV: Prolog and theorem proving** Prolog terms. Case-sensitive lexing. Parsing and printing, including list syntax. Unification. Backtracking search. Prolog examples. Prolog-style theorem proving. Manipulating formulas; negation normal form. Basic prover; the use of continuations. Examples: Pelletier problems and whodunit.

Contents

Chapter 1

Introduction

Programs in traditional languages, such as FORTRAN, Algol, C and Modula-3, rely heavily on modifying the values of a collection of variables, called the *state*. If we neglect the input-output operations and the possibility that a program might run continuously (e.g. the controller for a manufacturing process), we can arrive at the following abstraction. Before execution, the state has some initial value σ, representing the inputs to the program, and when the program has finished, the state has a new value σ' including the result(s). Moreover, during execution, each command changes the state, which has therefore proceeded through some finite sequence of values:

$$\sigma = \sigma_0 \to \sigma_1 \to \sigma_2 \to \cdots \to \sigma_n = \sigma'$$

For example in a sorting program, the state initially includes an array of values, and when the program has finished, the state has been modified in such a way that these values are sorted, while the intermediate states represent progress towards this goal.

The state is typically modified by *assignment* commands, often written in the form `v = E` or `v := E` where `v` is a variable and `E` some expression. These commands can be executed in a sequential manner by writing them one after the other in the program, often separated by a semicolon. By using statements like `if` and `while`, one can execute these commands conditionally, and repeatedly, depending on other properties of the current state. The program amounts to a set of instructions on how to perform these state changes, and therefore this style of programming is often called *imperative* or *procedural*. Correspondingly, the traditional languages intended to support it are known as imperative or procedural languages.

Functional programming represents a radical departure from this model. Essentially, a functional program is simply an expression, and execution means evaluation of the expression.[1] We can see how this might be possible, in gen-

[1]Functional programming is often called 'applicative programming' since the basic mecha-

eral terms, as follows. Assuming that an imperative program (as a whole) is deterministic, i.e. the output is completely determined by the input, we can say that the final state, or whichever fragments of it are of interest, is some function of the initial state, say $\sigma' = f(\sigma)$.[2] In functional programming this view is emphasized: the program is actually an expression that corresponds to the mathematical function f. Functional languages support the construction of such expressions by allowing rather powerful functional constructs.

Functional programming can be contrasted with imperative programming either in a negative or a positive sense. Negatively, functional programs do not use variables — there *is* no state. Consequently, they cannot use assignments, since there is nothing to assign to. Furthermore the idea of executing commands in sequence is meaningless, since the first command can make no difference to the second, there being no state to mediate between them. Positively however, functional programs can use functions in much more sophisticated ways. Functions can be treated in exactly the same way as simpler objects like integers: they can be passed to other functions as arguments and returned as results, and in general calculated with. Instead of sequencing and looping, functional languages use recursive functions, i.e. functions that are defined in terms of themselves. By contrast, most traditional languages provide poor facilities in these areas. C allows some limited manipulation of functions via pointers, but does not allow one to create new functions dynamically. FORTRAN does not even support recursion at all.

To illustrate the distinction between imperative and functional programming, the factorial function might be coded imperatively in C (without using C's unusual assignment operations) as:

```
int fact(int n)
{ int x = 1;
  while (n > 0)
   { x = x * n;
     n = n - 1;
   }
  return x;
}
```

whereas it would be expressed in ML, the functional language we discuss later, as a recursive function:

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

nism is the *application* of functions to arguments.

[2]Compare Naur's remarks (Raphael 1966) that he can write any program in a single statement $Output = Program(Input)$.

In fact, this sort of definition can be used in C too. However for more sophisticated uses of functions, functional languages stand in a class by themselves.

1.1 The merits of functional programming

At first sight, a language without variables or sequencing might seem completely impractical. This impression cannot be dispelled simply by a few words here. But we hope that by studying the material that follows, readers will gain an appreciation of how it is possible to do a lot of interesting programming in the functional manner.

There is nothing sacred about the imperative style, familiar though it is. Many features of imperative languages have arisen by a process of abstraction from typical computer hardware, from machine code to assemblers, to macro assemblers, and then to FORTRAN and beyond. There is no reason to suppose that such languages represent the most palatable way for humans to communicate programs to a machine. After all, existing hardware designs are not sacred either, and computers are supposed to do our bidding rather than conversely. Perhaps the right approach is not to start from the hardware and work upwards, but to start with programming languages as an abstract notation for specifying algorithms, and then work *down* to the hardware (Dijkstra 1976). Actually, this tendency can be detected in traditional languages too. Even FORTRAN allows arithmetical expressions to be written in the usual way. The programmer is not burdened with the task of linearizing the evaluation of subexpressions and finding temporary storage for intermediate results.

This suggests that the idea of developing programming languages quite different from the traditional imperative ones is certainly defensible. However, to emphasize that we are not merely proposing change for change's sake, we should say a few words about why we might prefer functional programs to imperative ones.

Perhaps the main reason is that functional programs correspond more directly to mathematical objects, and it is therefore easier to reason about them. In order to get a firm grip on exactly what programs mean, we might wish to assign an abstract mathematical meaning to a program or command — this is the aim of *denotational semantics* (semantics = meaning). In imperative languages, this has to be done in a rather indirect way, because of the implicit dependency on the value of the state. In simple imperative languages, one can associate a command with a function $\Sigma \to \Sigma$, where Σ is the set of possible values for the state. That is, a command takes some state and produces another state. It may fail to terminate (e.g. `while true do x := x`), so this function may in general be partial. Alternative semantics are sometimes preferred, e.g. in terms of *predicate transformers* (Dijkstra 1976). But if we add features that can pervert the execution sequence in more complex ways, e.g. `goto`, or C's `break`

and `continue`, even these interpretations no longer work, since one command can cause the later commands to be skipped. Instead, one typically uses a more complicated semantics based on *continuations*.

By contrast functional programs, in the words of Henson (1987), 'wear their semantics on their sleeves'.[3] We can illustrate this using ML. The basic datatypes have a direct interpretation as mathematical objects. Using the standard notation of $[\![X]\!]$ for 'the semantics of X', we can say for example that $[\![\text{int}]\!] = \mathbb{Z}$. Now the ML function `fact` defined by:

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

has one argument of type `int`, and returns a value of type `int`, so it can simply be associated with an abstract partial function $\mathbb{Z} \to \mathbb{Z}$:

$$[\![\text{fact}]\!](n) = \begin{cases} n! & \text{if } n \geq 0 \\ \bot & \text{otherwise} \end{cases}$$

(Here $\bot$ denotes undefinedness, since for negative arguments, the program fails to terminate.) This kind of simple interpretation, however, fails in non-functional programs, since so-called 'functions' might not be functions at all in the mathematical sense. For example, the standard C library features a function `rand()`, which returns different, pseudo-random values on successive calls. This sort of thing can be implemented by using a local static variable to remember the previous result, e.g:

```
int rand(void)
{ static int n = 0;
  return n = 2147001325 * n + 715136305;
}
```

Thus, one can see the abandonment of variables and assignments as the logical next step after the abandonment of `goto`, since each step makes the semantics simpler. A simpler semantics makes reasoning about programs more straightforward. This opens up more possibilities for correctness proofs, and for provably correct transformations into more efficient programs.

Another potential advantage of functional languages is the following. Since the evaluation of expressions has no side-effect on any state, separate subexpressions can be evaluated in any order without affecting each other. This means that functional programs may lend themselves well to parallel implementation, i.e. the computer can automatically farm out different subexpressions to different

[3]More: denotational semantics can be seen as an attempt to turn imperative languages into functional ones by making the state explicit.

processors. By contrast, imperative programs often impose a fairly rigid order of execution, and even the limited interleaving of instructions in modern pipelined processors turns out to be complicated and full of technical problems.

Actually, ML is not a purely functional programming language; it does have variables and assignments if required. Most of the time, we will work inside the purely functional subset. But even if we do use assignments, and lose some of the preceding benefits, there are advantages too in the more flexible use of functions that languages like ML allow. Programs can often be expressed in a very concise and elegant style using higher-order functions (functions that operate on other functions).[4] Code can be made more general, since it can be parametrized even over other functions. For example, a program to add up a list of numbers and a program to multiply a list of numbers can be seen as instances of the same program, parametrized by the pairwise arithmetic operation and the corresponding identity. In one case it is given + and 0 and in the other case, $*$ and 1.[5] Finally, functions can also be used to represent *infinite* data in a convenient way — for example we will show later how to use functions to perform exact calculation with real numbers, as distinct from floating point approximations.

At the same time, functional programs are not without their problems. Since they correspond less directly the the eventual execution in hardware, it can be difficult to reason about their exact usage of resources such as time and space. Input-output is also difficult to incorporate neatly into a functional model, though there are ingenious techniques based on infinite sequences.

It is up to readers to decide, after reading this book, on the merits of the functional style. We do not wish to enforce any ideologies, merely to point out that there *are* different ways of looking at programming, and that in the right situations, functional programming may have considerable merits. Most of our examples are chosen from areas that might loosely be described as 'symbolic computation', for we believe that functional programs work well in such applications. However, as always one should select the most appropriate tool for the job. It may be that imperative programming, object-oriented programming or logic programming are more suited to certain tasks. Horses for courses.

1.2 Outline

For those used to imperative programming, the transition to functional programming is inevitably difficult, whatever approach is taken. While some will be im-

[4]Elegance is subjective and conciseness is not an end in itself. Functional languages, and other languages like APL, often create a temptation to produce very short tricky code which is elegant to cognoscenti but obscure to outsiders.

[5]This parallels the notion of abstraction in pure mathematics, e.g. that the additive and multiplicative structures over numbers are instances of the abstract notion of a monoid. This similarly avoids duplication and increases elegance.

patient to get quickly to real programming, we have chosen to start with lambda calculus, and show how it can be seen as the theoretical underpinning for functional languages. This has the merit of corresponding quite well to the actual historical line of development.

So first we introduce lambda calculus, and show how what was originally intended as a formal logical system for mathematics turned out to be a completely general programming language. We then discuss why we might want to add types to lambda calculus, and show how it can be done. This leads us into ML, which is essentially an extended and optimized implementation of typed lambda calculus with a certain evaluation strategy. We cover the practicalities of basic functional programming in ML, and discuss polymorphism and most general types. We then move on to more advanced topics including exceptions and ML's imperative features. We conclude with some substantial examples, which we hope provide evidence for the power of ML.

Further reading

Numerous textbooks on 'functional programming' include a general introduction to the field and a contrast with imperative programming — browse through a few and find one that you like. For example, Henson (1987) contains a good introductory discussion, and features a similar mixture of theory and practice to this text. A detailed and polemical advocacy of the functional style is given by Backus (1978), the main inventor of FORTRAN. Gordon (1994) discusses the problems of incorporating input-output into functional languages, and some solutions. Readers interested in denotational semantics, for imperative and functional languages, may look at Winskel (1993).

Chapter 2

Lambda calculus

Lambda calculus is based on the so-called 'lambda notation' for denoting functions. In informal mathematics, when one wants to refer to a function, one usually first gives the function an arbitrary name, and thereafter uses that name, e.g.

> Suppose $f : \mathbb{R} \to \mathbb{R}$ is defined by:
>
> $$f(x) = \begin{cases} 0 & \text{if } x = 0 \\ x^2 sin(1/x^2) & \text{if } x \neq 0 \end{cases}$$
>
> Then $f'(x)$ is not Lebesgue integrable over the unit interval $[0, 1]$.

Most programming languages, C for example, are similar in this respect: we can define functions only by giving them names. For example, in order to use the successor function (which adds 1 to its argument) in nontrivial ways (e.g. consider a pointer to it), then even though it is very simple, we need to name it via some function definition such as:

```
int suc(int n)
 { return n + 1;
 }
```

In either mathematics or programming, this seems quite natural, and generally works well enough. However it can get clumsy when higher order functions (functions that manipulate other functions) are involved. In any case, if we want to treat functions on a par with other mathematical objects, the insistence on naming is rather inconsistent. When discussing an arithmetical expression built up from simpler ones, we just write the subexpressions down, without needing to give them names. Imagine if we always had to deal with arithmetic expressions in this way:

> Define x and y by $x = 2$ and $y = 4$ respectively. Then $xx = y$.

Lambda notation allows one to denote functions in much the same way as any other sort of mathematical object. There is a mainstream notation sometimes used in mathematics for this purpose, though it's normally still used as part of the definition of a temporary name. We can write

$$x \mapsto t[x]$$

to denote the function mapping any argument x to some arbitrary expression $t[x]$, which usually, but not necessarily, contains x (it is occasionally useful to "throw away" an argument). However, we shall use a different notation developed by Church (1941):

$$\lambda x.\, t[x]$$

which should be read in the same way. For example, $\lambda x. x$ is the identity function which simply returns its argument, while $\lambda x.\, x^2$ is the squaring function.

The symbol λ is completely arbitrary, and no significance should be read into it. (Indeed one often sees, particularly in French texts, the alternative notation $[x]\ t[x]$.) Apparently it arose by a complicated process of evolution. Originally, the famous *Principia Mathematica* (Whitehead and Russell 1910) used the 'hat' notation $t[\hat{x}]$ for the function of x yielding $t[x]$. Church modified this to $\hat{x}.\, t[x]$, but since the typesetter could not place the hat on top of the x, this appeared as $\wedge x.\, t[x]$, which then mutated into $\lambda x.\, t[x]$ in the hands of another typesetter.

2.1 The benefits of lambda notation

Using lambda notation we can clear up some of the confusion engendered by informal mathematical notation. For example, it's common to talk sloppily about '$f(x)$', leaving context to determine whether we mean f itself, or the result of applying it to particular x. A further benefit is that lambda notation gives an attractive analysis of practically the whole of mathematical notation. If we start with variables and constants, and build up expressions using just lambda-abstraction and application of functions to arguments, we can represent very complicated mathematical expressions.

We will use the conventional notation $f(x)$ for the application of a function f to an argument x, except that, as is traditional in lambda notation, the brackets may be omitted, allowing us to write just $f\ x$. For reasons that will become clear in the next paragraph, we assume that function application associates to the left, i.e. $f\ x\ y$ means $(f(x))(y)$. As a shorthand for $\lambda x.\, \lambda y.\, t[x, y]$ we will use $\lambda x\ y.\, t[x, y]$, and so on. We also assume that the scope of a lambda abstraction extends as far to the right as possible. For example $\lambda x. x\ y$ means $\lambda x. (x\ y)$ rather than $(\lambda x.\, x)\ y$.

At first sight, we need some special notation for functions of several arguments. However there is a way of breaking down such applications into ordinary lambda notation, called *currying*, after the logician Curry (1930). (Actually the device had previously been used by both Frege (1893) and Schönfinkel (1924), but it's easy to understand why the corresponding appellations haven't caught the public imagination.) The idea is to use expressions like $\lambda x\ y.\ x + y$. This may be regarded as a function $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$, so it is said to be a 'higher order function' or 'functional' since when applied to one argument, it yields another function, which then accepts the second argument. In a sense, it takes its arguments one at a time rather than both together. So we have for example:

$$(\lambda x\ y.\ x + y)\ 1\ 2 = (\lambda y.\ 1 + y)\ 2 = 1 + 2$$

Observe that function application is assumed to associate to the left in lambda notation precisely because currying is used so much.

Lambda notation is particularly helpful in providing a unified treatment of bound variables. Variables in mathematics normally express the dependency of some expression on the value of that variable; for example, the value of $x^2 + 2$ depends on the value of x. In such contexts, we will say that a variable is *free*. However there are other situations where a variable is merely used as a place-marker, and does not indicate such a dependency. Two common examples are the variable m in

$$\sum_{m=1}^{n} m = \frac{n(n+1)}{2}$$

and the variable y in

$$\int_0^x 2y + a\ dy = x^2 + ax$$

In logic, the quantifiers $\forall x.\ P[x]$ ('for all x, $P[x]$') and $\exists x.\ P[x]$ ('there exists an x such that $P[x]$') provide further examples, and in set theory we have set abstractions like $\{x \mid P[x]\}$ as well as indexed unions and intersections. In such cases, a variable is said to be *bound*. In a certain subexpression it is free, but in the whole expression, it is bound by a *variable-binding operation* like summation. The part 'inside' this variable-binding operation is called the *scope* of the bound variable.

A similar situation occurs in most programming languages, at least from Algol 60 onwards. Variables have a definite scope, and the formal arguments of procedures and functions are effectively bound variables, e.g. n in the C definition of the successor function given above. One can actually regard variable declarations as binding operations for the enclosed instances of the corresponding variable(s). Note, by the way, that the *scope* of a variable should be distinguished sharply from its *lifetime*. In the C function `rand` that we gave in the introduction, n had

a textually limited scope but it retained its value even outside the execution of that part of the code.

We can freely change the name of a bound variable without changing the meaning of the expression, e.g.

$$\int_0^x 2z + a \; dz = x^2 + ax$$

Similarly, in lambda notation, $\lambda x.\ E[x]$ and $\lambda y.\ E[y]$ are equivalent; this is called *alpha*-equivalence and the process of transforming between such pairs is called alpha-conversion. We should add the proviso that y is not a free variable in $E[x]$, or the meaning clearly may change, just as

$$\int_0^x 2a + a \; da \neq x^2 + ax$$

It is possible to have identically-named free and bound variables in the same expression; though this can be confusing, it is technically unambiguous, e.g.

$$\int_0^x 2x + a \; dx = x^2 + ax$$

In fact the usual Leibniz notation for derivatives has just this property, e.g. in:

$$\frac{d}{dx}x^2 = 2x$$

x is used both as a bound variable to indicate that differentiation is to take place with respect to x, and as a free variable to show where to evaluate the resulting derivative. This can be confusing; e.g. $f'(g(x))$ is usually taken to mean something different from $\frac{d}{dx}f(g(x))$. Careful writers, especially in multivariate work, often make the separation explicit by writing:

$$|\frac{d}{dx}x^2|_x = 2x$$

or

$$|\frac{d}{dz}z^2|_x = 2x$$

Part of the appeal of lambda notation is that all variable-binding operations like summation, differentiation and integration can be regarded as functions applied to lambda-expressions. Subsuming all variable-binding operations by lambda abstraction allows us to concentrate on the technical problems of bound variables in one particular situation. For example, we can view $\frac{d}{dx}x^2$ as a syntactic sugaring of $D\ (\lambda x.\ x^2)\ x$ where $D : (\mathbb{R} \to \mathbb{R}) \to \mathbb{R} \to \mathbb{R}$ is a differentiation operator, yielding the derivative of its first (function) argument at the point indicated by its second argument. Breaking down the everyday syntax completely into lambda notation, we have $D\ (\lambda x.\ \mathrm{EXP}\ x\ 2)\ x$ for some constant EXP representing the exponential function.

In this way, lambda notation is an attractively general 'abstract syntax' for mathematics; all we need is the appropriate stock of constants to start with. Lambda abstraction seems, in retrospect, to be the appropriate primitive in terms of which to analyze variable binding. This idea goes back to Church's encoding of higher order logic in lambda notation, and as we shall see in the next chapter, Landin has pointed out how many constructs from programming languages have a similar interpretation. In recent times, the idea of using lambda notation as a universal abstract syntax has been put especially clearly by Martin-Löf, and is often referred to in some circles as 'Martin-Löf's theory of expressions and arities'.[1]

2.2 Russell's paradox

As we have said, one of the appeals of lambda notation is that it permits an analysis of more or less all of mathematical syntax. Originally, Church hoped to go further and include set theory, which, as is well known, is powerful enough to form a foundation for much of modern mathematics. Given any set S, we can form its so-called *characteristic predicate* χ_S, such that:

$$\chi_S(x) = \begin{cases} true & \text{if } x \in S \\ false & \text{if } x \notin S \end{cases}$$

Conversely, given any unary predicate (i.e. function of one argument) P, we can consider the set of all x satisfying $P(x)$ — we will just write $P(x)$ for $P(x) = true$. Thus, we see that sets and predicates are just different ways of talking about the same thing. Instead of regarding S as a set, and writing $x \in S$, we can regard it as a predicate and write $S(x)$.

This permits a natural analysis into lambda notation: we can allow arbitrary lambda expressions as functions, and hence indirectly as sets. Unfortunately, this turns out to be inconsistent. The simplest way to see this is to consider the Russell paradox of the set of all sets that do not contain themselves:

$$R = \{x \mid x \notin x\}$$

We have $R \in R \Leftrightarrow R \notin R$, a stark contradiction. In terms of lambda defined functions, we set $R = \lambda x.\neg(x\ x)$, and find that $R\ R = \neg(R\ R)$, obviously counter to the intuitive meaning of the negation operator $\neg$.

To avoid such paradoxes, Church (1940) followed Russell in augmenting lambda notation with a notion of *type*; we shall consider this in a later chapter. However the paradox itself is suggestive of some interesting possibilities in the standard, untyped, system, as we shall see later.

[1] This was presented at the Brouwer Symposium in 1981, but was not described in the printed proceedings.

2.3 Lambda calculus as a formal system

We have taken for granted certain obvious facts, e.g. that $(\lambda y.\ 1 + y)\ 2 = 1 + 2$, since these reflect the intended meaning of abstraction and application, which are in a sense converse operations. Lambda *calculus* arises if we enshrine certain such principles, and *only* those, as a set of formal rules. The appeal of this is that the rules can then be used mechanically, just as one might transform $x - 3 = 5 - x$ into $2x = 5 + 3$ without pausing each time to think about *why* these rules about moving things from one side of the equation to the other are valid. As Whitehead (1919) says, symbolism and formal rules of manipulation:

> [...] have invariably been introduced to make things easy. [...] by the aid of symbolism, we can make transitions in reasoning almost mechanically by the eye, which otherwise would call into play the higher faculties of the brain. [...] Civilisation advances by extending the number of important operations which can be performed without thinking about them.

2.3.1 Lambda terms

Lambda calculus is based on a formal notion of lambda term, and these terms are built up from variables and some fixed set of constants using the operations of function application and lambda abstraction. This means that every lambda term falls into one of the following four categories:

1. **Variables:** these are indexed by arbitrary alphanumeric strings; typically we will use single letters from towards the end of the alphabet, e.g. x, y and z.

2. **Constants:** how many constants there are in a given syntax of lambda terms depends on context. Sometimes there are none at all. We will also denote them by alphanumeric strings, leaving context to determine when they are meant to be constants.

3. **Combinations**, i.e. the application of a function s to an argument t; both these components s and t may themselves be arbitrary λ-terms. We will write combinations simply as $s\ t$. We often refer to s as the 'rator' and t as the 'rand' (short for 'operator' and 'operand' respectively).

4. **Abstractions** of an arbitrary lambda-term s over a variable x (which may or may not occur free in s), denoted by $\lambda x.\ s$.

Formally, this defines the set of lambda terms inductively, i.e. lambda terms arise *only* in these four ways. This justifies our: