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:

- Defining functions over lambda terms by primitive recursion.
- Proving properties of lambda terms by structural induction.

A formal discussion of inductive generation, and the notions of primitive recursion and structural induction, may be found elsewhere. We hope most readers who are unfamiliar with these terms will find that the examples below give them a sufficient grasp of the basic ideas.

We can describe the syntax of lambda terms by a BNF (Backus-Naur form) grammar, just as we do for programming languages.

$$Exp = Var \mid Const \mid Exp\ Exp \mid \lambda\, Var.\ Exp$$

and, following the usual computer science view, we will identify lambda terms with abstract syntax trees, rather than with sequences of characters. This means that conventions such as the left-association of function application, the reading of $\lambda x\ y.\ s$ as $\lambda x.\ \lambda y.\ s$, and the ambiguity over constant and variable names are purely a matter of parsing and printing for human convenience, and not part of the formal system.

One feature worth mentioning is that we use single characters to stand for variables and constants in the formal system of lambda terms and as so-called 'metavariables' standing for arbitrary terms. For example, $\lambda x.\ s$ might represent the constant function with value s, or an arbitrary lambda abstraction using the variable x. To make this less confusing, we will normally use letters such as s, t and u for metavariables over terms. It would be more precise if we denoted the variable x by V_x (the x'th variable) and likewise the constant k by C_k — then all the variables in terms would have the same status. However this makes the resulting terms a bit cluttered.

2.3.2 Free and bound variables

We now formalize the intuitive idea of free and bound variables in a term, which, incidentally, gives a good illustration of defining a function by primitive recursion. Intuitively, a variable in a term is free if it does not occur inside the scope of a corresponding abstraction. We will denote the set of free variables in a term s by $FV(s)$, and define it by recursion as follows:

$$\begin{aligned}
FV(x) &= \{x\} \\
FV(c) &= \emptyset \\
FV(s\ t) &= FV(s) \cup FV(t) \\
FV(\lambda x.\ s) &= FV(s) - \{x\}
\end{aligned}$$

Similarly we can define the set of bound variables in a term $BV(s)$:

$$\begin{aligned} BV(x) &= \emptyset \\ BV(c) &= \emptyset \\ BV(s\ t) &= BV(s) \cup BV(t) \\ BV(\lambda x.\ s) &= BV(s) \cup \{x\} \end{aligned}$$

For example, if $s = (\lambda x\ y.\ x)\ (\lambda x.\ z\ x)$ we have $FV(s) = \{z\}$ and $BV(s) = \{x, y\}$. Note that in general a variable can be both free and bound in the same term, as illustrated by some of the mathematical examples earlier. As an example of using structural induction to establish properties of lambda terms, we will prove the following theorem (a similar proof works for BV too):

Theorem 2.1 *For any lambda term s, the set $FV(s)$ is finite.*
Proof: *By structural induction. Certainly if s is a variable or a constant, then by definition $FV(s)$ is finite, since it is either a singleton or empty. If s is a combination $t\ u$ then by the inductive hypothesis, $FV(t)$ and $FV(u)$ are both finite, and then $FV(s) = FV(t) \cup FV(u)$, which is therefore also finite (the union of two finite sets is finite). Finally, if s is of the form $\lambda x.\ t$ then $FV(t)$ is finite, by the inductive hypothesis, and by definition $FV(s) = FV(t) - \{x\}$ which must be finite too, since it is no larger. Q.E.D.*

2.3.3 Substitution

The rules we want to formalize include the stipulation that lambda abstraction and function application are inverse operations. That is, if we take a term $\lambda x.\ s$ and apply it as a function to an argument term t, the answer is the term s with all free instances of x replaced by t. We often make this more transparent in discussions by using the notation $\lambda x.\ s[x]$ and $s[t]$ for the respective terms.

However this simple-looking notion of substituting one term for a variable in another term is surprisingly difficult. Some notable logicians have made faulty statements regarding substitution. In fact, this difficulty is rather unfortunate since as we have said, the appeal of formal rules is that they can be applied mechanically.

We will denote the operation of substituting a term s for a variable x in another term t by $t[s/x]$. One sometimes sees various other notations, e.g. $t[x{:=}s]$, $[s/x]t$, or even $t[x/s]$. The notation we use is perhaps most easily remembered by noting the vague analogy with multiplication of fractions: $x[t/x] = t$. At first sight, one can define substitution formally by recursion as follows:

$$\begin{aligned} x[t/x] &= t \\ y[t/x] &= y \text{ if } x \neq y \end{aligned}$$

$$\begin{array}{rcl} c[t/x] & = & c \\ (s_1\ s_2)[t/x] & = & s_1[t/x]\ s_2[t/x] \\ (\lambda x.\ s)[t/x] & = & \lambda x.\ s \\ (\lambda y.\ s)[t/x] & = & \lambda y.\ (s[t/x]) \text{ if } x \neq y \end{array}$$

However this isn't quite right. For example $(\lambda y.\, x+y)[y/x] = \lambda y.\, y+y$, which doesn't correspond to the intuitive answer.[2] The original lambda term was 'the function that adds x to its argument', so after substitution, we might expect to get 'the function that adds y to its argument'. What we actually get is 'the function that doubles its argument'. The problem is that the variable y that we have substituted is *captured* by the variable-binding operation $\lambda y. \ldots$. We should first rename the bound variable:

$$(\lambda y.\ x+y) = (\lambda w.\ x+w)$$

and only now perform a naive substitution operation:

$$(\lambda w.\ x+w)[y/x] = \lambda w.\ y+w$$

We can take two approaches to this problem. Either we can add a condition on all instances of substitution to disallow it wherever variable capture would occur, or we can modify the formal definition of substitution so that it performs the appropriate renamings automatically. We will opt for this latter approach. Here is the definition of substitution that we use:

$$\begin{array}{rcl} x[t/x] & = & t \\ y[t/x] & = & y \text{ if } x \neq y \\ c[t/x] & = & c \\ (s_1\ s_2)[t/x] & = & s_1[t/x]\ s_2[t/x] \\ (\lambda x.\ s)[t/x] & = & \lambda x.\ s \\ (\lambda y.\ s)[t/x] & = & \lambda y.\ (s[t/x]) \text{ if } x \neq y \text{ and either } x \notin FV(s) \text{ or } y \notin FV(t) \\ (\lambda y.\ s)[t/x] & = & \lambda z.\ (s[z/y][t/x]) \text{ otherwise, where } z \notin FV(s) \cup FV(t) \end{array}$$

The only difference is in the last two lines. We substitute as before in the two safe situations where either x isn't free in s, so the substitution is trivial, or where y isn't free in t, so variable capture won't occur (at this level). However where these conditions fail, we first rename y to a new variable z, chosen not to be free in either s or t, then proceed as before. For definiteness, the variable z can be chosen in some canonical way, e.g. the lexicographically first name not occurring as a free variable in either s or t.[3]

[2] We will continue to use infix syntax for standard operators; strictly we should write $+\ x\ y$ rather than $x+y$.

[3] Cognoscenti may also be worried that this definition is not in fact, *primitive* recursive,

2.3.4 Conversions

Lambda calculus is based on three 'conversions', which transform one term into another one intuitively equivalent to it. These are traditionally denoted by the Greek letters α (alpha), β (beta) and η (eta).[4] Here are the formal definitions of the operations, using annotated arrows for the conversion relations.

- Alpha conversion: $\lambda x.\ s \underset{\alpha}{\longrightarrow} \lambda y.\ s[y/x]$ provided $y \notin FV(s)$. For example, $\lambda u.\ u\ v \underset{\alpha}{\longrightarrow} \lambda w.\ w\ v$, but $\lambda u.\ u\ v \underset{\alpha}{\not\longrightarrow} \lambda v.\ v\ v$. The restriction avoids another instance of variable capture.
- Beta conversion: $(\lambda x.\ s)\ t \underset{\beta}{\longrightarrow} s[t/x]$.
- Eta conversion: $\lambda x.t\ x \underset{\eta}{\longrightarrow} t$, provided $x \notin FV(t)$. For example $\lambda u.v\ u \underset{\eta}{\longrightarrow} v$ but $\lambda u.\ u\ u \underset{\eta}{\not\longrightarrow} u$.

Of the three, β-conversion is the most important one to us, since it represents the evaluation of a function on an argument. α-conversion is a technical device to change the names of bound variables, while η-conversion is a form of *extensionality* and is therefore mainly of interest to those taking a logical, not a programming, view of lambda calculus.

2.3.5 Lambda equality

Using these conversion rules, we can define formally when two lambda terms are to be considered equal. Roughly, two terms are equal if it is possible to get from one to the other by a finite sequence of conversions (α, β or η), either forward or backward, at any depth inside the term. We can say that lambda equality is the *congruence closure* of the three reduction operations together, i.e. the smallest relation containing the three conversion operations and closed under reflexivity, symmetry, transitivity and substitutivity. Formally we can define it inductively as follows, where the horizontal lines should be read as 'if what is above the line holds, then so does what is below'.

$$\frac{s \underset{\alpha}{\longrightarrow} t \text{ or } s \underset{\beta}{\longrightarrow} t \text{ or } s \underset{\eta}{\longrightarrow} t}{s = t}$$

$$\frac{}{t = t}$$

because of the last clause. However it can easily be modified into a primitive recursive definition of multiple, parallel, substitution. This procedure is analogous to strengthening an induction hypothesis during a proof by induction. Note that by construction the pair of substitutions in the last line can be done in parallel rather than sequentially without affecting the result.

[4] These names are due to Curry. Church originally referred to α-conversion and β-conversion as 'rule of procedure I' and 'rule of procedure II' respectively.

$$\frac{s = t}{t = s}$$

$$\frac{s = t \text{ and } t = u}{s = u}$$

$$\frac{s = t}{s\ u = t\ u}$$

$$\frac{s = t}{u\ s = u\ t}$$

$$\frac{s = t}{\lambda x.\ s = \lambda x.\ t}$$

Note that the use of the ordinary equality symbol (=) here is misleading. We are actually *defining* the relation of lambda equality, and it isn't clear that it corresponds to equality of the corresponding mathematical objects in the usual sense.[5] Certainly it must be distinguished sharply from equality at the *syntactic* level. We will refer to this latter kind of equality as 'identity' and use the special symbol $\equiv$. For example $\lambda x.\ x \not\equiv \lambda y.\ y$ but $\lambda x.\ x = \lambda y.\ y$.

For many purposes, α-conversions are immaterial, and often $\equiv_\alpha$ is used instead of strict identity. This is defined like lambda equality, except that only α-conversions are allowed. For example, $(\lambda x.\ x)y \equiv_\alpha (\lambda y.\ y)y$. Many writers use this as identity on lambda terms, i.e. consider equivalence classes of terms under $\equiv_\alpha$. There are alternative formalizations of syntax where bound variables are unnamed (de Bruijn 1972), and here syntactic identity corresponds to our $\equiv_\alpha$.

2.3.6 Extensionality

We have said that η-conversion embodies a principle of *extensionality*. In general philosophical terms, two properties are said to be *extensionally* equivalent (or *coextensive*) when they are satisfied by exactly the same objects. In mathematics, we usually take an extensional view of sets, i.e. say that two sets are equal precisely if they have the same elements. Similarly, we normally say that two functions are equal precisely if they have the same domain and give the same result on all arguments in that domain.

As a consequence of η-conversion, our notion of lambda equality is extensional. Indeed, if $f\ x$ and $g\ x$ are equal for any x, then in particular $f\ y = g\ y$ where y is chosen not to be free in either f or g. Therefore by the last rule above, $\lambda y.\ f\ y = \lambda y.\ g\ y$. Now by η-converting a couple of times at depth, we see that

[5] Indeed, we haven't been very precise about what the corresponding mathematical objects *are*. But there are models of the lambda calculus where our lambda equality is interpreted as actual equality.

$f = g$. Conversely, extensionality implies that all instances of η-conversion do indeed give a valid equation, since by β-reduction, $(\lambda x.\ t\ x)\ y = t\ y$ for any y when x is not free in t. This is the import of η-conversion, and having discussed that, we will largely ignore it in favour of the more computationally significant β-conversion.

2.3.7 Lambda reduction

Lambda equality, unsurprisingly, is a symmetric relation. Though it captures the notion of equivalence of lambda terms well, it is more interesting from a computational point of view to consider an asymmetric version. We will define a 'reduction' relation $\longrightarrow$ as follows:

$$\frac{s \underset{\alpha}{\longrightarrow} t \text{ or } s \underset{\beta}{\longrightarrow} t \text{ or } s \underset{\eta}{\longrightarrow} t}{s \longrightarrow t}$$

$$\frac{}{t \longrightarrow t}$$

$$\frac{s \longrightarrow t \text{ and } t \longrightarrow u}{s \longrightarrow u}$$

$$\frac{s \longrightarrow t}{s\ u \longrightarrow t\ u}$$

$$\frac{s \longrightarrow t}{u\ s \longrightarrow u\ t}$$

$$\frac{s \longrightarrow t}{\lambda x.\ s \longrightarrow \lambda x.\ t}$$

Actually the name 'reduction' (and one also hears β-conversion called β-reduction) is a slight misnomer, since it can cause the size of a term to grow, e.g.

$$\begin{aligned}
(\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x) &\longrightarrow (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x) \\
&\longrightarrow (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x) \\
&\longrightarrow \ldots
\end{aligned}$$

However reduction does correspond to a systematic attempt to evaluate a term by repeatedly evaluating combinations $f(x)$ where f is a lambda abstraction. When no more reductions except for α conversions are possible we say that the term is in *normal form*.

2.3.8 Reduction strategies

Let us recall, in the middle of these theoretical considerations, the relevance of all this to functional programming. A functional program is an *expression* and executing it means evaluating the expression. In terms of the concepts discussed here, we are proposing to start with the relevant term and keep on applying reductions until there is nothing more to be evaluated. But how are we to choose which reduction to apply at each stage? The reduction relation is not deterministic, i.e. for some terms t there are several t_i such that $t \longrightarrow t_i$. Sometimes this can make the difference between a finite and infinite reduction sequence, i.e. between a program terminating and failing to terminate. For example, by reducing the innermost *redex* (reducible expression) in the following, we have an infinite reduction sequence:

$$\begin{array}{ll} & (\lambda x.\ y)\ ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & (\lambda x.\ y)\ ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & (\lambda x.\ y)\ ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & \cdots \end{array}$$

and so ad infinitum. However the alternative of reducing the outermost redex first gives:

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

immediately, and there are no more reductions to apply.

The situation is clarified by the following theorems, whose proofs are too long to be given here. The first one says that the situation we have noted above is true in a more general sense, i.e. that reducing the leftmost outermost redex is the best strategy for ensuring termination.

Theorem 2.2 *If $s \longrightarrow t$ with t in normal form, then the reduction sequence that arises from s by always reducing the leftmost outermost redex is guaranteed to terminate in normal form.*

Formally, we define the 'leftmost outermost' redex recursively: for a term $(\lambda x.\ s)\ t$ it is the term itself; for any other term $s\ t$ it is the leftmost outermost redex of s, and for an abstraction $\lambda x.\ s$ it is the leftmost outermost redex of s. In terms of concrete syntax, we always reduce the redex whose λ is the furthest to the left.

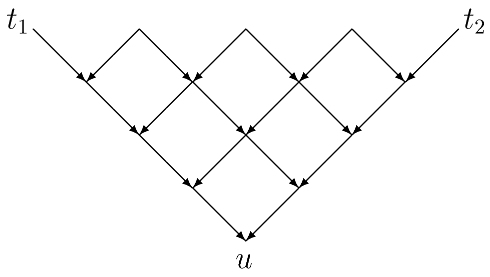
2.3.9 The Church-Rosser theorem

The next assertion, the famous Church-Rosser theorem, states that if we start from a term t and perform any two finite reduction sequences, there are always two more reduction sequences that bring the two back to the same term (though of course this might not be in normal form).

Theorem 2.3 *If $t \longrightarrow s_1$ and $t \longrightarrow s_2$, then there is a term u such that $s_1 \longrightarrow u$ and $s_2 \longrightarrow u$.*

This has at least the following important consequences:

Corollary 2.4 *If $t_1 = t_2$ then there is a term u with $t_1 \longrightarrow u$ and $t_2 \longrightarrow u$.*
Proof: *It is easy to see (by structural induction) that the equality relation $=$ is in fact the symmetric transitive closure of the reduction relation. Now we can proceed by induction over the construction of the symmetric transitive closure. However less formally minded readers will probably find the following diagram more convincing:*



We assume $t_1 = t_2$, so there is some sequence of reductions in both directions (i.e. the zigzag at the top) that connects them. Now the Church-Rosser theorem allows us to fill in the remainder of the sides in the above diagram, and hence reach the result by composing these reductions. Q.E.D.

Corollary 2.5 *If $t = t_1$ and $t = t_2$ with t_1 and t_2 in normal form, then $t_1 \equiv_\alpha t_2$, i.e. t_1 and t_2 are equal apart from α conversions.*
Proof: *By the first corollary, we have some u with $t_1 \longrightarrow u$ and $t_2 \longrightarrow u$. But since t_1 and t_2 are already in normal form, these reduction sequences to u can only consist of alpha conversions. Q.E.D.*

Hence normal forms, when they do exist, are unique up to alpha conversion. This gives us the first proof we have available that the relation of lambda equality isn't completely trivial, i.e. that there are any unequal terms. For example, since $\lambda x\ y.\ x$ and $\lambda x\ y.\ y$ are not interconvertible by alpha conversions alone, they cannot be equal.

Let us sum up the computational significance of all these assertions. In some sense, reducing the leftmost outermost redex is the best strategy, since it will work if any strategy will. This is known as *normal order reduction*. On the other hand *any* terminating reduction sequence will always give the same result, and moreover it is never too late to abandon a given strategy and start using normal order reduction. We will see later how this translates into practical terms.

2.4 Combinators

Combinators were actually developed as an independent theory by Schönfinkel (1924) before lambda notation came along. Moreover Curry rediscovered the theory soon afterwards, independently of Schönfinkel and of Church. (When he found out about Schönfinkel's work, Curry attempted to contact him, but by that time, Schönfinkel had been committed to a lunatic asylum.) We will distort the historical development by presenting the theory of combinators as an aspect of lambda calculus.

We will define a *combinator* simply to be a lambda term with no free variables. Such a term is also said to be *closed*; it has a fixed meaning independent of the values of any variables. Now we will later, in the course of functional programming, come across many useful combinators. But the cornerstone of the theory of combinators is that one can get away with just a few combinators, and express in terms of those and variables *any* term at all: the operation of lambda abstraction is unnecessary. In particular, a closed term can be expressed purely in terms of these few combinators. We start by defining:

$$\begin{aligned} I &= \lambda x.\ x \\ K &= \lambda x\ y.\ x \\ S &= \lambda f\ g\ x.\ (f\ x)(g\ x) \end{aligned}$$

We can motivate the names as follows.[6] I is the identity function. K produces constant functions:[7] when applied to an argument a it gives the function $\lambda y.\ a$. Finally S is a 'sharing' combinator, which takes two functions and an argument and shares out the argument among the functions. Now we prove the following:

Lemma 2.6 *For any lambda term t not involving lambda abstraction, there is a term u also not containing lambda abstractions, built up from S, K, I and variables, with $FV(u) = FV(t) - \{x\}$ and $u = \lambda x.\ t$, i.e. u is lambda-equal to $\lambda x.\ t$.*

Proof: *By structural induction on the term t. By hypothesis, it cannot be an abstraction, so there are just three cases to consider.*

- *If t is a variable, then there are two possibilities. If it is equal to x, then $\lambda x.\ x = I$, so we are finished. If not, say $t = y$, then $\lambda x.\ y = K\ y$.*
- *If t is a constant c, then $\lambda x.\ c = K\ c$.*
- *If t is a combination, say $s\ u$, then by the inductive hypothesis, there are lambda-free terms s' and u' with $s' = \lambda x.\ s$ and $u' = \lambda x.\ u$. Now we claim $S\ s'\ u'$ suffices. Indeed:*

[6] We are not claiming these are the historical reasons for them.

[7] Konstant — Schönfinkel was German, though in fact he originally used C.

$$\begin{aligned} S\ s'\ u'\ x &= S\ (\lambda x.\ s)\ (\lambda x.\ u)\ x \\ &= ((\lambda x.\ s)\ x)((\lambda x.\ u)\ x) \\ &= s\ u \\ &= t \end{aligned}$$

Therefore, by η-conversion, we have $S\ s'\ u' = \lambda x.\ S\ s'\ u'\ x = \lambda x.\ t$, since by the inductive hypothesis x is not free in s' or u'.

Q.E.D.

Theorem 2.7 *For any lambda term t, there is a lambda-free term t' built up from S, K, I and variables, with $FV(t') = FV(t)$ and $t' = t$.*
Proof: *By structural induction on t, using the lemma. For example, if t is $\lambda x.\ s$, we first find, by the inductive hypothesis, a lambda-free equivalent s' of s. Now the lemma can be applied to $\lambda x.\ s'$. The other cases are straightforward. Q.E.D.*

This remarkable fact can be strengthened, since I is definable in terms of S and K. Note that for any A:

$$\begin{aligned} S\ K\ A\ x &= (K\ x)(A\ x) \\ &= (\lambda y.\ x)(A\ x) \\ &= x \end{aligned}$$

So again by η-converting, we see that $I = S\ K\ A$ for any A. It is customary, for reasons that will become clear when we look at types, to use $A = K$. So $I = S\ K\ K$, and we can avoid the use of I in our combinatory expression.

Note that the proofs above are constructive, in the sense that they guide one in a definite procedure that, given a lambda term, produces a combinator equivalent. One proceeds bottom-up, and at each lambda abstraction, which by construction has a lambda-free body, applies the top-down transformations given in the lemma.

Although we have presented combinators as certain lambda terms, they can also be developed as a theory in their own right. That is, one starts with a formal syntax excluding lambda abstractions but including combinators. Instead of α, β and η conversions, one posits *conversion rules* for expressions involving combinators, e.g. $K\ x\ y \longrightarrow x$. As an independent theory, this has many analogies with lambda calculus, e.g. the Church-Rosser theorem holds for this notion of reduction too. Moreover, the ugly difficulties connected with bound variables are avoided completely. However the resulting system is, we feel, less intuitive, since combinatory expressions can get rather obscure.

Apart from their purely logical interest, combinators have a certain practical potential. As we have already hinted, and as will become much clearer in the later chapters, lambda calculus can be seen as a simple functional language, forming the core of real practical languages like ML. We might say that the theorem of combinatory completeness shows that lambda calculus can be 'compiled down' to a 'machine code' of combinators. This computing terminology is not as fanciful as it appears. Combinators have been used as an implementation technique for functional languages, and real hardware has been built to evaluate combinatory expressions.

Further reading

An encyclopedic but clear book on lambda calculus is Barendregt (1984). Another popular textbook is Hindley and Seldin (1986). Both these contain proofs of the results that we have merely asserted. A more elementary treatment focused towards computer science is Part II of Gordon (1988). Much of our presentation here and later is based on this last book.

Exercises

1. Find a normal form for $(\lambda x\ x\ x.\ x)\ a\ b\ c$.

2. Define $twice = \lambda f\ x.\ f(fx)$. What is the intuitive meaning of $twice$? Find a normal form for $twice\ twice\ twice\ f\ x$. (Remember that function application associates to the left.)

3. Find a term t such that $t \underset{\beta}{\longrightarrow} t$. Is it true to say that a term is in normal form if and only if whenever $t \longrightarrow t'$ then $t \equiv_\alpha t'$?

4. What are the circumstances under which $s[t/x][u/y] \equiv_\alpha s[u/y][t/x]$?

5. Find an equivalent in terms of the S, K and I combinators alone for $\lambda f\ x.\ f(x\ x)$.

6. Find a *single* combinator X such that all λ-terms are equal to a term built from X and variables. You may find it helpful to consider $A = \lambda p.p\ K\ S\ K$ and then think about $A\ A\ A$ and $A\ (A\ A)$.

7. Prove that any X is a fixed point combinator if and only if it is itself a fixed point of G, where $G = \lambda y\ m.\ m(y\ m)$.

Chapter 3

Lambda calculus as a programming language

One of the central questions studied by logicians in the 1930s was the *Entscheidungsproblem* or 'decision problem'. This asked whether there was some systematic, mechanical procedure for deciding validity in first order logic. If it turned out that there was, it would have fundamental philosophical, and perhaps practical, significance. It would mean that a huge variety of complicated mathematical problems could in principle be solved purely by following some single fixed method — we would now say *algorithm* — no additional creativity would be required.

As it stands the question is hard to answer, because we need to define in mathematical terms what it means for there to be a 'systematic, mechanical' procedure for something. Perhaps Turing (1936) gave the best analysis, by arguing that the mechanical procedures are those that could in principle be performed by a reasonably intelligent clerk with no knowledge of the underlying subject matter. He abstracted from the behaviour of such a clerk and arrived at the famous notion of a 'Turing machine'. Despite the fact that it arose by considering a *human* 'computer', and was purely a mathematical abstraction, we now see a Turing machine as a very simple computer. Despite being simple, a Turing machine is capable of performing any calculation that can be done by a physical computer.[1] A model of computation of equal power to a Turing machine is said to be *Turing complete*.

At about the same time, there were several other proposed definitions of 'mechanical procedure'. Most of these turned out to be equivalent in power to Turing machines. In particular, although it originally arose as a foundation for mathematics, lambda calculus can also be seen as a programming language, to be executed by performing β-conversions. In fact, Church proposed before Turing that the set of operations that can be expressed in lambda calculus formalizes the

[1] Actually more, since it neglects the necessary finite limit to the computer's store. Arguably any existing computer is actually a finite state machine, but the assumption of unbounded memory seems a more fruitful abstraction.

intuitive idea of a mechanical procedure, a postulate known as *Church's thesis*. Church (1936) showed that if this was accepted, the *Entscheidungsproblem* is unsolvable. Turing later showed that the functions definable in lambda calculus are precisely those computable by a Turing machine, which lent more credibility to Church's claims.

To a modern programmer, Turing machine programs are just about recognizable as a very primitive kind of machine code. Indeed, it seems that Turing machines, and in particular Turing's construction of a 'universal machine',[2] were a key influence in the development of modern stored program computers, though the extent and nature of this influence is still a matter for controversy (Robinson 1994). It is remarkable that several of the other proposals for a definition of 'mechanical procedure', often made before the advent of electronic computers, correspond closely to real programming methods. For example, Markov algorithms, the standard approach to computability in the (former) Soviet Union, can be seen as underlying the programming language SNOBOL. In what follows, we are concerned with the influence of lambda calculus on functional programming languages.

LISP, the second oldest high level language (FORTRAN being the oldest) took a few ideas from lambda calculus, in particular the notation '(LAMBDA $\cdots$)' for anonymous functions. However it was not really based on lambda calculus at all. In fact the early versions, and some current dialects, use a system of *dynamic binding* that does not correspond to lambda calculus. (We will discuss this later.) Moreover there was no real support for higher order functions, while there was considerable use of imperative features. Nevertheless LISP deserves to be considered as the first functional programming language, and pioneered many of the associated techniques such as automatic storage allocation and garbage collection.

The influence of lambda calculus really took off with the work of Landin and Strachey in the 60s. In particular, Landin showed how many features of current (imperative) languages could usefully be analyzed in terms of lambda calculus, e.g. the notion of variable scoping in Algol 60. Landin (1966) went on to propose the use of lambda calculus as a core for programming languages, and invented a functional language ISWIM ('If you See What I Mean'). This was widely influential and spawned many real languages.

ML started life as the metalanguage (hence the name ML) of a theorem proving system called Edinburgh LCF (Gordon, Milner, and Wadsworth 1979). That is, it was intended as a language in which to write algorithms for proving theorems in a formal deductive calculus. It was strongly influenced by ISWIM but added an innovative polymorphic type system including abstract types, as well as a system of *exceptions* to deal with error conditions. These facilities were dictated

[2] A single Turing machine program capable of simulating any other — we would now regard it as an *interpreter*. You will learn about this in the course on Computation Theory.

by the application at hand, and this policy resulted in a coherent and focused design. This narrowness of focus is typical of successful languages (C is another good example) and contrasts sharply with the failure of committee-designed languages like Algol 60 which have served more as a source of important ideas than as popular practical tools. We will have more to say about ML later. Let us now see how *pure lambda calculus* can be used as a programming language.

3.1 Representing data in lambda calculus

Programs need data to work on, so we start by fixing lambda expressions to encode data. Furthermore, we define a few basic operations on this data. In many cases, we show how a string of human-readable syntax s can be translated directly into a lambda expression s'. This procedure is known as 'syntactic sugaring' — it makes the bitter pill of pure lambda notation easier to digest. We write such definitions as:

$$s \triangleq s'$$

This notation means '$s = s'$ by definition'; one also commonly sees this written as '$s =_{def} s'$'. If preferred, we could always regard these as defining a single constant denoting that operation, which is then applied to arguments in the usual lambda calculus style, the surface notation being irrelevant. For example we can imagine 'if E then E_1 else E_2' as 'COND E E_1 E_2' for some constant COND. In such cases, any variables on the left of the definition need to be abstracted over, e.g. instead of

$$\text{fst } p \triangleq p \text{ true}$$

(see below) we could write

$$\text{fst} \triangleq \lambda p.\, p \text{ true}$$

3.1.1 Truth values and the conditional

We can use any unequal lambda expressions to stand for 'true' and 'false', but the following work particularly smoothly:

$$\begin{aligned} \text{true} &\triangleq \lambda x\ y.\, x \\ \text{false} &\triangleq \lambda x\ y.\, y \end{aligned}$$

Given those definitions, we can easily define a conditional expression just like C's '`?:`' construct. Note that this is a conditional *expression* not a conditional *command* (this notion makes no sense in our context) and therefore the 'else' arm is compulsory:

$$\text{if } E \text{ then } E_1 \text{ else } E_2 \triangleq E\ E_1\ E_2$$

indeed we have:

$$\begin{aligned}\text{if true then } E_1 \text{ else } E_2 &= \text{true } E_1\ E_2 \\ &= (\lambda x\ y.\ x)\ E_1\ E_2 \\ &= E_1\end{aligned}$$

and

$$\begin{aligned}\text{if false then } E_1 \text{ else } E_2 &= \text{false } E_1\ E_2 \\ &= (\lambda x\ y.\ y)\ E_1\ E_2 \\ &= E_2\end{aligned}$$

Once we have the conditional, it is easy to define all the usual logical operations:

$$\begin{aligned}\text{not } p &\triangleq \text{if } p \text{ then false else true} \\ p \text{ and } q &\triangleq \text{if } p \text{ then } q \text{ else false} \\ p \text{ or } q &\triangleq \text{if } p \text{ then true else } q\end{aligned}$$

3.1.2 Pairs and tuples

We can represent ordered pairs as follows:

$$(E_1, E_2) \triangleq \lambda f.\ f\ E_1\ E_2$$

The parentheses are not obligatory, but we often use them for the sake of familiarity or to enforce associations. In fact we simply regard the comma as an infix operator like $+$. Given the above definition, the corresponding destructors for pairs can be defined as:

$$\begin{aligned}\text{fst } p &\triangleq p \text{ true} \\ \text{snd } p &\triangleq p \text{ false}\end{aligned}$$

It is easy to see that these work as required:

$$\begin{aligned}
\text{fst } (p,q) &= (p,q) \text{ true} \\
&= (\lambda f.\ f\ p\ q) \text{ true} \\
&= \text{true } p\ q \\
&= (\lambda x\ y.\ x)\ p\ q \\
&= p
\end{aligned}$$

and

$$\begin{aligned}
\text{snd } (p,q) &= (p,q) \text{ false} \\
&= (\lambda f.\ f\ p\ q) \text{ false} \\
&= \text{false } p\ q \\
&= (\lambda x\ y.\ y)\ p\ q \\
&= q
\end{aligned}$$

We can build up triples, quadruples, quintuples, indeed arbitrary n-tuples, by iterating the pairing construct:

$$(E_1, E_2, \ldots, E_n) = (E_1, (E_2, \ldots E_n))$$

We need simply say that the infix comma operator is right-associative, and can understand this without any other conventions. For example:

$$\begin{aligned}
(p,q,r,s) &= (p,(q,(r,s))) \\
&= \lambda f.\ f\ p\ (q,(r,s)) \\
&= \lambda f.\ f\ p\ (\lambda f.\ f\ q\ (r,s)) \\
&= \lambda f.\ f\ p\ (\lambda f.\ f\ q\ (\lambda f.\ f\ r\ s)) \\
&= \lambda f.\ f\ p\ (\lambda g.\ g\ q\ (\lambda h.\ h\ r\ s))
\end{aligned}$$

where in the last line we have performed an alpha-conversion for the sake of clarity. Although tuples are built up in a flat manner, it is easy to create arbitrary finitely-branching tree structures by using tupling repeatedly. Finally, if one prefers conventional functions over Cartesian products to our 'curried' functions, one can convert between the two using the following:

$$\begin{aligned}
\text{CURRY } f &\triangleq \lambda x\ y.\ f(x,y) \\
\text{UNCURRY } g &\triangleq \lambda p.\ g\ (\text{fst } p)\ (\text{snd } p)
\end{aligned}$$

These special operations for pairs can easily be generalized to arbitrary n-tuples. For example, we can define a selector function to get the i^{th} component of a flat tuple p. We will write this operation as $(p)_i$ and define $(p)_1 = \text{fst } p$ and the others by $(p)_i = \text{fst } (\text{snd}^{i-1}\ p)$. Likewise we can generalize CURRY and UNCURRY:

$$\begin{aligned} \text{CURRY}_n\ f &\triangleq \lambda x_1\ \cdots\ x_n.\ f(x_1, \ldots, x_n) \\ \text{UNCURRY}_n\ g &\triangleq \lambda p.\ g\ (p)_1\ \cdots\ (p)_n \end{aligned}$$

We can now write $\lambda(x_1, \ldots, x_n).\ t$ as an abbreviation for:

$$\text{UNCURRY}_n\ (\lambda x_1\ \cdots\ x_n.\ t)$$

giving a natural notation for functions over Cartesian products.

3.1.3 The natural numbers

We will represent each natural number n as follows:[3]

$$n \triangleq \lambda f\ x.\ f^n\ x$$

that is, $0 = \lambda f\ x.\ x$, $1 = \lambda f\ x.\ f\ x$, $2 = \lambda f\ x.\ f\ (f\ x)$ etc. These representations are known as *Church numerals*, though the basic idea was used earlier by Wittgenstein (1922).[4] They are not a very efficient representation, amounting essentially to counting in unary, $1, 11, 111, 1111, 11111, 111111, \ldots$. There are, from an efficiency point of view, much better ways of representing numbers, e.g. tuples of trues and falses as binary expansions. But at the moment we are only interested in computability 'in principle' and Church numerals have various nice formal properties. For example, it is easy to give lambda expressions for the common arithmetic operators such as the successor operator which adds one to its argument:

$$\text{SUC} \triangleq \lambda n\ f\ x.\ n\ f\ (f\ x)$$

Indeed

$$\begin{aligned} \text{SUC}\ n &= (\lambda n\ f\ x.\ n\ f\ (f\ x))(\lambda f\ x.\ f^n\ x) \\ &= \lambda f\ x.\ (\lambda f\ x.\ f^n\ x) f\ (f\ x) \\ &= \lambda f\ x.\ (\lambda x.\ f^n\ x)(f\ x) \end{aligned}$$

[3] The n in $f^n\ x$ is just a meta-notation, and does not mean that there is anything circular about our definition.

[4] '6.021 A number is the exponent of an operation'.

$$
\begin{aligned}
&= \lambda f\ x.\ f^{n}\ (f\ x) \\
&= \lambda f\ x.\ f^{n+1}\ x \\
&= n+1
\end{aligned}
$$

Similarly it is easy to test a numeral to see if it is zero:

$$\text{ISZERO}\ n \triangleq n\ (\lambda x.\ \text{false})\ \text{true}$$

since:

$$\text{ISZERO}\ 0 = (\lambda f\ x.\ x)(\lambda x.\ \text{false})\ \text{true} = \text{true}$$

and

$$
\begin{aligned}
\text{ISZERO}\ (n+1) &= (\lambda f\ x.\ f^{n+1} x)(\lambda x.\ \text{false})\text{true} \\
&= (\lambda x.\ \text{false})^{n+1}\ \text{true} \\
&= (\lambda x.\ \text{false})((\lambda x.\ \text{false})^{n}\ \text{true}) \\
&= \text{false}
\end{aligned}
$$

The following perform addition and multiplication on Church numerals:

$$
\begin{aligned}
m+n &\triangleq \lambda f\ x.\ m\ f\ (n\ f\ x) \\
m * n &\triangleq \lambda f\ x.\ m\ (n\ f)\ x
\end{aligned}
$$

Indeed:

$$
\begin{aligned}
m+n &= \lambda f\ x.\ m\ f\ (n\ f\ x) \\
&= \lambda f\ x.\ (\lambda f\ x.\ f^{m}\ x)\ f\ (n\ f\ x) \\
&= \lambda f\ x.\ (\lambda x.\ f^{m}\ x)\ (n\ f\ x) \\
&= \lambda f\ x.\ f^{m}\ (n\ f\ x) \\
&= \lambda f\ x.\ f^{m}\ ((\lambda f\ x.\ f^{n}\ x)\ f\ x) \\
&= \lambda f\ x.\ f^{m}\ ((\lambda x.\ f^{n}\ x)\ x) \\
&= \lambda f\ x.\ f^{m}\ (f^{n}\ x) \\
&= \lambda f\ x.\ f^{m+n} x
\end{aligned}
$$

and:

$$
\begin{aligned}
m * n &= \lambda f\ x.\ m\ (n\ f)\ x \\
&= \lambda f\ x.\ (\lambda f\ x.\ f^{m}\ x)\ (n\ f)\ x
\end{aligned}
$$

$$\begin{aligned}
&= \lambda f\ x.\ (\lambda x.\ (n\ f)^m\ x)\ x \\
&= \lambda f\ x.\ (n\ f)^m\ x \\
&= \lambda f\ x.\ ((\lambda f\ x.\ f^n\ x)\ f)^m\ x \\
&= \lambda f\ x.\ ((\lambda x.\ f^n\ x)^m\ x \\
&= \lambda f\ x.\ (f^n)^m\ x \\
&= \lambda f\ x.\ f^{mn} x
\end{aligned}$$

Although those operations on natural numbers were fairly easy, a 'predecessor' function is much harder. What is required is a a lambda expression PRE so that $PRE\ 0 = 0$ and $PRE\ (n+1) = n$. Finding such an expression was the first piece of mathematical research by the logician Kleene (1935). His trick was, given $\lambda f\ x.\, f^n\ x$, to 'throw away' one of the applications of f. The first step is to define a function 'PREFN' that operates on pairs such that:

$$\text{PREFN}\ f\ (\text{true}, x) = (\text{false}, x)$$

and

$$\text{PREFN}\ f\ (\text{false}, x) = (\text{false}, f\ x)$$

Given this function, then $(\text{PREFN}\ f)^{n+1}(\text{true}, x) = (\text{false}, f^n\ x)$, which is enough to let us define a predecessor function without much difficulty. Here is a suitable definition of 'PREFN':

$$\text{PREFN} \triangleq \lambda f\ p.\ (\text{false}, \text{if fst}\ p\ \text{then snd}\ p\ \text{else}\ f(\text{snd}\ p)$$

Now we define:

$$\text{PRE}\ n \triangleq \lambda f\ x.\ \text{snd}(n\ (\text{PREFN}\ f)\ (\text{true}, x))$$

It is left as an exercise to the reader to see that this works correctly.

3.2 Recursive functions

Being able to define functions by recursion is a central feature of functional programming: it is the only general way of performing something comparable to iteration. At first sight, it appears that we have no way of doing this in lambda calculus. Indeed, it would seem that the *naming* of the function is a vital part of making a recursive definition, since otherwise, how can we refer to it on the right hand side of the definition without descending into infinite regress? Rather surprisingly, it can be done, although as with the existence of a predecessor function, this fact was only discovered after considerable effort.

The key is the existence of so-called *fixed point combinators*. A closed lambda term Y is said to be a fixed point (or fixpoint) combinator when for all lambda terms f, we have $f(Y\ f) = Y\ f$. That is, a fixed point combinator, given any term f as an argument, returns a fixed point for f, i.e. a term x such that $f(x) = x$. The first such combinator to be found (by Curry) is usually called Y. It can be motivated by recalling the Russell paradox, and for this reason is often called the 'paradoxical combinator'. We defined:

$$R = \lambda x.\ \neg(x\ x)$$

and found that:

$$R\ R = \neg(R\ R)$$

That is, $R\ R$ is a fixed point of the negation operator. So in order to get a general fixed point combinator, all we need to do is generalize $\neg$ into whatever function is given it as argument. Therefore we set:

$$Y \triangleq \lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))$$

It is a simple matter to show that it works:

$$\begin{aligned} Yf &= (\lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x)))\ f \\ &= (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x)) \\ &= f((\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))) \\ &= f(Y\ f) \end{aligned}$$

Though valid from a mathematical point of view, this is not so attractive from a programming point of view since the above was only valid for lambda equality, not reduction (in the last line we performed a *backwards* beta conversion). For this reason the following alternative, due to Turing, may be preferred:

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

(The proof that $T\ f \longrightarrow f(T\ f)$ is left as an exercise.) However it does no harm if we simply allow Yf to be beta-reduced throughout the reduction sequence for a recursively defined function. Now let us see how to use a fixed point combinator (say Y) to implement recursive functions. We will take the factorial function as an example. We want to define a function fact with:

$$\mathrm{fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \mathrm{fact}(\mathrm{PRE}\ n)$$

The first step is to transform this into the following equivalent:

$$\mathrm{fact} = \lambda n.\ \text{if ISZERO } n \text{ then } 1 \text{ else } n * \mathrm{fact}(\mathrm{PRE}\ n)$$

which in its turn is equivalent to:

$$\text{fact} = (\lambda f\ n.\ \text{if ISZERO}\ n\ \text{then}\ 1\ \text{else}\ n * f(\text{PRE}\ n))\ \text{fact}$$

This exhibits fact as the fixed point of a function F, namely:

$$F = \lambda f\ n.\ \text{if ISZERO}\ n\ \text{then}\ 1\ \text{else}\ n * f(\text{PRE}\ n)$$

Consequently we merely need to set fact $= Y\ F$. Similar techniques work for mutually recursive functions, i.e. a set of functions whose definitions are mutually dependent. A definition such as:

$$\begin{array}{rcl} f_1 & = & F_1\ f_1\ \cdots\ f_n \\ f_2 & = & F_2\ f_1\ \cdots\ f_n \\ \ldots & = & \ldots \\ f_n & = & F_n\ f_1\ \cdots\ f_n \end{array}$$

can be transformed, using tuples, into a single equation:

$$(f_1, f_2, \ldots, f_n) = (F_1\ f_1\ \cdots\ f_n, F_2\ f_1\ \cdots\ f_n, \ldots, F_n\ f_1\ \cdots\ f_n)$$

If we now write $t = (f_1, f_2, \ldots, f_n)$, then each of the functions on the right can be recovered by applying the appropriate selector to t: we have $f_i = (t)_i$. After abstracting out t, this gives an equation in the canonical form $t = F\ t$ which can be solved by $t = Y\ F$. Now the individual functions can be recovered from t by the same method of selecting components of the tuple.

3.3 Let expressions

We've touted the ability to write anonymous functions as one of the merits of lambda calculus, and shown that names are not necessary to define recursive functions. Nevertheless it is often useful to be able to give names to expressions, to avoid tedious repetition of large terms. A simple form of naming can be supported as another syntactic sugar on top of pure lambda calculus:

$$\text{let}\ x = s\ \text{in}\ t \triangleq (\lambda x.\ t)\ s$$

For example, as we would expect:

$$(\text{let}\ z = 2 + 3\ \text{in}\ z + z) = (\lambda z.\ z + z)\ (2 + 3) = (2 + 3) + (2 + 3)$$

We can achieve the binding of multiple names to expressions either in a serial or parallel fashion. For the first, we will simply iterate the above construct. For the latter, we will write multiple bindings separated by 'and':

$$\text{let } x_1 = s_1 \text{ and } \cdots \text{ and } x_n = s_n \text{ in } t$$

and regard it as syntactic sugar for:

$$(\lambda(x_1, \ldots, x_n).\ t)\ (s_1, \ldots, s_n)$$

To illustrate the distinction between serial and parallel binding, consider:

$$\text{let } x = 1 \text{ in let } x = 2 \text{ in let } y = x \text{ in } x + y$$

which should yield 4, while:

$$\text{let } x = 1 \text{ in let } x = 2 \text{ and } y = x \text{ in } x + y$$

should yield 3. We will also allow arguments to names bound by '`let`'; again this can be regarded as syntactic sugar, with $f\ x_1\ \cdots\ x_n = t$ meaning $f = \lambda x_1\ \cdots\ x_n.\ t$. Instead of a prefix 'let $x = s$ in t', we will allow a postfix version, which is sometimes more readable:

$$t \text{ where } x = s$$

For example, we might write '$y < y^2$ where $y = 1 + x$'. Normally, the 'let' and 'where' constructs will be interpreted as above, in a non-recursive fashion. For example:

$$\text{let } x = x - 1 \text{ in } \cdots$$

binds x to one less than whatever value it is bound to outside, rather than trying to bind x to the fixed point of $x = x-1$.[5] However where a recursive interpretation is required, we add the word 'rec', writing 'let rec' or 'where rec', e.g.

$$\text{let rec fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

This can be regarded as a shorthand for 'let fact $= Y\ F$' where

$$F = \lambda f\ n.\ \text{if ISZERO } n \text{ then } 1 \text{ else } n * f(\text{PRE } n)$$

as explained above.

[5]There is such a fixed point, but the lambda term involved has no normal form, so is in a sense 'undefined'. The semantics of nonterminating terms is subtle, and we will not consider it at length here. Actually, the salient question is not whether a term has a normal form but whether it has a so-called *head normal form* — see Barendregt (1984) and also Abramsky (1990).

3.4 Steps towards a real programming language

We have established a fairly extensive system of 'syntactic sugar' to support a human-readable syntax on top of pure lambda calculus. It is remarkable that this device is enough to bring us to the point where we can write the definition of the factorial function almost in the way we can in ML. In what sense, though, is the lambda calculus augmented with these notational conventions really a programming language?

Ultimately, a program is just a single expression. However, given the use of `let` to bind various important subexpressions to names, it is natural instead to view the program as a set of *definitions* of auxiliary functions, followed finally by the expression itself, e.g:

$$\begin{array}{l} \text{let rec fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n) \text{ in} \\ \cdots \\ fact(6) \end{array}$$

We can read these definitions of auxiliary functions as equations in the mathematical sense. Understood in this way, they do not give any explicit instructions on how to evaluate expressions, or even in which direction the equations are to be used. For this reason, functional programming is often referred to as a *declarative* programming method, as also is logic programming (e.g. PROLOG).[6] The program does not incorporate explicit instructions, merely declares certain properties of the relevant notions, and leaves the rest to the machine.

At the same time, the program is useless, or at least ambiguous, unless the machine somehow reacts sensibly to it. Therefore it must be understood that the program, albeit from a superficial point of view purely declarative, is to be executed in a certain way. In fact, the expression is evaluated by expanding all instances of defined notions (i.e. reading the equations from left to right), and then repeatedly performing β-conversions. That is, although there seems to be no procedural information in the program, *a particular execution strategy is understood.* In a sense then, the term 'declarative' is largely a matter of human psychology.

Moreover, there must be a definite convention for the reduction strategy to use, for we know that different choices of β-redex can lead to different behaviour with respect to termination. Consequently, it is only when we specify this that we really get a definite programming language. We will see in later chapters how different functional languages make different choices here. But first we pause to consider the incorporation of types.

[6] Apparently Landin preferred 'denotative' to 'declarative'.

3.5 Further reading

Many of the standard texts already cited include details about the matters discussed here. In particular Gordon (1988) gives a clear proof that lambda calculus is Turing-complete, and that the lambda calculus analog of the 'halting problem' (whether a term has a normal form) is unsolvable. The influence of lambda calculus on real programming languages and the evolution of functional languages are discussed by Hudak (1989).

Exercises

1. Justify 'generalized β-conversion', i.e. prove that:

$$(\lambda(x_1, \ldots, x_n).\ t[x_1, \ldots, x_n])(t_1, \ldots, t_n) = t[t_1, \ldots, t_n]$$

2. Define $f \circ g \triangleq \lambda x.\ f(g\ x)$, and recall that $I = \lambda x.\ x$. Prove that CURRY $\circ$ UNCURRY $= I$. Is it also true that UNCURRY $\circ$ CURRY $= I$?.

3. What arithmetical operation does $\lambda n\ f\ x.\ f(n\ f\ x)$ perform on Church numerals? What about $\lambda m\ n.\ n\ m$?

4. Prove that the following (due to Klop) is a fixed point combinator:

$$££££££££££££££££££££££££££$$

where:

$$£ \triangleq \lambda abcdefghijklmnopqstuvwxyzr.\ r(thisisafixedpointcombinator)$$

5. Make a recursive definition of natural number subtraction.

6. Consider the following representation of lists, due to Mairson (1991):

$$\begin{array}{rcl}
\text{nil} & = & \lambda c\ n.\ n \\
\text{cons} & = & \lambda x\ l\ c\ n.\ c\ x\ (l\ c\ n) \\
\text{head} & = & \lambda l.\ l(\lambda x\ y.\ x)\ \text{nil} \\
\text{tail} & = & \lambda l.\ \text{snd}\ (l(\lambda x\ p.\ (\text{cons}\ x\ (\text{fst}\ p), \text{fst}\ p))\ (\text{nil}, \text{nil})) \\
\text{append} & = & \lambda l_1\ l_2.\ l_1\ \text{cons}\ l_2 \\
\text{append_lists} & = & \lambda L.\ L\ \text{append}\ \text{nil}
\end{array}$$

$$\begin{aligned}
\text{map} &= \lambda f\ l.\ l\ (\lambda x.\ \text{cons}\ (f\ x))\ \text{nil} \\
\text{length} &= \lambda l.\ l(\lambda x.\ \text{SUC})0 \\
\text{tack} &= \lambda x\ l.\ l\ \text{cons}\ (\text{cons}\ x\ \text{nil}) \\
\text{reverse} &= \lambda l.\ l\ \text{tack}\ \text{nil} \\
\text{filter} &= \lambda l\ test.\ l\ (\lambda x.\ (test\ x)(\text{cons}\ x)(\lambda y.\ y))\ \text{nil}
\end{aligned}$$

Why does Mairson call them 'Church lists'? What do all these functions do?

Chapter 4

Types

Types are a means of distinguishing different sorts of data, like booleans, natural numbers and functions, and making sure that these distinctions are respected, by, for example, ensuring that functions cannot be applied to arguments of the wrong type. Why should we want to add types to lambda calculus, and to programming languages derived from it? We can distinguish reasons from logic and reasons from programming.

From the logical point of view, we have seen that the Russell paradox makes it difficult to extend lambda calculus with set theory without contradiction. The Russell paradox may well arise because of the peculiar self-referential nature of the trick involved: we apply a function to itself. Even if it weren't necessary to avoid paradox, we might feel that intuitively we really have a poor grip of what's going on in a system where we can do such things. Certainly applying some functions to themselves, e.g. the identity function $\lambda x.\, x$ or a constant function $\lambda x.\, y$, seems quite innocuous. But surely we would have a clearer picture of what sort of functions lambda terms denote if we knew exactly what their domains and codomains were, and only applied them to arguments in their domains. These were the sorts of reasons why Russell originally introduced types in *Principia Mathematica*.

Types also arose, perhaps at first largely independently of the above considerations, in programming languages, and since we view lambda calculus as a programming language, this motivates us too. Even FORTRAN distinguished integers and floating point values. One reason was clearly efficiency: the machine can generate more efficient code, and use storage more effectively, by knowing more about a variable. For example the implementation of C's pointer arithmetic varies according to the size of the referenced objects. If p is a pointer to objects occupying 4 bytes, then what C programmers write as $p+1$ becomes $p+4$ inside a byte-addressed machine. C's ancestor BCPL is untyped, and is therefore unable to distinguish pointers from integers; to support the same style of pointer arithmetic it is necessary to apply a scaling at every indirection, a significant penalty.

Apart from efficiency, as time went by, types also began to be appreciated more and more for their value in providing limited static checks on programs. Many programming errors, either trivial typos or more substantial conceptual errors, are suggested by type clashes, and these can be detected before running the program, during the compilation process. Moreover, types often serve as useful documentation for people reading code. Finally, types can be used to achieve better modularization and data hiding by 'artificially' distinguishing some data structure from its internal representation.

At the same time, some programmers dislike types, because they find the restrictions placed on their programming style irksome. There are untyped programming languages, imperative ones like BCPL and functional ones like ISWIM, SASL and Erlang. Others, like PL/I, have only *weak typing*, with the compiler allowing some mixtures of different types and casting them appropriately. There are also languages like LISP that perform typechecking *dynamically*, during execution, rather than statically during compilation. This can (at least in principle) degrade runtime performance, since it means that the computer is spending some time on making sure types are respected during execution — compare checking of array bounds during execution, another common runtime overhead. By contrast, we have suggested that static typing, if anything, can improve performance.

Just how restrictive a type system is found depends a lot on the nature of the programmer's applications and programming style. Finding a type system that allows useful static typechecking, while at the same time allowing programmers plenty of flexibility, is still an active research area. The ML type system is an important step along this road, since it features *polymorphism*, allowing the same function to be used with various different types. This retains strong static typechecking while giving some of the benefits of weak or dynamic typing.[1] Moreover, programmers never need to specify *any* types in ML — the computer can infer a most general type for every expression, and reject expressions that are not typable. We will consider polymorphism in the setting of lambda calculus below. Certainly, the ML type system provides a congenial environment for many kinds of programming. Nevertheless, we do not want to give the impression that it is necessarily the answer to all programming problems.

4.1 Typed lambda calculus

It is a fairly straightforward matter to modify lambda calculus with a notion of type, but the resulting changes, as we shall see, are far-reaching. The basic idea is that every lambda term has a *type*, and a term s can only be applied to a term t in a combination $s\ t$ when the types match up appropriately, i.e. s has the type of a function $\sigma \to \tau$ and t has type σ. The result, $s\ t$, then has type τ. This is, in programming language parlance, *strong typing*. The term t must have *exactly*

[1] As well, perhaps, as some of the overheads.

the type σ; there is no notion of subtyping or coercion. This contrasts with some programming languages like C where a function expecting an argument of type `float` or `double` will accept one of type `int` and appropriately cast it. Similar notions of subtyping and coercion can also be added to lambda calculus, but to consider these here would lead us too far astray.

We will use '$t : \sigma$' to mean 't has type σ'. This is already the standard notation in mathematics where function spaces are concerned, because $f : \sigma \to \tau$ means that f is a function from the set σ to the set τ. We will think of types as sets in which the corresponding objects live, and imagine $t : \sigma$ to mean $t \in \sigma$. Though the reader may like to do likewise, as a heuristic device, we will view typed lambda calculus purely as a formal system and make the rules independent of any such interpretation.

4.1.1 The stock of types

The first stage in our formalization is to specify exactly what the types are. We suppose first of all that we have some set of *primitive types*, which might, for example, contain `bool` and `int`. We can construct composite types from these using the function space *type constructor*. Formally, we make the following inductive definition of the set Ty_C of types based on a set of primitive types C:

$$\frac{\sigma \in C}{\sigma \in Ty_C}$$

$$\frac{\sigma \in Ty_C \quad \tau \in Ty_C}{\sigma \to \tau \in Ty_C}$$

For example, possible types might be *int*, *bool* $\to$ *bool* and (*int* $\to$ *bool*) $\to$ *int* $\to$ *bool*. We assume that the function arrow associates to the right, i.e. $\sigma \to \tau \to \upsilon$ means $\sigma \to (\tau \to \upsilon)$. This fits naturally with the other syntactic conventions connected with currying.

Before long, we shall extend the type system in two ways. First, we will allow so-called *type variables* as well as primitive type constants — these are the vehicle for polymorphism. Secondly, we will allow the introduction of other type constructors besides the function arrow. For example, we will later introduce a constructor $\times$ for the product type. In that case a new clause:

$$\frac{\sigma \in Ty_C \quad \tau \in Ty_C}{\sigma \times \tau \in Ty_C}$$

needs to be added to the inductive definition. Once we move to the more concrete case of ML, there is the possibility of new, user-defined types and type constructors, so we will get used to arbitrary sets of constructors of any arity. We will write $(\alpha_1, \ldots, \alpha_n)con$ for the application of an n-ary type constructor con to the arguments α_i. (Only in a few special cases like $\to$ and $\times$ do we use infix syntax, for the sake of familiarity.) For example $(\sigma)list$ is the type of lists whose elements all have type σ.

An important property of the types, provable because of *free* inductive generation, is that $\sigma \to \tau \neq \sigma$. (In fact, more generally, a type cannot be the same as any proper syntactic subexpression of itself.) This rules out the possibility of applying a term to itself, unless the two instances in question have distinct types.

4.1.2 Church and Curry typing

There are two major approaches to defining typed lambda calculus. One approach, due to Church, is *explicit*. A (single) type is attached to each term. That is, in the construction of terms, the untyped terms that we have presented are modified with an additional field for the type. In the case of constants, this type is pre-assigned, but variables may be of any type. The rules for valid term formation are then:

$$\frac{}{v : \sigma}$$

$$\frac{\text{Constant } c \text{ has type } \sigma}{c : \sigma}$$

$$\frac{s : \sigma \to \tau \quad t : \sigma}{s\ t : \tau}$$

$$\frac{v : \sigma \quad t : \tau}{\lambda v.\, t : \sigma \to \tau}$$

For our purposes, however, we prefer Curry's approach to typing, which is purely *implicit*. The terms are exactly as in the untyped case, and a term may or may not have a type, and if it does, may have many different types.[2] For example, the identity function $\lambda x.\, x$ may be given any type of the form $\sigma \to \sigma$, reasonably enough. There are two related reasons why we prefer this approach. First, it provides a smoother treatment of ML's polymorphism, and secondly, it corresponds well to the actual use of ML, where it is never necessary to specify types explicitly.

At the same time, some of the formal details of Curry-style type assignment are a bit more complex. We do not merely define a relation of typability in isolation, but with respect to a *context*, i.e. a finite set of typing assumptions about variables. We write:

$$\Gamma \vdash t : \sigma$$

[2] Some purists would argue that this isn't properly speaking 'typed lambda calculus' but rather 'untyped lambda calculus with a separate notion of type assignment'.

to mean 'in context Γ, the term t can be given type σ'. (We will, however, write $\vdash t : \sigma$ or just $t : \sigma$ when the typing judgement holds in the empty context.) The elements of Γ are of the form $v : \sigma$, i.e. they are themselves typing assumptions about variables, typically those that are components of the term. We assume that Γ never gives contradictory assignments to the same variable; if preferred we can think of it as a partial function from the indexing set of the variables into the set of types. The use of the symbol $\vdash$ corresponds to its general use for judgements in logic of the form $\Gamma \vdash \phi$, read as 'ϕ follows from assumptions Γ'.

4.1.3 Formal typability rules

The rules for typing expressions are fairly natural. Remember the interpretation of $t : \sigma$ as 't *could* be given type σ'.

$$\frac{v : \sigma \in \Gamma}{\Gamma \vdash v : \sigma}$$

$$\frac{\text{Constant } c \text{ has type } \sigma}{c : \sigma}$$

$$\frac{\Gamma \vdash s : \sigma \to \tau \quad \Gamma \vdash t : \sigma}{\Gamma \vdash s\ t : \tau}$$

$$\frac{\Gamma \cup \{v : \sigma\} \vdash t : \tau}{\Gamma \vdash \lambda v.\, t : \sigma \to \tau}$$

Once again, this is to be regarded as an inductive definition of the typability relation, so a term only has a type if it can be derived by the above rules. For example, let us see how to type the identity function. By the rule for variables we have:

$$\{x : \sigma\} \vdash x : \sigma$$

and therefore by the last rule we get:

$$\emptyset \vdash \lambda x.\, x : \sigma \to \sigma$$

By our established convention for empty contexts, we write simply $\lambda x.\, x : \sigma \to \sigma$. This example also illustrates the need for the context in Curry typing, and why it can be avoided in the Church version. Without the context we could deduce $x : \tau$ for any τ, and then by the last rule get $\lambda x.\, x : \sigma \to \tau$, clearly at variance with the intuitive interpretation of the identity function. This problem doesn't arise in Church typing, since in that case either both variables have type σ, in which case the rules imply that $\lambda x.x : \sigma \to \sigma$, or else the two x's are actually *different variables*, since their types differ and the types are a component of the term. Now, indeed, $\lambda x : \sigma.\, (x : \tau) : \sigma \to \tau$, but this is reasonable because the term is alpha-equivalent to $\lambda x : \sigma. (y : \tau) : \sigma \to \tau$. In the Curry version of typing, the types are not attached to the terms, and so we need some way of connecting instances of the same variable.

4.1.4 Type preservation

Since the underlying terms of our typed lambda calculus are the same as in the untyped case, we can take over the calculus of conversions unchanged. However, for this to make sense, we need to check that all the conversions preserve typability, a property called *type preservation.* This is not difficult; we will sketch a formal proof for η-conversion and leave the other, similar cases to the reader. First we prove a couple of lemmas that are fairly obvious but which need to be established formally. First, adding new entries to the context cannot inhibit typability:

Lemma 4.1 *If $\Gamma \vdash t : \sigma$ and $\Gamma \subseteq \Delta$ then $\Delta \vdash t : \sigma$.*
Proof: *By induction on the structure of t. Formally we are fixing t and proving the above for all Γ and Δ, since in the inductive step for abstractions, the sets concerned change. If t is a variable we must have that $t : \sigma \in \Gamma$. Therefore $t : \sigma \in \Delta$ and the result follows. If t is a constant the result is immediate since the stock of constants and their possible types is independent of the context. If t is a combination $s\ u$ then we have, for some type τ, that $\Gamma \vdash s : \tau \to \sigma$ and $\Gamma \vdash u : \tau$. By the inductive hypothesis, $\Delta \vdash s : \tau \to \sigma$ and $\Delta \vdash u : \tau$ and the result follows. Finally, if t is an abstraction $\lambda x.\, s$ then we must have, by the last rule, that σ is of the form $\tau \to \tau'$ and that $\Gamma \cup \{x : \tau\} \vdash s : \tau'$. Since $\Gamma \subseteq \Delta$ we also have $\Gamma \cup \{x : \tau\} \subseteq \Delta \cup \{x : \tau\}$, and so by the inductive hypothesis $\Delta \cup \{x : \tau\} \vdash s : \tau'$. By applying the rule for abstractions, the result follows. Q.E.D.*

Secondly, entries in the context for variables not free in the relevant term may be discarded.

Lemma 4.2 *If $\Gamma \vdash t : \sigma$, then $\Gamma_t \vdash t : \sigma$ where Γ_t contains only entries for variables free in t, or more formally $\Gamma_t = \{x : \alpha \mid x : \alpha \in \Gamma \text{ and } x \in FV(t)\}$.*
Proof: *Again, we prove the above for all Γ and the corresponding Γ_t by structural induction over t. If t is a variable, then since $\Gamma \vdash t : \sigma$ we must have an entry $x : \sigma$ in the context. But by the first rule we know $\{x : \sigma\} \vdash x : \sigma$ as required. In the case of a constant, then the typing is independent of the context, so the result holds trivial ly. If t is a combination $s\ u$ then we have for some τ that $\Gamma \vdash s : \tau \to \sigma$ and $\Gamma \vdash u : \tau$. By the inductive hypothesis, we have $\Gamma_s \vdash s : \tau \to \sigma$ and $\Gamma_u \vdash u : \tau$. By the monotonicity lemma, we therefore have $\Gamma_{su} \vdash s : \tau \to \sigma$ and $\Gamma_{su} \vdash u : \tau$, since $FV(s\ u) = FV(s) \cup FV(u)$. Hence by the rule for combinations we get $\Gamma_{su} \vdash t : \sigma$. Finally, if t has the form $\lambda x.\, s$, we must have*

$\Gamma \cup \{x : \tau\} \vdash s : \tau'$ with σ of the form $\tau \to \tau'$. By the inductive hypothesis we have $(\Gamma \cup \{x : \tau\})_s \vdash s : \tau'$, and so $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \vdash (\lambda x.\ s) : \sigma$. Now we need simply observe that $(\Gamma \cup \{x : \tau\})_s - \{x : \tau\} \subseteq \Gamma_t$ and appeal once more to monotonicity. Q.E.D.

Now we come to the main result.

Theorem 4.3 *If $\Gamma \vdash t : \sigma$ and $t \underset{\eta}{\longrightarrow} t'$, then also $\Gamma \vdash t' : \sigma$*

Proof: *Since by hypothesis t is an η-redex, it must be of the form $(\lambda x.t\ x)$ with $x \notin FV(t)$. Now, its type can only have arisen by the last typing rule, and therefore σ must be of the form $\tau \to \tau'$, and we must have $\{x : \tau\} \vdash (t\ x) : \tau'$. Now, this typing can only have arisen by the rule for combinations. Since the context can only give one typing to each variable, we must therefore have $\{x : \tau\} \vdash t : \tau \to \tau'$. But since by hypothesis, $x \notin FV(t)$, the lemma yields $\vdash t : \tau \to \tau'$ as required. Q.E.D.*

Putting together all these results for other conversions, we see that if $\Gamma \vdash t : \sigma$ and $t \longrightarrow t'$, then we have $\Gamma \vdash t' : \sigma$. This is reassuring, since if the evaluation rules that are applied during execution could change the types of expressions, it would cast doubt on the whole enterprise of static typing.

4.2 Polymorphism

The Curry typing system already gives us a form of *polymorphism*, in that a given term may have different types. Let us start by distinguish between the similar concepts of polymorphism and *overloading*. Both of them mean that an expression may have multiple types. However in the case of polymorphism, all the types bear a systematic relationship to each other, and all types following the pattern are allowed. For example, the identity function may have type $\sigma \to \sigma$, or $\tau \to \tau$, or $(\sigma \to \tau) \to (\sigma \to \tau)$, but all the instances have the same structure. By contrast, in overloading, a given function may have different types that bear no structural relation to each other, or only certain instances may be selected. For example, a function + may be allowed to have type $int \to int \to int$ or type $float \to float \to float$, but not $bool \to bool \to bool$.[3] Yet a third related notion is subtyping, which is a more principled version of overloading, regarding certain types as embedded in others. This, however, turns out to be a lot more complex than it seems.[4]

[3]Strachey, who coined the term 'polymorphism', referred to what we call polymorphism and overloading as *parametric* and *ad hoc* polymorphism respectively.

[4]For example, if α is a subtype of α', should $\alpha \to \beta$ be a subtype or a supertype of $\alpha' \to \beta$? One can defend either interpretation ('covariant' or 'contravariant') in certain situations.

4.2.1 Let polymorphism

Unfortunately the type system we have so far places some unpleasant restrictions on polymorphism. For example, the following expression is perfectly acceptable:

$$\text{if } (\lambda x.\ x) \text{ true then } (\lambda x.\ x)\ 1 \text{ else } 0$$

Here is a proof according to the above rules. We assume that the constants can be typed in the empty environment, and we fold together two successive applications of the rule for combinations as a rule for `if`, remembering that 'if b then t_1 else t_2' is shorthand for 'COND b t_1 t_2'.

$$\dfrac{\dfrac{\dfrac{\{x : bool\} \vdash x : bool}{\vdash (\lambda x.\ x) : bool \to bool} \quad \dfrac{}{\vdash \text{true} : bool}}{\vdash (\lambda x.\ x) \text{ true} : bool} \quad \dfrac{\dfrac{\{x : int\} \vdash x : int}{\vdash (\lambda x.\ x) : int \to int} \quad \dfrac{}{\vdash 1 : int}}{\vdash (\lambda x.\ x)\ 1 : int} \quad \dfrac{}{\vdash 0 : int}}{\vdash \text{if } (\lambda x.\ x) \text{ true then } (\lambda x.\ x)\ 1 \text{ else } 0 : int}$$

The two instances of the identity function get types $bool \to bool$ and $int \to int$. But consider the following:

$$\text{let } I = \lambda x.\ x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0$$

According to our definitions, this is just syntactic sugar for:

$$(\lambda I.\ \text{if } I \text{ true then } I\ 1 \text{ else } 0)\ (\lambda x.\ x)$$

As the reader may readily confirm, this cannot be typed according to our rules. There is just *one* instance of the identity function and it must be given a single type. This feature is unbearable, since in practical functional programming we rely heavily on using `let`. Unless we change the typing rules, many of the benefits of polymorphism are lost. Our solution is to make `let` primitive, rather than interpret it as syntactic sugar, and add the new typing rule:

$$\frac{\Gamma \vdash s : \sigma \quad \Gamma \vdash t[s/x] : \tau}{\Gamma \vdash \text{let } x = s \text{ in } t : \tau}$$

This rule, which sets up *let polymorphism*, expresses the fact that, with respect to typing at least, we treat `let` bindings just as if separate instances of the named expression were written out. The additional hypothesis $\Gamma \vdash s : \sigma$ merely ensures that s is typable, the exact type being irrelevant. This is to avoid admitting as well-typed terms such as:

$$\text{let } x = \lambda f.\ f\ f \text{ in } 0$$

Now, for example, we can type the troublesome example:

$$\dfrac{\dfrac{\overline{\{x : \sigma\} \vdash x : \sigma}}{\vdash \lambda x.\, x : \sigma \to \sigma} \quad \dfrac{\textbf{As given above}}{\vdash \text{if } (\lambda x.\, x) \text{ true then } (\lambda x.\, x)\ 1 \text{ else } 0 : int}}{\vdash \text{let } I = \lambda x.\, x \text{ in if } I \text{ true then } I\ 1 \text{ else } 0 : int}$$

4.2.2 Most general types

As we have said, some expressions have no type, e.g. $\lambda f.\ f\ f$ or $\lambda f.\ (f \text{ true}, f\ 1)$. Typable expressions normally have many types, though some, e.g. *true*, have exactly one. We have already commented that the form of polymorphism in ML is *parametric*, i.e. all the different types an expression can have bear a structural similarity to each other. In fact more is true: there exists for each typable expression a *most general type* or *principal type*, and all possible types for the expression are instances of this most general type. Before stating this precisely we need to establish some terminology.

First, we extend our notion of types with *type variables*. That is, types can be built up by applying type constructors either to type constants or variables. Normally we use the letters α and β to represent type variables, and σ and τ arbitrary types. Given this extension, we can define what it means to substitute one type for a type variable in another type. This is precisely analogous to substitution at the term level, and we even use the same notation. For example $(\sigma \to bool)[(\sigma \to \tau)/\sigma] = (\sigma \to \tau) \to bool$. The formal definition is much simpler than at the term level, because we do not have to worry about bindings. At the same time, it is convenient to extend it to multiple parallel substitutions:

$$\begin{array}{rcl}
\alpha_i[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_k] & = & \tau_i \text{ if } \alpha_i \neq \beta \text{ for } 1 \leq i \leq k \\
\beta[\tau_1/\alpha_1, \ldots, \tau_n/\alpha_k] & = & \beta \text{ if } \alpha_i \neq \beta \text{ for } 1 \leq i \leq k \\
(\sigma_1, \ldots, \sigma_n)con[\theta] & = & (\sigma_1[\theta], \ldots, \sigma_n[\theta])con
\end{array}$$

For simplicity we treat type constants as nullary constructors, e.g. consider $()int$ rather than int, but if preferred it is easy to include them as a special case. Given this definition of substitution, we can define the notion of a type σ being *more general* than a type σ', written $\sigma \preceq \sigma'$.[5] This is defined to be true precisely if there is a set of substitutions θ such that $\sigma' = \sigma\theta$. For example we have:

$$\begin{array}{rcl}
\alpha & \preceq & \sigma \\
\alpha \to \alpha & \preceq & \beta \to \beta \\
\alpha \to bool & \preceq & (\beta \to \beta) \to bool
\end{array}$$

[5] This relation is reflexive, so we should really say 'at least as general as' rather than 'more general than'.

$$\begin{array}{rcl} \beta \to \alpha & \preceq & \alpha \to \beta \\ \alpha \to \alpha & \not\preceq & (\beta \to \beta) \to \beta \end{array}$$

Now we come to the main theorem:

Theorem 4.4 *Every typable term has a principal type, i.e. if $t : \tau$, then there is some σ such that $t : \sigma$ and for any σ', if $t : \sigma'$ then $\sigma \preceq \sigma'$.*

It is easy to see that the relation $\preceq$ is a preorder relation, i.e. is reflexive and transitive. The principal type is not unique, but it is unique up to renaming of type variables. More precisely, if σ and τ are both principal types for an expression, then $\sigma \sim \tau$, meaning that $\sigma \preceq \tau$ and $\tau \preceq \sigma$.

We will not even sketch a proof of the principal type theorem; it is not particularly difficult, but is rather long. What is important to understand is that the proof gives a concrete procedure for finding a principal type. This procedure is known as *Milner's algorithm*, or often the Hindley-Milner algorithm.[6] All implementations of ML, and several other functional languages, incorporate a version of this algorithm, so expressions can automatically be allocated a principal type, or rejected as ill-typed.

4.3 Strong normalization

Recall our examples of terms with no normal form, such as:

$$\begin{array}{rl} & ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & ((\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)\ (\lambda x.\ x\ x\ x)) \\ \longrightarrow & (\cdots) \end{array}$$

In typed lambda calculus, this cannot happen by virtue of the following *strong normalization* theorem, whose proof is too long for us to give here.

Theorem 4.5 *Every typable term has a normal form, and every possible reduction sequence starting from a typable term terminates.*[7]

At first sight this looks good — a functional program respecting our type discipline can be evaluated in any order, and it will always terminate in the unique normal form. (The uniqueness comes from the Church-Rosser theorem, which is

[6]The above principal type theorem in the form we have presented it is due to Milner (1978), but the theorem and an associated algorithm had earlier been discovered by Hindley in a similar system of combinatory logic.

[7]Weak normalization means the first part only, i.e. every term can be reduced to normal form, but there may still be reduction sequences that fail to terminate.

still true in the typed system.) However, the ability to write nonterminating functions is essential to Turing completeness,[8] so we are no longer able to define all computable functions, not even all total ones.

This might not matter if we could define all 'practically interesting' functions. However this is not so; the class of functions definable as typed lambda terms is fairly limited. Schwichtenberg (1976) has proved that if Church numerals are used, then the definable functions are those that are polynomials or result from polynomials using definition by cases. Note that this is strictly intensional: if another numeral representation is used, a different class of functions is obtained. In any case, it is not enough for a general programming language.

Since all definable functions are total, we are clearly unable to make arbitrary recursive definitions. Indeed, the usual fixpoint combinators must be untypable; $Y = \lambda f.\ (\lambda x.\ f(x\ x))(\lambda x.\ f(x\ x))$ clearly isn't well-typed because it applies x to itself and x is bound by a lambda. In order to regain Turing-completeness we simply add a way of defining arbitrary recursive functions that *is* well-typed. We introduce a polymorphic recursion operator with all types of the form:

$$Rec : ((\sigma \to \tau) \to (\sigma \to \tau)) \to \sigma \to \tau$$

and the extra reduction rule that for any $F : (\sigma \to \tau) \to (\sigma \to \tau)$ we have:

$$Rec\ F \longrightarrow F\ (Rec\ F)$$

From now on, we will suppose that recursive definitions, signalled by 'let rec', map down to the use of these recursion operators.

Further reading

Barendregt (1984) and Hindley and Seldin (1986) also consider typed lambda calculus. The original paper by Milner (1978) is still a good source of information about polymorphic typing and the algorithm for finding principal types. A nice elementary treatment of typed lambda calculus, which gives a proof of Strong Normalization and discusses some interesting connections with logic, is Girard, Lafont, and Taylor (1989). This also discusses a more advanced version of typed lambda calculus called 'System F' where even though strong normalization holds, the majority of 'interesting' functions are definable.

Exercises

1. Is it true that if $\Gamma \vdash t : \sigma$ then for any substitution θ we have $\Gamma \vdash t : (\sigma\theta)$?

[8] Given any recursive enumeration of total computable functions, we can always create one not in the list by diagonalization. This will be illustrated in the Computability Theory course.

2. Prove formally the type preservation theorems for α and β conversion.

3. Show that the converse to type preservation does not hold, i.e. it is possible that $t \longrightarrow t'$ and $\Gamma \vdash t' : \sigma$ but not $\Gamma \vdash t : \sigma$.

4. (*) Prove that every term of typed lambda calculus with *principal* type $(\alpha \to \alpha) \to (\alpha \to \alpha)$ reduces to a Church numeral.

5. (*) To what extent can the typechecking process be reversed, i.e. the term inferred from the type? For instance, is it true that in pure typed lambda calculus with type variables but no constants and no recursion operator, that any $t : \alpha \to \beta \to \alpha$ is in fact equal to $K = \lambda x\ y.\ x$ in the usual sense of lambda equality? If so, how far can this be generalized?[9]

6. (*) We will say that an arbitrary reduction relation $\longrightarrow$ is *weakly Church-Rosser* if whenever $t \longrightarrow t_1$ and $t \longrightarrow t_2$, then there is a u with $t_1 \longrightarrow^* u$ and $t_2 \longrightarrow^* u$, where $\longrightarrow^*$ is the reflexive transitive closure of $\longrightarrow$. Prove *Newman's Lemma*, which states that if a relation is weakly Church-Rosser and obeys strong normalization, then $\longrightarrow^*$ is Church-Rosser. (Hint: use a form of wellfounded induction.)[10]

[9] See Mairson (1991) for more on this question.
[10] If you get stuck, there is a nice proof in Huet (1980).

Chapter 5

A taste of ML

In the last few chapters, we have started with pure lambda calculus and systematically extended it with new primitive features. For example we added the 'let' construct as primitive for the sake of making the polymorphic typing more useful, and a recursion operation in order to restore the computational power that was lost when adding types. By travelling further along this path, we will eventually arrive at ML, although it's still valuable to retain the simple worldview of typed lambda calculus.

The next stage is to abandon the encodings of datatypes like booleans and natural numbers via lambda terms, and instead make them primitive. That is, we have new primitive types like `bool` and `int` (actually positive and negative integers) and new type constructors like $\times$, a step we have in many ways anticipated in the last chapter. Associated with these are new constants and new conversion rules. For example, the expression $2 + 2$ is evaluated using machine arithmetic, rather than by translation into Church numerals and performing β-conversions. These additional conversions, regarded as augmenting the usual lambda operations, are often referred to as 'δ-conversions'. We will see in due course how the ML language is extended in other ways in comparison with pure lambda calculus. First, we must address the fundamental question of ML's evaluation strategy.

5.1 Eager evaluation

We have said that from a theoretical point of view, normal order (top down left-to-right) reduction of expressions is to be preferred, since if any strategy terminates, this one will.[1] However it has some practical defects. For example, consider the following:

$$(\lambda x.\ x + x + x)\ (10 + 5)$$

[1]This strategy is familiar from a few traditional languages like Algol 60 where it is referred to as *call by name*.

A normal order reduction results in $(10 + 5) + (10 + 5) + (10 + 5)$, and the following reduction steps need to evaluate three separate instances of the same expression. This is unacceptable in practice. There are two main solutions to this problem, and these solutions divide the world of functional programming languages into two camps.

The first alternative is to stick with normal order reduction, but attempt to optimize the implementation so multiple subexpressions arising in this way are shared, and never evaluated more than once. Internally, expressions are represented as directed acyclic graphs rather than as trees. This is known as *lazy* or *call-by-need* evaluation, since expressions are only evaluated when absolutely necessary.

The second approach is to turn the theoretical considerations about reduction strategy on their head and evaluate arguments to functions before passing the values to the function. This is known as *applicative order* or *eager* evaluation. The latter name arises because function arguments are evaluated even when they may be unnecessary, e.g. t in $(\lambda x.\ y)\ t$. Of course, eager evaluation means that some expressions may loop that would terminate in a lazy regime. But this is considered acceptable, since these instances are generally easy to avoid in practice. In any case, an eager evaluation strategy is the standard in many programming languages like C, where it is referred to as *call by value*.

ML adopts eager evaluation, for two main reasons. Choreographing the reductions and sharings that occur in lazy evaluation is quite tricky, and implementations tend to be relatively inefficient and complicated. Unless the programmer is very careful, memory can fill up with pending unevaluated expressions, and in general it is hard to understand the space behaviour of programs. In fact many implementations of lazy evaluation try to optimize it to eager evaluation in cases where there is no semantic difference.[2] By contrast, in ML, we always first evaluate the arguments to functions and only then perform the β-reduction — this is simple and efficient, and is easy to implement using standard compiler technology.

The second reason for preferring applicative order evaluation is that ML is not a *pure* functional language, but includes imperative features (variables, assignments etc.). Therefore the order of evaluation of subexpressions can make a *semantic* difference, rather than merely affecting efficiency. If lazy evaluation is used, it seems to become practically impossible for the programmer to visualize, in a nontrivial program, exactly when each subexpression gets evaluated. In the eager ML system, one just needs to remember the simple evaluation rule.

It is important to realize, however, that the ML evaluation strategy is *not* simply bottom-up, the exact opposite of normal order reduction. In fact, ML *never evaluates underneath a lambda*. (In particular, it never reduces η-redexes,

[2] They try to perform *strictness analysis*, a form of static analysis that can often discover that arguments are necessarily going to be evaluated (Mycroft 1981).

only β-redexes.) In evaluating $(\lambda x.\ s[x])\ t$, t is evaluated first. However $s[x]$ is not touched since it is underneath a lambda, and any subexpressions of t that happen to be in the scope of a lambda are also left alone. The precise evaluation rules are as follows:

- Constants evaluate to themselves.
- Evaluation stops immediately at λ-abstractions and does not look inside them. In particular, there is no η-conversion.
- When evaluating a combination $s\ t$, then *first* both s and t are evaluated. Then, assuming that the evaluated form of s is a lambda abstraction, a toplevel β-conversion is performed and the process is repeated.

The order of evaluation of s and t varies between versions of ML. In the version we will use, t is always evaluated first. Strictly, we should also specify a rule for `let` expressions, since as we have said they are by now accepted as a primitive. However from the point of view of evaluation they can be understood in the original way, as a lambda applied to an argument, with the understanding that the rand will be evaluated first. To make this explicit, the rule for

$$\text{let } x = s \text{ in } t$$

is that first of all s is evaluated, the resulting value is substituted for x in t, and finally the new version of t is evaluated. Let us see some examples of evaluating expressions:

$$\begin{aligned}(\lambda x.\ (\lambda y.\ y + y)\ x)(2 + 2) &\longrightarrow (\lambda x.\ (\lambda y.\ y + y)\ x)4 \\ &\longrightarrow (\lambda y.\ y + y)4 \\ &\longrightarrow 4 + 4 \\ &\longrightarrow 8\end{aligned}$$

Note that the subterm $(\lambda y.\ y + y)\ x$ is *not reduced*, since it is within the scope of a lambda. However, terms that are reducible and *not* enclosed by a lambda in both function and argument get reduced before the function application itself is evaluated, e.g. the second step in the following:

$$\begin{aligned}((\lambda f\ x.\ f\ x)\ (\lambda y.\ y + y))\ (2 + 2) &\longrightarrow ((\lambda f\ x.\ f\ x)\ (\lambda y.\ y + y))\ 4 \\ &\longrightarrow (\lambda x.\ (\lambda y.\ y + y)\ x)\ 4 \\ &\longrightarrow (\lambda y.\ y + y)\ 4 \\ &\longrightarrow 4 + 4\end{aligned}$$

The fact that ML does not evaluate under lambdas is of crucial importance to advanced ML programmers. It gives precise control over the evaluation of expressions, and can be used to mimic many of the helpful cases of lazy evaluation. We shall see a very simple instance in the next section.

5.2 Consequences of eager evaluation

The use of eager evaluation forces us to make additional features primitive, with their own special reduction procedures, rather than implementing them directly in terms of lambda calculus. In particular, we can no longer regard the conditional construct:

$$\text{if } b \text{ then } e_1 \text{ else } e_2$$

as the application of an ordinary ternary operator:

$$\text{COND } b\ e_1\ e_2$$

The reason is that in any particular instance, because of eager evaluation, we evaluate all the expressions b, e_1 and e_2 first, before evaluating the body of the COND. Usually, this is disastrous. For example, recall our definition of the factorial function:

$$\text{let rec fact}(n) = \text{if ISZERO } n \text{ then } 1 \text{ else } n * \text{fact}(\text{PRE } n)$$

If the conditional evaluates all its arguments, then in order to evaluate fact(0), the 'else' arm must be evaluated, in its turn causing fact(PRE 0) to be evaluated. This in its turn causes the evaluation of fact(PRE (PRE 0)), and so on. Consequently the computation will loop indefinitely.

Accordingly, we make the conditional a new primitive construct, and modify the usual reduction strategy so that *first* the boolean expression is evaluated, and only then is *exactly one* of the two arms evaluated, as appropriate.

What about the process of recursion itself? We have suggested understanding recursive definitions in terms of a recursion operator *Rec* with its own reduction rule:

$$Rec\ f \longrightarrow f(Rec\ f)$$

This would also loop using an eager evaluation strategy:

$$Rec\ f \longrightarrow f(Rec\ f) \longrightarrow f(f(Rec\ f)) \longrightarrow f(f(f(Rec\ f))) \longrightarrow \cdots$$

However we only need a very simple modification to the reduction rule to make things work correctly:

$$Rec\ f \longrightarrow f(\lambda x.\ Rec\ f\ x)$$

Now the lambda on the right means that $\lambda x.\ Rec\ f\ x$ evaluates to itself, and so only after the expression has been further reduced following substitution in the body of f will evaluation continue.

5.3 The ML family

We have been talking about 'ML' as if it were a single language. In fact there are many variants of ML, even including 'Lazy ML', an implementation from Chalmers University in Sweden that is based on lazy evaluation. The most popular version of ML in education is 'Standard ML', but we will use another version, called CAML ('camel') Light.[3] We chose CAML Light for several reasons:

- The implementation is small and highly portable, so can be run effectively on Unix machines, PCs, Macs etc.
- The system is simple, syntactically and semantically, making it relatively easy to approach.
- The system is well suited to practical programming, e.g. it can easily be interfaced to C and supports standard `make`-compatible separate compilation.

However, we will be teaching quite general techniques, and any code we write can be run with a few minor syntactic changes in any version of ML, and indeed often in other functional languages.

5.4 Starting up ML

ML has already been installed on Thor. In order to use it, you need to put the CAML binary directory on your `PATH`. This can be done (assuming you use the 'bash' shell or one of its family), as follows:

```
PATH="$PATH:/home/jrh13/caml/bin"
export PATH
```

To avoid doing this every time you log on, you can insert these lines near the end of your `.bash_profile`, or the equivalent for your favourite shell. Now to use CAML in its normal interactive and interpretive mode, you simply need to type `camllight`, and the system should fire up and present its prompt ('`#`'):

[3] The name stands for 'Categorical Abstract Machine', the underlying implementation method.

```
$ camllight
>       Caml Light version 0.73

#
```

In order to exit the system, simply type `ctrl/d` or `quit();;` at the prompt. If you are interested in installing CAML Light on your own machine, you should consult the following Web page for detailed information:

```
http://pauillac.inria.fr/caml/
```

5.5 Interacting with ML

When ML presents you with its prompt, you can type in expressions, terminated by two successive semicolons, and it will evaluate them and print the result. In computing jargon, the ML system sits in a read-eval-print loop: it repeatedly reads an expression, evaluates it, and prints the result. For example, ML can be used as a simple calculator:

```
#10 + 5;;
- : int = 15
```

The system not only returns the answer, but also the *type* of the expression, which it has inferred automatically. It can do this because it knows the type of the built-in addition operator `+`. On the other hand, if an expression is not typable, the system will reject it, and try to give some idea about how the types fail to match up. In complicated cases, the error messages can be quite tricky to understand.

```
#1 + true;;
Toplevel input:
>let it = 1 + true;;
>             ^^^^
This expression has type bool,
but is used with type int.
```

Since ML is a functional language, expressions are allowed to have function type. The ML syntax for a lambda abstraction $\lambda x.\ t[x]$ is `fun x -> t[x]`. For example we can define the successor function:

```
#fun x -> x + 1;;
- : int -> int = <fun>
```

Again, the type of the expression, this time `int -> int`, is inferred and displayed. However the function itself is not printed; the system merely writes `<fun>`. This is because, in general, the internal representations of functions are not very readable.[4] Functions are applied to arguments just as in lambda calculus, by juxtaposition. For example:

```
#(fun x -> x + 1) 4;;
- : int = 5
```

Again as in lambda calculus, function application associates to the left, and you can write curried functions using the same convention of eliding multiple λ's (i.e. `fun`'s). For example, the following are all equivalent:

```
#((fun x -> (fun y -> x + y)) 1) 2;;
- : int = 3
#(fun x -> fun y -> x + y) 1 2;;
- : int = 3
#(fun x y -> x + y) 1 2;;
- : int = 3
```

5.6 Bindings and declarations

It is not convenient, of course, to evaluate the whole expression in one go; rather, we want to use `let` to bind useful subexpressions to names. This can be done as follows:

```
#let successor = fun x -> x + 1 in
 successor(successor(successor 0));;
- : int = 3
```

Function bindings may use the more elegant sugaring:

```
#let successor x = x + 1 in
successor(successor(successor 0));;
- : int = 3
```

and can be made recursive just by adding the `rec` keyword:

```
#let rec fact n = if n = 0 then 1
                  else n * fact(n - 1) in
 fact 6;;
- : int = 720
```

By using `and`, we can make several binding simultaneously, and define mutually recursive functions. For example, here are two simple, though highly inefficient, functions to decide whether or not a natural number is odd or even:

[4] CAML does not store them simply as syntax trees, but compiles them into bytecode.

```
#let rec even n = if n = 0 then true else odd (n - 1)
     and odd n  = if n = 0 then false else even (n - 1);;
even : int -> bool = <fun>
odd : int -> bool = <fun>
#even 12;;
- : bool = true
#odd 14;;
- : bool = false
```

In fact, any bindings can be performed separately from the final application. ML remembers a set of variable bindings, and the user can add to that set interactively. Simply omit the `in` and terminate the phrase with a double semicolon:

```
#let successor = fun x -> x + 1;;
successor : int -> int = <fun>
```

After this declaration, any later phrase may use the function `successor`, e.g:

```
#successor 11;;
- : int = 12
```

Note that we are not making *assignments* to *variables*. Each binding is only done once when the system analyses the input; it cannot be repeated or modified. It can be overwritten by a new definition using the same name, but this is not assignment in the usual sense, since the sequence of events is only connected with the *compilation* process, not with the dynamics of program *execution*. Indeed, apart from the more interactive feedback from the system, we could equally replace all the double semicolons after the declarations by `in` and evaluate everything at once. On this view we can see that the overwriting of a declaration really corresponds to the definition of a new local variable that hides the outer one, according to the usual lambda calculus rules. For example:

```
#let x = 1;;
x : int = 1
#let y = 2;;
y : int = 2
#let x = 3;;
x : int = 3
#x + y;;
- : int = 5
```

is the same as:

```
#let x = 1 in
 let y = 2 in
 let x = 3 in
 x + y;;
- : int = 5
```

Note carefully that, also following lambda calculus, variable binding is *static*, i.e. the first binding of `x` is still used until an inner binding occurs, and any uses of it until that point are not affected by the inner binding. For example:

```
#let x = 1;;
x : int = 1
#let f w = w + x;;
f : int -> int = <fun>
#let x = 2;;
x : int = 2
#f 0;;
- : int = 1
```

The first version of LISP, however, used *dynamic* binding, where a rebinding of a variable propagated to earlier uses of the variable, so that the analogous sequence to the above would return 2. This was in fact originally regarded as a bug, but soon programmers started to appreciate its convenience. It meant that when some low-level function was modified, the change propagated automatically to all applications of it in higher level functions, without the need for recompilation. The feature survived for a long time in many LISP dialects, but eventually the view that static binding is better prevailed. In Common LISP, static binding is the default, but dynamic binding is available if desired via the keyword `special`.

5.7 Polymorphic functions

We can define polymorphic functions, like the identity operator:

```
#let I = fun x -> x;;
I : 'a -> 'a = <fun>
```

ML prints type variables as `'a`, `'b` etc. These are supposed to be ASCII representations of α, β and so on. We can now use the polymorphic function several times with different types:

```
#I true;;
- : bool = true
#I 1;;
- : int = 1
#I I I I 12;;
- : int = 12
```

Each instance of `I` in the last expression has a different type, and intuitively corresponds to a different function. In fact, let's define all the combinators:

```
#let I x = x;;
I : 'a -> 'a = <fun>
#let K x y = x;;
K : 'a -> 'b -> 'a = <fun>
#let S f g x = (f x) (g x);;
S : ('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c = <fun>
```

Note that the system keeps track of the types for us, even though in the last case they were quite complicated. Now, recall that $I = S\ K\ K$; let us try this out in ML:[5]

```
#let I' = S K K;;
I' : '_a -> '_a = <fun>
```

It has the right type,[6] and it may easily be checked in all concrete cases, e.g:

```
#I' 3 = 3;;
- : bool = true
```

In the above examples of polymorphic functions, the system very quickly infers a most general type for each expression, and the type it infers is simple. This usually happens in practice, but there are pathological cases, e.g. the following example due to Mairson (1990). The type of this expression takes about 10 seconds to calculate, and occupies over 4000 lines on an 80-column terminal.

```
let pair x y = fun z -> z x y in
let x1 = fun y -> pair y y in
let x2 = fun y -> x1(x1 y) in
let x3 = fun y -> x2(x2 y) in
let x4 = fun y -> x3(x3 y) in
let x5 = fun y -> x4(x4 y) in
x5(fun z -> z);;
```

We have said that the ML programmer need never enter a type. This is true in the sense that ML will already allocate as general a type as possible to an expression. However it may sometimes be convenient to *restrict* the generality of a type. This cannot make code work that didn't work before, but it may serve as documentation regarding the intended purpose of the code; it is also possible to use shorter synonyms for complicated types. Type restriction can be achieved in ML by adding *type annotations* after some expression(s). These type annotations consist of a colon followed by a type. It usually doesn't matter exactly where these annotations are added, provided they enforce the appropriate constraints. For example, here are some alternative ways of constraining the identity function to type `int -> int`:

[5]We remarked that from the untyped point of view, $S\ K\ A = I$ for any A. However, the reader may try, for example, $S\ K\ S$ and see that the principal type is less general than expected.

[6]Ignore the underscores for now. This is connected with the typing of imperative features, and we will discuss it later.

```
#let I (x:int) = x;;
I : int -> int = <fun>
#let I x = (x:int);;
I : int -> int = <fun>
#let (I:int->int) = fun x -> x;;
I : int -> int = <fun>
#let I = fun (x:int) -> x;;
I : int -> int = <fun>
#let I = ((fun x -> x):int->int);;
I : int -> int = <fun>
```

5.8 Equality of functions

Instead of comparing the actions of I and I' on particular arguments like 3, it would seem that we can settle the matter definitively by comparing the functions themselves. However this doesn't work:

```
#I' = I;;
Uncaught exception: Invalid_argument "equal: functional value"
```

It is in general *forbidden* to compare functions for equality, though a few special instances, where the functions are obviously the same, yield `true`:

```
#let f x = x + 1;;
f : int -> int = <fun>
#let g x = x + 1;;
g : int -> int = <fun>
#f = f;;
- : bool = true
#f = g;;
Uncaught exception: Invalid_argument "equal: functional value"
#let h = g;;
h : int -> int = <fun>
#h = f;;
Uncaught exception: Invalid_argument "equal: functional value"
#h = g;;
- : bool = true
```

Why these restrictions? Aren't functions supposed to be first-class objects in ML? Yes, but unfortunately, (extensional) function equality is not computable. This follows from a number of classic theorems in recursion theory, such as the *unsolvability of the halting problem* and *Rice's theorem*.[7] Let us give a concrete illustration of why this might be so. It is still an open problem whether the

[7] You will see these results proved in the Computation Theory course. Rice's theorem is an extremely strong undecidability result which asserts that *any* nontrivial property of the function corresponding to a program is uncomputable from its text. An excellent computation theory textbook is Davis, Sigal, and Weyuker (1994).

following function terminates for all arguments, the assertion that it does being known as the *Collatz conjecture*:[8]

```
#let rec collatz n =
   if n <= 1 then 0
   else if even(n) then collatz(n / 2)
   else collatz(3 * n + 1);;
collatz : int -> int = <fun>
```

What is clear, though, is that if it does halt it returns 0. Now consider the following trivial function:

```
#let f (x:int) = 0;;
f : int -> int = <fun>
```

By deciding the equation `collatz = f`, the computer would settle the Collatz conjecture. It is easy to concoct other examples for open mathematical problems.

It is possible to trap out applications of the equality operator to functions and datatypes built up from them as part of typechecking, rather than at runtime. Types that do not involve functions in these ways are known as *equality types*, since it is always valid to test objects of such types for equality. On the negative side, this makes the type system much more complicated. However one might argue that static typechecking should be extended as far as feasibility allows.

Further reading

Many books on functional programming bear on the general issues we have discussed here, such as evaluation strategy. A good elementary introduction to CAML Light and functional programming is Mauny (1995). Paulson (1991) is another good textbook, though based on Standard ML.

Exercises

1. Suppose that a 'conditional' function defined by `ite(b,x,y) = if b then x else y` is the only function available that operates on arguments of type `bool`. Is there any way to write a factorial function?

2. Use the typing rules in the last chapter to write out a formal proof that the S combinator has the type indicated by the ML system.

[8] A good survey of this problem, and attempts to solve it, is given by Lagarias (1985). Strictly, we should use unlimited precision integers rather than machine arithmetic. We will see later how to do this.

3. Write a simple recursively defined function to perform exponentiation of integers, i.e. calculate x^n for $n \geq 0$. Write down two ML functions, the equality of which corresponds to the truth of Fermat's last theorem: there are no integers x, y, z and natural number $n > 2$ with $x^n + y^n = z^n$ except for the trivial case where $x = 0$ or $y = 0$.

Chapter 6

Further ML

In this chapter, we consolidate the previous examples by specifying the basic facilities of ML and the syntax of phrases more precisely, and then go on to treat some additional features such as recursive types. We might start by saying more about interaction with the system.

So far, we have just been typing phrases into ML's toplevel read-eval-print loop and observing the result. However this is not a good method for writing nontrivial programs. Typically, you should write the expressions and declarations in a file. To try things out as you go, they can be inserted in the ML window using 'cut and paste'. This operation can be performed using X-windows and similar systems, or in an editor like Emacs with multiple buffers. However, this becomes laborious and time-consuming for large programs. Instead, you can use ML's `include` function to read in the file directly. For example, if the file `myprog.ml` contains:

```
let pythag x y z =
  x * x + y * y = z * z;;

pythag 3 4 5;;

pythag 5 12 13;;

pythag 1 2 3;;
```

then the toplevel phrase `include "myprog.ml";;` results in:

```
#include "myprog.ml";;
pythag : int -> int -> int -> bool = <fun>
- : bool = true
- : bool = true
- : bool = false
- : unit = ()
```

That is, the ML system responds just as if the phrases had been entered at the top level. The final line is the result of evaluating the `include` expression itself.

In large programs, it is often helpful to include comments. In ML, these are written between the symbols `(*` and `*)`, e.g.

```
(* ---------------------------------------------------- *)
(* This function tests if (x,y,z) is a Pythagorean triple *)
(* ---------------------------------------------------- *)

let pythag x y z =
    x * x + y * y = z * z;;

(*comments*) pythag (*can*) 3 (*go*) 4 (*almost*) 5 (*anywhere*)
(* and (* can (* be (* nested *) quite *) arbitrarily *) *);;
```

6.1 Basic datatypes and operations

ML features several built-in primitive types. From these, composite types may be built using various type constructors. For the moment, we will only use the function space constructor `->` and the Cartesian product constructor `*`, but we will see in due course which others are provided, and how to define new types and type constructors. The primitive types that concern us now are:

- The type `unit`. This is a 1-element type, whose only element is written `()`. Obviously, something of type `unit` conveys no information, so it is commonly used as the return type of imperatively written 'functions' that perform a side-effect, such as `include` above. It is also a convenient argument where the only use of a function type is to delay evaluation.

- The type `bool`. This is a 2-element type of booleans (truth-values) whose elements are written `true` and `false`.

- The type `int`. This contains some finite subset of the positive and negative integers. Typically the permitted range is from -2^{30} (-1073741824) up to $2^{30}-1$ (1073741823).[1] The numerals are written in the usual way, optionally with a negation sign, e.g. `0`, `32`, `-25`.

- The type `string` contains strings (i.e. finite sequences) of characters. They are written and printed between double quotes, e.g. `"hello"`. In order to encode include special characters in strings, C-like escape sequences are used. For example, `\"` is the double quote itself, and `\n` is the newline character.

[1] This is even more limited than expected for machine arithmetic because a bit is stolen for the garbage collector. We will see later how to use an alternative type of integers with unlimited precision.

The above values like `()`, `false`, `7` and `"caml"` are all to be regarded as constants in the lambda calculus sense. There are other constants corresponding to *operations* on the basic types. Some of these may be written as infix operators, for the sake of familiarity. These have a notion of precedence so that expressions are grouped together as one would expect. For example, we write `x + y` rather than `+ x y` and `x < 2 * y + z` rather than `< x (+ (* 2 y) z)`. The logical operator `not` also has a special parsing status, in that the usual left-associativity rule is reversed for it: `not not p` means `not (not p)`. User-defined functions may be granted infix status via the `#infix` directive. For example, here is a definition of a function performing composition of functions:

```
#let successor x = x + 1;;
successor : int -> int = <fun>
#let o f g = fun x -> f(g x);;
o : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
#let add3 = o successor (o successor successor);;
add3 : int -> int = <fun>
#add3 0;;
- : int = 3
##infix "o";;
#let add3' = successor o successor o successor;;
add3' : int -> int = <fun>
#add3' 0;;
- : int = 3
```

It is not possible to specify the precedence of user-defined infixes, nor to make user-defined non-infix functions right-associative. Note that the implicit operation of 'function application' has a higher precedence than any binary operator, so `successor 1 * 2` parses as `(successor 1) * 2`. If it is desired to use a function with special status as an ordinary constant, simply precede it by `prefix`. For example:

```
#o successor successor;;
Toplevel input:
>o successor successor;;
>^
Syntax error.
#prefix o successor successor;;
- : int -> int = <fun>
#(prefix o) successor successor;;
- : int -> int = <fun>
```

With these questions of concrete syntax out of the way, let us present a systematic list of the operators on the basic types above. The unary operators are:

Operator	Type	Meaning
`-`	`int -> int`	Numeric negation
`not`	`bool -> bool`	Logical negation

and the binary operators, in approximately decreasing order of precedence, are:

Operator	Type	Meaning
`mod`	`int -> int -> int`	Modulus (remainder)
`*`	`int -> int -> int`	Multiplication
`/`	`int -> int -> int`	Truncating division
`+`	`int -> int -> int`	Addition
`-`	`int -> int -> int`	Subtraction
`^`	`string -> string -> string`	String concatenation
`=`	`'a -> 'a -> bool`	Equality
`<>`	`'a -> 'a -> bool`	Inequality
`<`	`'a -> 'a -> bool`	Less than
`<=`	`'a -> 'a -> bool`	Less than or equal
`>`	`'a -> 'a -> bool`	Greater than
`>=`	`'a -> 'a -> bool`	Greater than or equal
`&`	`bool -> bool -> bool`	Boolean 'and'
`or`	`bool -> bool -> bool`	Boolean 'or'

For example, `x > 0 & x < 1` is parsed as `& (> x 0) (< x 1)`. Note that all the comparisons, not just the equality relation, are polymorphic. They not only order integers in the expected way, and strings alphabetically, but all other primitive types and composite types in a fairly natural way. Once again, however, they are not in general allowed to be used on functions.

The two boolean operations `&` and `or` have their own special evaluation strategy, like the conditional expression. In fact, they can be regarded as synonyms for conditional expressions:

$$\begin{aligned} p \mathbin{\&} q &\triangleq \text{if } p \text{ then } q \text{ else false} \\ p \text{ or } q &\triangleq \text{if } p \text{ then true else } q \end{aligned}$$

Thus, the 'and' operation evaluates its first argument, and only if it is true, evaluates its second. Conversely, the 'or' operation evaluates its first argument, and only if it is false evaluates its second.

6.2 Syntax of ML phrases

Expressions in ML can be built up from constants and variables; any identifier that is not currently bound is treated as a variable. Declarations bind names to values of expressions, and declarations can occur locally inside expressions. Thus, the syntax classes of expressions and declarations are mutually recursive. We can represent this by the following BNF grammar.[2]

[2] We neglect many constructs that we won't be concerned with. A few will be introduced later. See the CAML manual for full details.

$$
\begin{array}{rcl}
expression & ::= & variable \\
& | & constant \\
& | & expression\ expression \\
& | & expression\ infix\ expression \\
& | & \text{not}\ expression \\
& | & \text{if}\ expression\ \text{then}\ expression\ \text{else}\ expression \\
& | & \text{fun}\ pattern\ \texttt{->}\ expression \\
& | & (expression) \\
& | & declaration\ \text{in}\ expression \\
declaration & ::= & \text{let}\ let\_bindings \\
& | & \text{let rec}\ let\_bindings \\
let\_bindings & ::= & let\_binding \\
& | & let\_binding\ \text{and}\ let\_bindings \\
let\_binding & ::= & pattern = expression \\
pattern & ::= & variables \\
variables & ::= & variable \\
& | & variable\ variables
\end{array}
$$

The syntax class *pattern* will be expanded and explained more thoroughly later on. For the moment, all the cases we are concerned with are either just *variable* or *variable variable* $\cdots$ *variable*. In the first case we simply bind an expression to a name, while the second uses the special syntactic sugar for function declarations, where the arguments are written after the function name to the left of the equals sign. For example, the following is a valid declaration of a function `add4`, which can be used to add 4 to its argument:

```
#let add4 x =
  let y = successor x in
  let z = let w = successor y in
          successor w in
  successor z;;
add4 : int -> int = <fun>
#add4 1;;
- : int = 5
```

It is instructive to unravel this declaration according to the above grammar. A toplevel phrase, terminated by two successive semicolons, may be either an expression or a declaration.

6.3 Further examples

It is easy to define by recursion a function that takes a positive integer n and a function f and returns f^n, i.e. $f \circ \cdots \circ f$ (n times):

```
#let rec funpow n f x =
   if n = 0 then x
   else funpow (n - 1) f (f x);;
funpow : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```

In fact, a little thought will show that the function `funpow` takes a machine integer n and returns the Church numeral corresponding to n. Since functions aren't printed, we can't actually look at the lambda expression representing a Church numeral:

```
#funpow 6;;
- : ('_a -> '_a) -> '_a -> '_a = <fun>
```

However it is straightforward to define an inverse function to `funpow` that takes a Church numeral back to a machine integer:

```
#let defrock n = n (fun x -> x + 1) 0;;
defrock : ((int -> int) -> int -> 'a) -> 'a = <fun>
#defrock(funpow 32);;
- : int = 32
```

We can try out some of the arithmetic operations on Church numerals:

```
#let add m n f x = m f (n f x);;
add : ('a -> 'b -> 'c) -> ('a -> 'd -> 'b) -> 'a -> 'd -> 'c = <fun>
#let mul m n f x = m (n f) x;;
mul : ('a -> 'b -> 'c) -> ('d -> 'a) -> 'd -> 'b -> 'c = <fun>
#let exp m n f x = n m f x;;
exp : 'a -> ('a -> 'b -> 'c -> 'd) -> 'b -> 'c -> 'd = <fun>
#let test bop x y = defrock (bop (funpow x) (funpow y));;
test :
 ((('a -> 'a) -> 'a -> 'a) ->
  (('b -> 'b) -> 'b -> 'b) -> (int -> int) -> int -> 'c) ->
  int -> int -> 'c = <fun>
#test add 2 10;;
- : int = 12
#test mul 2 10;;
- : int = 20
#test exp 2 10;;
- : int = 1024
```

The above is not a very efficient way of performing arithmetic operations. ML does not have a function for exponentiation, but it is easy to define one by recursion:

```
 #let rec exp x n =
   if n = 0 then 1
   else x * exp x (n - 1);;
 exp : int -> int -> int = <fun>
```

However this performs n multiplications to calculate x^n. A more efficient way is to exploit the facts that $x^{2n} = (x^n)^2$ and $x^{2n+1} = x(x^n)^2$ as follows:

```
#let square x = x * x;;
square : int -> int = <fun>
#let rec exp x n =
  if n = 0 then 1
  else if n mod 2 = 0 then square(exp x (n / 2))
  else x * square(exp x (n / 2));;
exp : int -> int -> int = <fun>
#infix "exp";;
#2 exp 10;;
- : int = 1024
#2 exp 20;;
- : int = 1048576
```

Another classic operation on natural numbers is to find their greatest common divisor (highest common factor) using Euclid's algorithm:

```
#let rec gcd x y =
     if y = 0 then x else gcd y (x mod y);;
gcd : int -> int -> int = <fun>
#gcd 100 52;;
- : int = 4
#gcd 7 159;;
- : int = 1
#gcd 24 60;;
- : int = 12
```

We have used a notional recursion operator `Rec` to explain recursive definitions. We can actually define such a thing, and use it:

```
#let rec Rec f = f(fun x -> Rec f x);;
Rec : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
#let fact = Rec (fun f n -> if n = 0 then 1 else n * f(n - 1));;
fact : int -> int = <fun>
#fact 3;;
it : int = 6
```

Note, however, that the lambda was essential, otherwise the expression `Rec f` goes into an infinite recursion before it is even applied to its argument:

```
#let rec Rec f = f(Rec f);;
Rec : ('a -> 'a) -> 'a = <fun>
#let fact = Rec (fun f n -> if n = 0 then 1 else n * f(n - 1));;
Uncaught exception: Out_of_memory
```

6.4 Type definitions

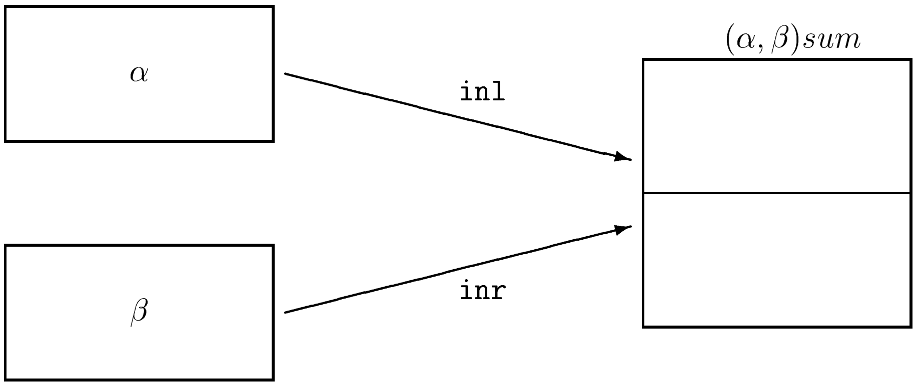
We have promised that ML has facilities for declaring new type constructors, so that composite types can be built up out of existing ones. In fact, ML goes further and allows a composite type to be built up not only out of preexisting types but also from the composite type itself. Such types, naturally enough, are said to be *recursive*. They are declared using the `type` keyword followed by an equation indicating how the new type is built up from existing ones and itself. We will illustrate this by a few examples. The first one is the definition of a *sum* type, intended to correspond to the disjoint union of two existing types.

```
#type ('a,'b)sum = inl of 'a | inr of 'b;;
Type sum defined.
```

Roughly, an object of type `('a,'b)sum` is *either* something of type `'a` *or* something of type `'b`. More formally, however, all these things have different types. The type declaration also declares the so-called *constructors* `inl` and `inr`. These are functions that take objects of the component types and inject them into the new type. Indeed, we can see their types in the ML system and apply them to objects:

```
#inl;;
- : 'a -> ('a, 'b) sum = <fun>
#inr;;
- : 'a -> ('b, 'a) sum = <fun>
#inl 5;;
- : (int, 'a) sum = inl 5
#inr false;;
- : ('a, bool) sum = inr false
```

We can visualize the situation via the following diagram. Given two existing types α and β, the type $(\alpha, \beta)sum$ is composed precisely of separate copies of α and β, and the two constructors map onto the respective copies:



This is similar to a `union` in C, but in ML the copies of the component types are kept apart and one always knows which of these an element of the union belongs to. By contrast, in C the component types are overlapped, and the programmer is responsible for this book-keeping.

6.4.1 Pattern matching

The constructors in such a definition have three very important properties:

- They are exhaustive, i.e. every element of the new type is obtainable either by `inl x` for some `x` or `inr y` for some `y`. That is, the new type contains nothing besides copies of the component types.
- They are injective, i.e. an equality test `inl x = inl y` is true if and only if `x = y`, and similarly for `inr`. That is, the new type contains a faithful copy of each component type without identifying any elements.
- They are distinct, i.e. their ranges are disjoint. More concretely this means in the above example that `inl x = inr y` is false whatever `x` and `y` might be. That is, the copy of each component type is kept apart in the new type.

The second and third properties of constructors justify our using *pattern matching*. This is done by using more general *varstructs* as the arguments in a lambda, e.g.

```
#fun (inl n) -> n > 6
   | (inr b) -> b;;
- : (int, bool) sum -> bool = <fun>
```

This function has the property, naturally enough, that when applied to `inl n` it returns `n > 6` and when applied to `inr b` it returns `b`. It is precisely because of the second and third properties of the constructors that we know this does give a welldefined function. Because the constructors are injective, we can uniquely recover `n` from `inl n` and `b` from `inr b`. Because the constructors are distinct, we know that the two clauses cannot be mutually inconsistent, since no value can correspond to both patterns.

In addition, because the constructors are exhaustive, we know that each value will fall under one pattern or the other, so the function is defined everywhere. Actually, it is permissible to relax this last property by omitting certain patterns, though the ML system then issues a warning:

```
#fun (inr b) -> b;;
Toplevel input:
>fun (inr b) -> b;;
>^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
- : ('a, 'b) sum -> 'b = <fun>
```

If this function is applied to something of the form `inl x`, then it will not work:

```
#let f = fun (inr b) -> b;;
Toplevel input:
>let f = fun (inr b) -> b;;
>        ^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
f : ('a, 'b) sum -> 'b = <fun>
#f (inl 3);;
Uncaught exception: Match_failure ("", 452, 468)
```

Though booleans are built into ML, they are effectively defined by a rather trivial instance of a recursive type, often called an *enumerated type*, where the constructors take no arguments:

```
#type bool = false | true;;
```

Indeed, it is perfectly permissible to define things by matching over the truth values. The following two phrases are completely equivalent:

```
#if 4 < 3 then 1 else 0;;
- : int = 0
#(fun true -> 1 | false -> 0) (4 < 3);;
- : int = 0
```

Pattern matching is, however, not limited to casewise definitions over elements of recursive types, though it is particularly convenient there. For example, we can define a function that tells us whether an integer is zero as follows:

```
#fun 0 -> true | n -> false;;
- : int -> bool = <fun>
#(fun 0 -> true | n -> false) 0;;
- : bool = true
#(fun 0 -> true | n -> false) 1;;
- : bool = false
```

In this case we no longer have mutual exclusivity of patterns, since `0` matches either pattern. The patterns are examined in order, one by one, and the first matching one is used. Note carefully that unless the matches are mutually exclusive, there is no guarantee that each clause holds as a mathematical equation. For example in the above, the function does not return `false` for any `n`, so the second clause is not universally valid.

Note that only *constructors* may be used in the above special way as components of patterns. Ordinary constants will be treated as new variables bound inside the pattern. For example, consider the following:

```
#let true_1 = true;;
true_1 : bool = true
#let false_1 = false;;
false_1 : bool = false
#(fun true_1 -> 1 | false_1 -> 0) (4 < 3);;
Toplevel input:
>(fun true_1 -> 1 | false_1 -> 0) (4 < 3);;
>                   ^^^^^^^
Warning: this matching case is unused.
- : int = 1
```

In general, the unit element `()`, the truth values, the integer numerals, the string constants and the pairing operation (infix comma) have constructor status, as well as other constructors from predefined recursive types. When they occur in a pattern the target value must correspond. All other identifiers match any expression and in the process become bound.

As well as the varstructs in lambda expressions, there are other ways of performing pattern matching. Instead of creating a function via pattern matching and applying it to an expression, one can perform pattern-matching over the expression directly using the following construction:

$$\text{match } \mathit{expression} \text{ with } \mathit{pattern}_1\texttt{->}E_1 \mid \cdots \mid \mathit{pattern}_n\texttt{->}E_n$$

The simplest alternative of all is to use

$$\text{let } \mathit{pattern} = \mathit{expression}$$

but in this case only a single pattern is allowed.

6.4.2 Recursive types

The previous examples have all been recursive only vacuously, in that we have not defined a type in terms of itself. For a more interesting example, we will declare a type of lists (finite ordered sequences) of elements of type `'a`.

```
#type ('a)list = Nil | Cons of 'a * ('a)list;;
Type list defined.
```

Let us examine the types of the constructors:

```
#Nil;;
- : 'a list = Nil
#Cons;;
- : 'a * 'a list -> 'a list = <fun>
```

The constructor `Nil`, which takes no arguments, simply creates some object of type `('a)list` which is to be thought of as the empty list. The other constructor `Cons` takes an element of type `'a` and an element of the new type `('a)list` and gives another, which we think of as arising from the old list by adding one element to the front of it. For example, we can consider the following:

```
#Nil;;
- : 'a list = Nil
#Cons(1,Nil);;
- : int list = Cons (1, Nil)
#Cons(1,Cons(2,Nil));;
- : int list = Cons (1, Cons (2, Nil))
#Cons(1,Cons(2,Cons(3,Nil)));;
- : int list = Cons (1, Cons (2, Cons (3, Nil)))
```

Because the constructors are distinct and injective, it is easy to see that all these values, which we think of as lists $[]$, $[1]$, $[1;2]$ and $[1;2;3]$, are distinct. Indeed, purely from these properties of the constructors, it follows that arbitrarily long lists of elements may be encoded in the new type. Actually, ML already has a type `list` just like this one defined. The only difference is syntactic: the empty list is written `[]` and the recursive constructor `::`, has infix status. Thus, the above lists are actually written:

```
#[];;
- : 'a list = []
#1::[];;
- : int list = [1]
#1::2::[];;
- : int list = [1; 2]
#1::2::3::[];;
- : int list = [1; 2; 3]
```

The lists are printed in an even more natural notation, and this is also allowed for input. Nevertheless, when the exact expression in terms of constructors is needed, it must be remembered that this is only a surface syntax. For example, we can define functions to take the head and tail of a list, using pattern matching.

```
#let hd (h::t) = h;;
Toplevel input:
>let hd (h::t) = h;;
>    ^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
hd : 'a list -> 'a = <fun>
#let tl (h::t) = t;;
Toplevel input:
>let tl (h::t) = t;;
>    ^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
tl : 'a list -> 'a list = <fun>
```

The compiler warns us that these both fail when applied to the empty list, since there is no pattern to cover it (remember that the constructors are distinct). Let us see them in action:

```
#hd [1;2;3];;
- : int = 1
#tl [1;2;3];;
- : int list = [2; 3]
#hd [];;
Uncaught exception: Match_failure
```

Note that the following is not a correct definition of `hd`. In fact, it constrains the input list to have exactly two elements for matching to succeed, as can be seen by thinking of the version in terms of the constructors:

```
#let hd [x;y] = x;;
Toplevel input:
>let hd [x;y] = x;;
>    ^^^^^^^^^^^^
Warning: this matching is not exhaustive.
hd : 'a list -> 'a = <fun>
#hd [5;6];;
- : int = 5
#hd [5;6;7];;
Uncaught exception: Match_failure
```

Pattern matching can be combined with recursion. For example, here is a function to return the length of a list:

```
#let rec length =
   fun [] -> 0
     | (h::t) -> 1 + length t;;
length : 'a list -> int = <fun>
#length [];;
- : int = 0
#length [5;3;1];;
- : int = 3
```

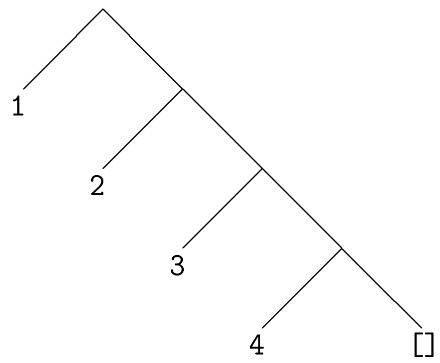
Alternatively, this can be written in terms of our earlier 'destructor' functions `hd` and `tl`:

```
#let rec length l =
   if l = [] then 0
   else 1 + length(tl l);;
```

This latter style of function definition is more usual in many languages, notably LISP, but the direct use of pattern matching is often more elegant.

6.4.3 Tree structures

It is often helpful to visualize the elements of recursive types as tree structures, with the recursive constructors at the branch nodes and the other datatypes at the leaves. The recursiveness merely says that plugging subtrees together gives another tree. In the case of lists the 'trees' are all rather spindly and one-sided, with the list `[1;2;3;4]` being represented as:



It is not difficult to define recursive types which allow more balanced trees, e.g.

```
#type ('a)btree = Leaf of 'a
                | Branch of ('a)btree * ('a)btree;;
```

In general, there can be several different recursive constructors, each with a different number of descendants. This gives a very natural way of representing the *syntax trees* of programming (and other formal) languages. For example, here is a type to represent arithmetical expressions built up from integers by addition and multiplication:

```
#type expression = Integer of int
                 | Sum of expression * expression
                 | Product of expression * expression;;
```

and here is a recursive function to evaluate such expressions:

```
#let rec eval =
   fun (Integer i) -> i
     | (Sum(e1,e2)) -> eval e1 + eval e2
     | (Product(e1,e2)) -> eval e1 * eval e2;;
eval : expression -> int = <fun>
#eval (Product(Sum(Integer 1,Integer 2),Integer 5));;
- : int = 15
```

Such abstract syntax trees are a useful representation which allows all sorts of manipulations. Often the first step programming language compilers and related tools take is to translate the input text into an 'abstract syntax tree' according to the parsing rules. Note that conventions such as precedences and bracketings are not needed once we have reached the level of abstract syntax; the tree structure makes these explicit. We will use such techniques to write ML versions of the formal rules of lambda calculus that we were considering earlier. First, the following type is used to represent lambda terms:

```
#type term = Var of string
           | Const of string
           | Comb of term * term
           | Abs of string * term;;
Type term defined.
```

Note that we make `Abs` take a variable name and a term rather than two terms in order to prohibit the formation of terms that aren't valid. For example, we represent the term $\lambda x\ y.\ y\ (x\ x\ y)$ by:

```
Abs("x",Abs("y",Comb(Var "y",Comb(Comb(Var "x",Var "x"),Var "y"))))
```

Here is the recursive definition of a function `free_in` to decide whether a variable is free in a term:

```
#let rec free_in x =
    fun (Var v) -> x = v
      | (Const c) -> false
      | (Comb(s,t)) -> free_in x s or free_in x t
      | (Abs(v,t)) -> not x = v & free_in x t;;
free_in : string -> term -> bool = <fun>
#free_in "x" (Comb(Var "f",Var "x"));;
- : bool = true
#free_in "x" (Abs("x",Comb(Var "x",Var "y")));;
- : bool = false
```

Similarly, we can define substitution by recursion. First, though, we need a function to rename variables to avoid name clashes. We want the ability to convert an existing variable name to a new one that is not free in some given expression. The renaming is done by adding prime characters.

```
#let rec variant x t =
   if free_in x t then variant (x^"'") t
   else x;;
variant : string -> term -> string = <fun>
#variant "x" (Comb(Var "f",Var "x"));;
- : string = "x'"
#variant "x" (Abs("x",Comb(Var "x",Var "y")));;
- : string = "x"
#variant "x" (Comb(Var "f",Comb(Var "x",Var "x'")));;
- : string = "x''"
```

Now we can define substitution just as we did in abstract terms:

```
#let rec subst u (t,x) =
  match u with
    Var y -> if x = y then t else Var y
  | Const c -> Const c
  | Comb(s1,s2) -> Comb(subst s1 (t,x),subst s2 (t,x))
  | Abs(y,s) -> if x = y then Abs(y,s)
                else if free_in x s & free_in y t then
                  let z = variant y (Comb(s,t)) in
                  Abs(z,subst (subst s (Var y,z)) (t,x))
                else Abs(y,subst s (t,x));;
subst : term -> term * string -> term = <fun>
```

Note the very close parallels between the standard mathematical presentation of lambda calculus, as we presented earlier, and the ML version here. All we really need to make the analogy complete is a means of reading and writing the terms in some more readable form, rather than by explicitly invoking the constructors. We will return to this issue later, and see how to add these features.

6.4.4 The subtlety of recursive types

A recursive type may contain nested instances of other type constructors, including the function space constructor. For example, consider the following:

```
#type ('a)embedding = K of ('a)embedding->'a;;
Type embedding defined.
```

If we stop to think about the underlying semantics, this looks disquieting. Consider for example the special case when `'a` is `bool`. We then have an injective function `K:((bool)embedding->bool)->(bool)embedding`. This directly contradicts Cantor's theorem that the set of all subsets of X cannot be injected into X.[3] Hence we need to be more careful with the semantics of types. In fact $\alpha \to \beta$ cannot be interpreted as the full function space, or recursive type constructions like the above are inconsistent. However, since all functions we can actually create are computable, it is reasonable to restrict ourselves to computable functions only. With that restriction, a consistent semantics is possible, although the details are complicated.

The above definition also has interesting consequences for the type system. For example, we can now define the Y combinator, by using K as a kind of type cast. Note that if the Ks are deleted, this is essentially the usual definition of

[3]Proof: consider $C = \{i(s) \mid s \in \wp(X) \text{ and } i(s) \notin s\}$. If $i : \wp(X) \to X$ is injective, we have $i(C) \in C \Leftrightarrow i(C) \notin C$, a contradiction. This is similar to the Russell paradox, and in fact probably inspired it. The analogy is even closer if we consider the equivalent form that there is no surjection $j : X \to \wp(X)$, and prove it by considering $\{s \mid s \notin j(s)\}$.

the Y combinator in untyped lambda calculus. The use of `let` is only used for the sake of efficiency, but we *do* need the η-redex involving `z` in order to prevent looping under ML's evaluation strategy.

```
#let Y h =
  let g (K x) z = h (x (K x)) z in
  g (K g);;
Y : (('a -> 'b) -> 'a -> 'b) -> 'a -> 'b = <fun>
#let fact = Y (fun f n -> if n = 0 then 1 else n * f(n - 1));;
fact : int -> int = <fun>
#fact 6;;
- : int = 720
```

Thus, recursive types are a powerful addition to the language, and allow us to back off the change of making the recursion operator a primitive.

Exercises

1. What goes wrong in our definition of `subst` if we replace

   `Var y -> if x = y then t else Var y`

   by these separate patterns?

   `Var x -> t | Var y -> Var y`

2. It would be possible, though inefficient, to define natural numbers using the following recursive type:

   ```
   #type num = Zero | Suc of num;;
   ```

   Define functions to perform arithmetic on natural numbers in this form.

3. Use the type definition for the syntax of lambda terms that we gave above. Write a function to convert an arbitrary lambda term into an equivalent in terms of the S, K and I combinators, based on the technique we gave earlier. Represent the combinators by `Const "S"` etc.

4. Extend the syntax of lambda terms to a incorporate Church-style typing. Write functions that check the typing conditions before allowing terms to be formed, but otherwise map down to the primitive constructors.

5. Consider a type of binary trees with strings at the nodes defined by:

   ```
   #type stree = Leaf | Branch of stree * string * stree;;
   ```

Write a function `strings` to return a list of the strings occurring in the tree in left-to-right order, i.e. at each node it appends the list resulting from the left branch, the singleton list at the current node, and the list occurring at the right branch, in that order. Write a function that takes a tree where the corresponding list is alphabetically sorted and a new string, and returns a tree with the new string inserted in the appropriate place.

6. (*) Refine the above functions so that the tree remains almost balanced at each stage, i.e. the maximum depths occurring at the left and right branches differ by at most one.[4]

7. (*) Can you characterize the types of expressions that can be written down in simply typed lambda calculus, without a recursion operator?[5] What is the type of the following recursive function?

```
let rec f i = (i o f) (i o i);;
```

Show how it can be used to write down ML expressions with completely polymorphic type α. Can you write a terminating expression with this property? What about an arbitrary function type $\alpha \to \beta$?

[4]See Adel'son-Vel'skii and Landis (1962) and various algorithm books for more details of these 'AVL trees' and similar methods for balanced trees.

[5]The answer lies in the 'Curry-Howard isomorphism' — see Girard, Lafont, and Taylor (1989) for example.

Chapter 7

Proving programs correct

As programmers know through bitter personal experience, it can be very difficult to write a program that is *correct*, i.e. performs its intended function. Most large programs have bugs. The consequences of these bugs can vary widely. Some bugs are harmless, some merely irritating. Some are deadly. For example the programs inside heart pacemakers, aircraft autopilots, car engine management systems and antilock braking systems, radiation therapy machines and nuclear reactor controllers are *safety critical*. An error in one of them can lead directly to loss of life, possibly on a large scale. As computers become ever more pervasive in society, the number of ways in which program bugs can kill people increases.[1]

How can we make sure a program is correct? One very useful method is simply to test it in a variety of situations. One tries the program out on inputs designed to exercise many different parts of it and reveal possible bugs. But usually there are too many possible situations to try them all exhaustively, so there may still be bugs lying undetected. As repeatedly emphasized by various writers, notably Dijkstra, program testing can be very useful for demonstrating the presence of bugs, but it is only in a few unusual cases where it can demonstrate their absence.

Instead of testing a program, one can try to *verify* it mathematically. A program in a sufficiently precisely defined programming language has a definite mathematical meaning. Similarly, the requirements on a program can be expressed using mathematics and logic as a precise *specification*. One can then try to perform a mathematical *proof* that the program meets its specification.

One can contrast testing and verification of programs by considering the following simple mathematical assertion:

$$\Sigma_{n=0}^{N} n = \frac{N(N+1)}{2}$$

It is claimed that this holds for any N. We can certainly test it for any number of particular values of N. This may lead to our intuitive confidence in the formula, such that we can hardly believe it could ever fail. However in

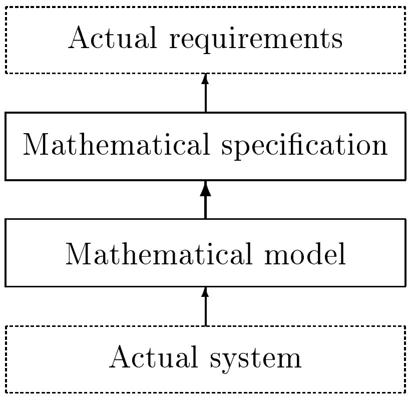
[1]Similar remarks apply to hardware, but we will focus on software.

general a preponderance of numerical evidence can be deceptive; there are well known cases in number theory where things turned out to be false against the weight of many particular cases.[2] A more reliable procedure is to *prove* the above formula mathematically. This can easily be done using induction on N. In the same way, we hope to replace testing a program on a range of inputs by some completely general proof that it always works correctly. It is important, however, to appreciate two limitations of program proofs.

- One has no guarantee that the execution of the program on the computer corresponds exactly to the abstract mathematical model. One must be on the lookout for discrepancies, caused perhaps by bugs in compilers and operating systems, or physical defects in the underlying technology. This is, of course, a general problem in applications of mathematics to science. For example, assuming that arithmetic operations correspond exactly to their mathematical counterparts is a natural simplifying assumption, just as one might neglect air resistance in the analysis of a simple dynamical system. But both can be invalid in certain cases.

- The program is to be verified by proof against a mathematical specification. It is possible that this specification does not capture accurately what is actually wanted in the real world. In fact, it can be remarkably difficult to arrive at a precise mathematical version of informal requirements. There is evidence that many of the serious problems in computer systems are caused not by coding errors in the usual sense, but by a mistaken impression of what is actually wanted. Just formalizing the requirements is often valuable in itself.

We can represent this situation by the following diagram:



We are trying to establish a link between the bottom and top boxes, i.e. the actual system and the actual requirements in real life. To do this, we proceed by

[2]For example, Littlewood proved in 1914 that $\pi(n) - li(n)$ changes sign infinitely often, where $\pi(n)$ is the number of primes $\leq n$ and $li(n) = \int_0^n du/ln(u)$. This came as a surprise since not a single sign change had been found despite extensive testing up to 10^{10}.

producing a mathematical version of each. It is only the central link, between the mathematical model of the system and the mathematical version of the specification, that is mathematically precise. The other links remain informal. All we can do is try to keep them small by using a realistic model of the system and a high-level mathematical specification that is reasonably readable.

These reservations notwithstanding, program proving has considerable merit. Compared with testing, it establishes correctness once and for all, possibly for a whole class of programs at once (e.g. controlled by the setting of some parameter). Moreover because it is a more principled analytical approach, the process of proving a program (or failing to do so) can lead to a deeper appreciation of the program and of the task in hand.

7.1 Functional programs as mathematical objects

In the introduction, we remarked that (pure) functional programs really correspond directly to functions in the mathematical sense. For this reason, it is often suggested that they are more amenable to formal proof than imperative programs. Even if this is true, and many would dispute it, it is worth realizing that the gap between this mathematical abstraction and the final execution in hardware is greater than for typical imperative languages. In particular, there can be substantial storage requirements hidden in the eventual realization on the computer. So one may just be getting easier proofs because they prove less about what really matters. We won't settle this argument here, but we want to show that reasoning about simple functional programs *is* often fairly straightforward.

The example at the end of the previous chapter showed that a naive association of function spaces of ML and function spaces of mathematics is not quite accurate. However, if we stay away from the higher reaches of recursive types and are not concerned with the space of *all* functions, only with some particular ones, we can safely identify the objects of ML with those of abstract mathematical realms.[3]

We intend to model functional programs as mathematical functions. Given the equations — typically recursive — that define an ML function, we want to interpret them as 'axioms' about the corresponding mathematical objects. For example, we would assume from the definition of the factorial function:

```
#let rec fact = fun 0 -> 1
                  | n -> n * fact(n - 1);;
```

that $\mathrm{fact}(0) = 1$ and that for all $n \neq 0$ we have $\mathrm{fact}(n) = n * \mathrm{fact}(n-1)$. This is right, but we need to tread carefully because of the question of termination. For

[3] We will neglect arithmetic overflow whenever we use arithmetic. There is a facility in CAML for arbitrary precision arithmetic, at the cost of, in principle, unlimited storage requirements.

negative n, the program fails to terminate, so the second equation is true only in the vacuous sense that both sides are 'undefined'. It is much easier to reason about functions when we know they are total, so sometimes one uses a separate argument to establish termination. In the examples that follow, this is done in parallel with the correctness proof, i.e. we prove together that for each argument, the function terminates and the result satisfies our specification.

As for the actual proof that the function obeys the specification, this can be a mathematical proof in the most general sense. However it often happens that when functions are defined by recursion, properties of them, dually, can be proved by induction; this includes termination. Moreover, the exact form of induction (arithmetical, structural, wellfounded) usually corresponds to the mode of recursion used in the definition. This should become clearer when we have seen a few examples.

7.2 Exponentiation

Recall our simple definition of exponentiation:

```
#let rec exp x n =
  if n = 0 then 1
  else x * exp x (n - 1);;
```

We will prove the following property of `exp`:

Theorem 7.1 *For all $n \geq 0$ and x, $exp\ x\ n$ is defined and $exp\ x\ n = x^n$.*
Proof: *The ML function* `exp` *was defined by primitive, step-by-step, recursion. It is therefore no surprise that our proof proceeds by ordinary, step-by-step induction. We prove that it holds for $n = 0$ and then that if it holds for any $n \geq 0$ it also holds for $n + 1$.*

1. *If $n = 0$, then by definition $exp\ x\ n = 1$. By definition, for any integer x, we have $x^0 = 1$, so the desired fact is established. Note that we assume $0^0 = 1$, an example of how one must state the specification precisely — what does x^n mean in general? — and how it might surprise some people.*

2. *Suppose that for $n \geq 0$ we have $exp\ x\ n = x^n$. Since $n \geq 0$, we have $n + 1 \neq 0$ and so by definition $exp\ x\ (n + 1) = x * exp\ x\ ((n + 1) - 1)$. Therefore:*

$$\begin{aligned} exp\ x\ (n+1) &= x * exp\ x\ ((n+1)-1) \\ &= x * exp\ x\ n \\ &= x * x^n \\ &= x^{n+1} \end{aligned}$$

Q.E.D.

7.3 Greatest common divisor

Recall our function to calculate the greatest common divisor $gcd(m,n)$ of two natural numbers m and n:

```
#let rec gcd x y =
     if y = 0 then x else gcd y (x mod y);;
```

In fact, we claim this works for any integers, not just positive ones. However, one needs to understand the precise definition of *gcd* in that case. Let us define the relation '$u|v$', or 'u divides v' for any two integers u and v to mean 'v is an integral multiple of u', i.e. there is some integer d with $v = du$. For example, $0|0$, $1|11$, $-2|4$ but $0 \not| 1$, $3 \not| 5$. We will say that *d is a greatest common divisor of x and y* precisely if:

- $d|x$ and $d|y$
- For any other integer d', if $d'|x$ and $d'|y$ then $d'|d$.

Note that we say $d'|d$ not $d' \leq d$. This belies the use of 'greatest', but it is in fact the standard definition in algebra books. Observe that any pair of numbers (except 0 and 0) has two gcds, since if d is a gcd of x and y, so is $-d$.

Our specification is now: *for any integers x and y, $d = gcd\ x\ y$ is a gcd of x and y.* This is another good example of how providing a precise specification, even for such a simple function, is harder than it looks. This specification also exemplifies the common property that it does not completely specify the answer, merely places certain constraints on it. If we defined:

```
#let rec ngcd x y = -(gcd x y);;
```

then the function `ngcd` would satisfy the specification just as well as `gcd`. Of course, we are free to tighten the specification if we wish, e.g. insist that if x and y are positive, so is the gcd returned.

The `gcd` function is not defined simply by primitive recursion. In fact, `gcd x y` is defined in terms of `gcd y (x mod y)` in the step case. Correspondingly, we do not use step-by-step induction, but wellfounded induction. We want a wellfounded relation that decreases over recursive calls; this will guarantee termination and act as a handle for proofs by wellfounded induction. In general, complicated relations on the arguments are required. But often it is easy to concoct a *measure* that maps the arguments to natural numbers such that the measure decreases over recursive calls. Such is the case here: the measure is $|y|$.

Theorem 7.2 *For any integers x and y, $gcd\ x\ y$ terminates in a gcd of x and y.*
Proof: *Take some arbitrary n, and suppose that for all x and y with $|y| < n$ we have that $gcd\ x\ y$ terminates with a gcd of x and y, and try to prove that the same holds for all x and y with $|y| = n$. This suffices for the main result, since any y will have* some *n with $|y| = n$. So, let us consider an x and y with $|y| = n$. There are two cases to consider, according to the case split used in the definition.*

- *Suppose $y = 0$. Then $gcd\ x\ y = x$ by definition. Now trivially $x|x$ and $x|0$, so it is a common divisor. Suppose d is another common divisor, i.e. $d|x$ and $d|0$. Then indeed we have $d|x$ immediately, so x must be a greatest common divisor.*

- *Suppose $y \neq 0$. We want to apply the inductive hypothesis to $gcd\ y\ (x\ mod\ y)$. We will write $r = x\ mod\ y$ for short. The basic property of the* `mod` *function is that, since $y \neq 0$, for some integer q we have $x = qy + r$ and $|r| < |y|$. Since $|r| < |y|$ the inductive hypothesis tells us that $d = gcd\ y\ (x\ mod\ y)$ is a gcd of y and r. We just need to show that it is a gcd of x and y. It is certainly a common divisor, since if $d|y$ and $d|r$ we have $d|x$, as $x = qy+r$. Now suppose $d'|x$ and $d'|y$. By the same equation, we find that $d'|r$. Thus d' is a common divisor of y and r, but then by the inductive hypothesis, $d'|d$ as required.*

Q.E.D.

Note that the basic property of the modulus operation that we used needs careful checking against the specification of `mod` in the CAML manual. There are well known differences between languages, and between implementations of the same language, when moduli involving negative numbers are concerned. If ever one is in doubt over the validity of a necessary assumption, it can be made explicit in the theorem: 'if ... then ...'.

7.4 Appending

We will now have an example of a function defined over lists. The function `append` is intended, not surprisingly, to append, or join together, two lists. For example, if we append $[3; 2; 5]$ and $[6; 3; 1]$ we get $[3; 2; 5; 6; 3; 1]$.

```
#let rec append l1 l2 =
   match l1 with
     [] -> l2
   | (h::t) -> h::(append t l2);;
```

This is defined by primitive recursion over the type of lists. It is defined on the empty list, and then for $h :: t$ in terms of the value for t. Consequently, when proving theorems about it, it is natural to use the corresponding principle of structural induction for lists: if a property holds for the empty list, and whenever it holds for t it holds for any $h :: t$, then it holds for any list. However this is not obligatory, and if preferred, we could proceed by mathematical induction on the length of the list. We will aim at proving that the append operation is associative.

Theorem 7.3 *For any three lists l_1, l_2 and l_3 we have:*

$$append\, l_1\ (append\, l_2\ l_3) = append\, (append\, l_1\ l_2)\ l_3$$

Proof: *By structural induction on l_1, we prove this holds for any l_2 and l_3.*

- *If $l_1 = []$ then:*

$$\begin{aligned} append\, l_1\ (append\, l_2\ l_3) &= append\, []\ (append\, l_2\ l_3) \\ &= append\, l_2\ l_3 \\ &= append\, (append\, []\ l_2)\ l_3 \\ &= append\, (append\, l_1\ l_2)\ l_3 \end{aligned}$$

- *Now suppose $l_1 = h :: t$. We may assume that for any l_2 and l_3 we have:*

$$append\, t\ (append\, l_2\ l_3) = append\, (append\, t\ l_2)\ l_3$$

  Therefore:

$$\begin{aligned} append\, l_1\ (append\, l_2\ l_3) &= append\, (h :: t)\ (append\, l_2\ l_3) \\ &= h :: (append\, t\ (append\, l_2\ l_3)) \\ &= h :: (append\, (append\, t\ l_2)\ l_3) \\ &= append\, (h :: (append\, t\ l_2))\ l_3) \\ &= append\, (append\, (h :: t)\ l_2)\ l_3) \\ &= append\, (append\, l_1\ l_2)\ l_3) \end{aligned}$$

Q.E.D.

7.5 Reversing

It is not difficult to define a function to reverse a list:

```
#let rec rev =
   fun [] -> []
     | (h::t) -> append (rev t) [h];;
rev : 'a list -> 'a list = <fun>
#rev [1;2;3];;
- : int list = [3; 2; 1]
```

We will prove that `rev` is an involution, i.e. that

$$\mathrm{rev}(\mathrm{rev}\ l) = l$$

However, if we try to tackle this directly by list induction, we find that we need a couple of additional lemmas. We will prove these first.

Lemma 7.4 *For any list l we have $append\, l\, [] = l$.*
Proof: *Structural induction on l.*

- *If $l = []$ we have:*

$$\begin{aligned} append\, l\, [] &= append\, []\, [] \\ &= [] \\ &= l \end{aligned}$$

- *Now suppose $l = h :: t$ and we know that $append\, t\, [] = t$. We find:*

$$\begin{aligned} append\, l\, [] &= append\, (h :: t)\, [] \\ &= h :: (append\, t\, []) \\ &= h :: t \\ &= l \end{aligned}$$

Q.E.D.

Lemma 7.5 *For any lists l_1 and l_2 we have*

$$rev(append\, l_1\, l_2) = append\, (rev\, l_2)\, (rev\, l_1)$$

Proof: *Structural induction on l_1.*

- *If $l_1 = []$ we have:*

$$\begin{aligned} rev(append\, l_1\, l_2) &= rev(append\, []\, l_2) \\ &= rev\, l_2 \\ &= append\, (rev\, l_2)[] \\ &= append\, (rev\, l_2)\, (rev\, []) \end{aligned}$$

- *Now suppose* $l_1 = h :: t$ *and we know that*

$$rev(append\, t\; l_2) = append\,(rev\, l_2)\;(rev\, t)$$

then we find:

$$\begin{aligned}
rev(append\, l_1\; l_2) &= rev(append\,(h :: t)\; l_2) \\
&= rev(h :: (append\, t\; l_2)) \\
&= append\,(rev(append\, t\; l_2))\;[h] \\
&= append\,(append\,(rev\, l_2)\;(rev\, t))\;[h] \\
&= append\,(rev\, l_2)\;(append\,(rev\, t)\;[h]) \\
&= append\,(rev\, l_2)\;(rev\,(h :: t)) \\
&= append\,(rev\, l_2)\;(rev\, l_1)
\end{aligned}$$

Q.E.D.

Theorem 7.6 *For any list* l *we have* $rev(rev\, l) = l$.
Proof: *Structural induction on* l.

- *If* $l = []$ *we have:*

$$\begin{aligned}
rev(rev\, l) &= rev(rev\, []) \\
&= rev\, [] \\
&= [] \\
&= l
\end{aligned}$$

- *Now suppose* $l = h :: t$ *and we know that*

$$rev(rev\, t) = t$$

then we find:

$$\begin{aligned}
rev(rev\, l) &= rev(rev\,(h :: t)) \\
&= rev(append\,(rev\, t)\;[h]) \\
&= append\,(rev\,[h])\;(rev(rev\, t)) \\
&= append\,(rev\,[h])\; t \\
&= append\,(rev\,(h :: []))\; t
\end{aligned}$$

$$
\begin{aligned}
&= append\,(append\,(rev\,[])\;[h])\;t \\
&= append\,(append\,[]\;[h])\;t \\
&= append\,[h]\;t \\
&= append\,(h :: [])\;t \\
&= h :: (append\,[]\;t) \\
&= h :: t \\
&= l
\end{aligned}
$$

Q.E.D.

There are lots of theorems relating the list operations that can be proved in a similar style. Generally, one proceeds by list induction. In subtler cases, one has to split off lemmas which are themselves proved inductively, and sometimes to generalize the theorem before proceeding by induction. There are plenty of examples in the exercises for readers to try their hand at.

Further reading

Neumann (1995) catalogues the dangers arising from the use of computers in society, including those arising from software bugs. A very readable discussion of the issues, with extensive discussion of some interesting examples, is given by Peterson (1996). The general issue of software verification was at one time controversial, with DeMillo, Lipton, and Perlis (1979) among others arguing against it. A cogent discussion is given by Barwise (1989). There is now a large literature in software verification, though much of it is concerned with imperative programs. Many introductory functional programming books such as Paulson (1991) and Reade (1989) give some basic examples like the ones here. It is also worth looking at the work of Boyer and Moore (1979) on verifying properties of pure LISP functions expressed inside their theorem prover. One of the largest real proofs of the type we consider here is the verification by Aagaard and Leeser (1994) of a boolean simplifier used in VLSI logic synthesis.

Exercises

1. Prove the correctness of the more efficient program we gave for performing exponentiation:

```
#let square x = x * x;;
#let rec exp x n =
  if n = 0 then 1
  else if n mod 2 = 0 then square(exp x (n / 2))
  else x * square(exp x (n / 2));;
```

2. Recall the definition of `length`:

```
#let rec length =
   fun [] -> 0
     | (h::t) -> 1 + length t;;
```

Prove that $\text{length}(\text{rev}\ \ l) = \text{length}\ \ l$ and that $\text{length}(\text{append}\ \ l_1\ \ l_2) = \text{length}\ l_1 + \text{length}\ l_2$.

3. Define the `map` function, which applies a function to each element of a list, as follows:

```
#let rec map f =
   fun [] -> []
     | (h::t) -> (f h)::(map f t);;
```

Prove that if $l \neq []$ then $\text{hd}(\text{map}\ f\ \ l) = f(\text{hd}\ l)$, and that $\text{map}\ f\ \ (\text{rev}\ l) = \text{rev}\ (\text{map}\ f\ \ l)$. Recall the definition of function composition:

```
#let o f g = fun x -> f(g x);;
#infix "o";;
```

Prove that $\text{map}\ f\ \ (\text{map}\ g\ l) = \text{map}\ (f \circ g)\ l$.

4. McCarthy's '91 function' can be defined by:

```
#let rec f x = if x > 100 then x - 10
               else f(f(x + 11));;
```

Prove that for $n \leq 101$, we have $f(n) = 91$. Pay careful attention to establishing termination. (Hint: a possible measure is $101 - x$.)

5. The problem of the Dutch National Flag is to sort a list of 'colours' (red, white and blue) so that the reds come first, then the whites and then the blues. However, the only method permitted is to swap adjacent elements. Here is an ML solution. The function `dnf` returns 'true' iff it has made a change, and this function is then repeated until no change occurs.

```
type colour = Red | White | Blue;;

let rec dnf =
   fun [] -> [],false
     | (White::Red::rest) -> Red::White::rest,true
     | (Blue::Red::rest) -> Red::Blue::rest,true
     | (Blue::White::rest) -> White::Blue::rest,true
     | (x::rest) -> let fl,ch = dnf rest in x::fl,ch;;

let rec flag l =
  let l',changed = dnf l in
  if changed then flag l' else l';;
```

For example:

```
#flag [White; Red; Blue; Blue; Red; White; White; Blue; Red];;
- : colour list =
  [Red; Red; Red; White; White; White; Blue; Blue; Blue]
```

Prove that the function `flag` always terminates with a correctly sorted list.

6. (*) Define the following functions:

```
#let rec sorted =
   fun [] -> true
     | [h] -> true
     | (h1::h2::t) -> h1 <= h2 & sorted(h2::t);;

#let rec filter p =
   fun [] -> []
     | (h::t) -> let t' = filter p t in
                 if p h then h::t' else t';;

#let sameprop p l1 l2 =
   length(filter p l1) = length(filter p l2);;

#let rec permutes l1 l2 =
   fun [] -> true
     | (h::t) -> sameprop (fun x -> x = h) l1 l2 &
                 permutes l1 l2 t;;

#let permuted l1 l2 = permutes l1 l2 l1;;
```

What do they all do? Implement a sorting function `sort` and prove that for all lists l one has sorted(sort l) = true and permuted l (sort l) = true.

Chapter 8

Effective ML

In this chapter, we discuss some of the techniques and tricks that ML programmers can use to make programs more elegant and more efficient. We then go on to discuss some additional *imperative* features that can be used when the purely functional style seems inappropriate.

8.1 Useful combinators

The flexibility of higher order functions often means that one can write some very useful little functions that can be re-used for a variety of related tasks. These are often called *combinators*, and not simply because they can be regarded as lambda terms with no free variables. It often turns out that these functions are so flexible that practically anything can be implemented by plugging them together, rather than, say, explicitly making a recursive definition. In this way they correspond to the original view of combinators as ubiquitous building blocks for mathematical expressions.

For example, a very useful combinator for list operations, often called '`itlist`' or '`fold`', performs the following operation:

$$\texttt{itlist}\ f\ [x_1;\ x_2;\ \ldots\ ;x_n]\ b = f\ x_1\ (f\ x_2\ (f\ x_3\ (\cdots (f\ x_n\ b))))$$

A straightforward definition in ML is:

```
#let rec itlist f =
   fun [] b -> b
     | (h::t) b -> f h (itlist f t b);;
itlist : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

Quite commonly, when defining a recursive function over lists, all one is doing is repeatedly applying some operator in this manner. By using `itlist` with the appropriate argument, one can implement such functions very easily without explicit use of recursion. A typical use is a function to add all the elements of a list of numbers:

```
#let sum l = itlist (fun x sum -> x + sum) l 0;;
sum : int list -> int = <fun>
#sum [1;2;3;4;5];;
- : int = 15
#sum [];;
- : int = 0
#sum [1;1;1;1];;
- : int = 4
```

Those especially keen on brevity might prefer to code `sum` as:

```
#let sum l = itlist (prefix +) l 0;;
```

It is easy to modify this function to form a product rather than a sum:

```
#let prod l = itlist (prefix *) l 1;;
```

Many useful list operations can be implemented in this way. For example here is a function to filter out only those elements of a list satisfying a predicate:

```
#let filter p l = itlist (fun x s -> if p x then x::s else s) l [];;
filter : ('a -> bool) -> 'a list -> 'a list = <fun>
#filter (fun x -> x mod 2 = 0) [1;6;4;9;5;7;3;2];;
- : int list = [6; 4; 2]
```

Here are functions to find whether either all or some of the elements of a list satisfy a predicate:

```
#let forall p l = itlist (fun h a -> p(h) & a) l true;;
forall : ('a -> bool) -> 'a list -> bool = <fun>
#let exists p l = itlist (fun h a -> p(h) or a) l false;;
exists : ('a -> bool) -> 'a list -> bool = <fun>
#forall (fun x -> x < 3) [1;2];;
- : bool = true
#forall (fun x -> x < 3) [1;2;3];;
- : bool = false
```

and here are alternative versions of old favourites `length`, `append` and `map`:

```
#let length l = itlist (fun x s -> s + 1) l 0;;
length : 'a list -> int = <fun>
#let append l m = itlist (fun h t -> h::t) l m;;
append : 'a list -> 'a list -> 'a list = <fun>
#let map f l = itlist (fun x s -> (f x)::s) l [];;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

Some of these functions can themselves become useful combinators, and so on upwards. For example, if we are interested in treating lists as sets, i.e. avoiding duplicate elements, then many of the standard set operations can be expressed very simply in terms of the combinators above:

```
#let mem x l = exists (fun y -> y = x) l;;
mem : 'a -> 'a list -> bool = <fun>
#let insert x l =
  if mem x l then l else x::l;;
insert : 'a -> 'a list -> 'a list = <fun>
#let union l1 l2 = itlist insert l1 l2;;
union : 'a list -> 'a list -> 'a list = <fun>
#let setify l = union l [];;
setify : 'a list -> 'a list = <fun>
#let Union l = itlist union l [];;
Union : 'a list list -> 'a list = <fun>
#let intersect l1 l2 = filter (fun x -> mem x l2) l1;;
intersect : 'a list -> 'a list -> 'a list = <fun>
#let subtract l1 l2 = filter (fun x -> not mem x l2) l1;;
subtract : 'a list -> 'a list -> 'a list = <fun>
#let subset l1 l2 = forall (fun t -> mem t l2) l1;;
subset : 'a list -> 'a list -> bool = <fun>
```

The `setify` function is supposed to turn a list into a set by eliminating any duplicate elements.

8.2 Writing efficient code

Here we accumulate some common tricks of the trade, which can often make ML programs substantially more efficient. In order to justify some of them, we need to sketch in general terms how certain constructs are executed in hardware.

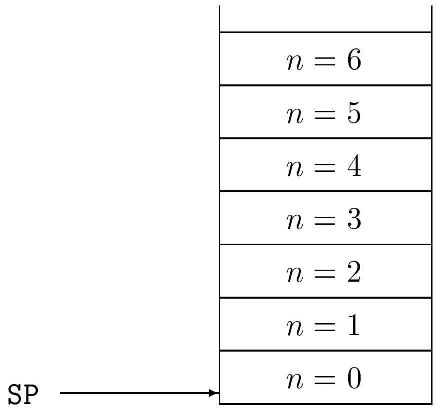
8.2.1 Tail recursion and accumulators

The principal control mechanism in functional programs is recursion. If we are interested in efficient programs, it behoves us to think a little about how recursion is implemented on conventional hardware. In fact, there is not, in this respect at least, much difference between the implementation of ML and many other languages with dynamic variables, such as C.

If functions cannot be called recursively, then we are safe in storing their local variables (which includes the values of arguments) at a fixed place in memory — this is what FORTRAN does. However, this is not possible in general if the function can be called recursively. A call to a function `f` with one set of arguments may include within it a call to `f` with a different set of arguments. The old ones would be overwritten, even if the outer version of f needs to refer to them again after the inner call has finished. For example, consider the factorial function yet again:

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

A call to `fact 6` causes another call to `fact 5` (and beyond), but when this call is finished and the value of `fact 5` is obtained, we still need the original value of `n`, namely 6, in order to do the multiplication yielding the final result. What normally happens in implementations is that each function call is allocated a new frame on a *stack*. Every new function call moves the stack pointer further down[1] the stack, creating space for new variables. When the function call is finished the stack pointer moves up and so the unwanted inner variables are discarded automatically. A diagram may make this clearer:



This is an imagined snapshot of the stack during execution of the innermost recursive call, i.e. `fact 0`. All the local variables for the upper stages are stacked up above, with each instance of the function having its own stack frame, and when the calls are finished the stack pointer `SP` moves back up.

Therefore, our implementation of `fact` requires n stack frames when applied to argument n. By contrast, consider the following implementation of the factorial function:

```
#let rec tfact x n =
   if n = 0 then x
   else tfact (x * n) (n - 1);;
tfact : int -> int -> int = <fun>
#let fact n = tfact 1 n;;
fact : int -> int = <fun>
#fact 6;;
- : int = 720
```

Although `tfact` is also recursive, the recursive call is the whole expression; it does not occur as a proper subexpression of some other expression involving values of variables. Such a call is said to be a *tail call* (because it is the very last thing the calling function does), and a function where all recursive calls are tail calls is said to be *tail recursive*.

[1] Despite the name, stacks conventionally grow downwards.

What is significant about tail calls? When making a recursive call to `tfact`, there is no need to preserve the old values of the local variables. Exactly the same, fixed, area of storage can be used. This of course depends on the compiler's being intelligent enough to recognize the fact, but most compilers, including CAML Light, are. Consequently, re-coding a function so that the recursive core of it is tail recursive can dramatically cut down the use of storage. For functions like the factorial, it is hardly likely that they will be called with large enough values of n to make the stack overflow. However the naive implementations of many list functions can cause such an effect when the argument lists are long.

The additional argument `x` of the `tfact` function is called an *accumulator*, because it accumulates the result as the recursive calls rack up, and is then returned at the end. Working in this way, rather than modifying the return value on the way back up, is a common way of making functions tail recursive.

We have remarked that a fixed area of storage can be used for the arguments to a tail recursive function. On this view, one can look at a tail recursive function as a thinly-veiled imperative implementation. There is an obvious parallel with our C implementation of the factorial as an iterative function:

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

The initialization `x = 1` corresponds to our setting of `x` to 1 by an outer wrapper function `fact`. The central while loop corresponds to the recursive calls, the only difference being that the arguments to the tail recursive function make explicit that part of the state we are interested in assigning to. Rather than assigning and looping, we make a recursive call with the variables updated. Using similar tricks and making the state explicit, one can easily write essentially imperative code in an ostensibly functional style, with the knowledge that under standard compiler optimizations, the effect inside the machine will, in fact, be much the same.

8.2.2 Minimizing consing

We have already considered the use of stack space. But various constructs in functional programs use another kind of store, usually allocated from an area called the *heap*. Whereas the stack grows and shrinks in a sequential manner based on the flow of control between functions, other storage used by the ML system cannot be reclaimed in such a simple way. Instead, the runtime system occasionally needs to check which bits of allocated memory aren't being used any more, and reclaim them for future applications, a process known as *garbage collection*. A particularly important example is the space used by constructors for recursive types, e.g. `::`. For example, when the following fragment is executed:

```
let l = 1::[] in tl l;;
```

a new block of memory, called a 'cons cell', is allocated to store the instance of the `::` constructor. Typically this might be three words of storage, one being an identifier for the constructor, and the other two being pointers to the head and tail of the list. Now in general, it is difficult to decide when this memory can be reclaimed. In the above example, we immediately select the tail of the list, so it is clear that the cons cell can be recycled immediately. But in general this can't be decided by looking at the program, since l might be passed to various functions that may or may not just look at the components of the list. Instead, one needs to analyze the memory usage dynamically and perform garbage collection of what is no longer needed. Otherwise one would eventually run out of storage even when only a small amount is ever needed simultaneously.

Implementors of functional languages work hard on making garbage collection efficient. Some claim that automatic memory allocation and garbage collection often works out faster than typical uses of explicit memory allocation in languages like C (`malloc` etc.) While we wouldn't go that far, it is certainly very convenient that memory allocation is always done automatically. It avoids a lot of tedious and notoriously error-prone parts of programming.

Many constructs beloved of functional programmers use storage that needs to be reclaimed by garbage collection. While worrying too much about this would cripple the style of functional programs, there are some simple measures that can be taken to avoid gratuitous consing (creation of cons cells). One very simple rule of thumb is to avoid using `append` if possible. As can be seen by considering the way the recursive calls unroll according to the definition

```
#let rec append l1 l2 =
   match l1 with
     [] -> l2
   | (h::t) -> h::(append t l2);;
```

this typically generates n cons cells where n is the length of the first argument list. There are often ways of avoiding appending, such as adding extra accumulator arguments to functions that can be augmented by direct use of consing. A striking example is the list reversal function, which we coded earlier as:

```
#let rec rev =
   fun [] -> []
     | (h::t) -> append (rev t) [h];;
```

This typically generates about $n^2/2$ cons cells, where n is the length of the list. The following alternative, using an accumulator, only generates n of them:

```
#let rev =
   let rec reverse acc =
     fun [] -> acc
       | (h::t) -> reverse (h::acc) t in
   reverse [];;
```

Moreover, the recursive core `reverse` is tail recursive, so we also save stack space, and win twice over.

For another typical situation where we can avoid appending by judicious use of accumulators, consider the problem of returning the fringe of a binary tree, i.e. a list of the leaves in left-to-right order. If we define the type of binary trees as:

```
#type btree = Leaf of string
            | Branch of btree * btree;;
```

then a simple coding is the following

```
#let rec fringe =
   fun (Leaf s) -> [s]
     | (Branch(l,r)) -> append (fringe l) (fringe r);;
```

However the following more refined version performs fewer conses:

```
#let fringe =
   let rec fr t acc =
     match t with
       (Leaf s) -> s::acc
     | (Branch(l,r)) -> fr l (fr r acc) in
   fun t -> fr t [];;
```

Note that we have written the accumulator as the second argument, so that the recursive call has a more natural left-to-right reading. Here is a simple example of how either version of `fringe` may be used:

```
#fringe (Branch(Branch(Leaf "a",Leaf "b"),
                Branch(Leaf "c",Leaf "d")));;
- : string list = ["a"; "b"; "c"; "d"]
```

The first version creates 6 cons cells, the second only 4. On larger trees the effect can be more dramatic. Another situation where gratuitous consing can crop up is in pattern matching. For example, consider the code fragment:

```
fun [] -> []
  | (h::t) -> if h < 0 then t else h::t;;
```

The 'else' arm creates a cons cell even though what it constructs was in fact the argument to the function. That is, it is taking the argument apart and then rebuilding it. One simple way of avoiding this is to recode the function as:

```
fun l ->
  match l with
    [] -> []
  | (h::t) -> if h < 0 then t else l;;
```

However ML offers a more flexible alternative: using the `as` keyword, a name may be identified with certain components of the pattern, so that it never needs to be rebuilt. For example:

```
fun [] -> []
  | (h::t as l) -> if h < 0 then t else l;;
```

8.2.3 Forcing evaluation

We have emphasized that, since ML does not evaluate underneath lambdas, one can use lambdas to delay evaluation. We will see some interesting examples later. Conversely, however, it can happen that one wants to force evaluation of expressions that are hidden underneath lambdas. For example, recall the tail recursive factorial above:

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

Since we never really want to use `tfact` directly, it seems a pity to bind it to a name. Instead, we can make it local to the factorial function:

```
#let fact1 n =
   let rec tfact x n =
     if n = 0 then x
     else tfact (x * n) (n - 1) in
   tfact 1 n;;
```

This, however, has the defect that the local recursive definition is only evaluated after `fact1` receives its argument, since before that it is hidden under a lambda. Moreover, it is then reevaluated each time `fact` is called. We can change this as follows

```
#let fact2 =
   let rec tfact x n =
     if n = 0 then x
     else tfact (x * n) (n - 1) in
   tfact 1;;
```

Now the local binding is only evaluated once, at the point of declaration of `fact2`. According to our tests, the second version of `fact` is about 20% faster when called on the argument `6`. The additional evaluation doesn't amount to much in this case, more or less just unravelling a recursive definition, yet the speedup is significant. In instances where there is a lot of computation involved in evaluating the local binding, the difference can be spectacular. In fact, there is a sophisticated research field of 'partial evaluation' devoted to performing optimizations like this, and much more sophisticated ones besides, automatically. In a sense, it is a generalization of standard compiler optimizations for ordinary languages such as 'constant folding'. In production ML systems, however, it is normally the responsibility of the user to force it, as here.

We might note, in passing, that if functions are implemented by plugging together combinators, with fewer explicit lambdas, there is more chance that as much of the expression as possible will be evaluated at declaration time. To take a trivial example, $f \circ g$ will perform any evaluation of f and g that may be possible, whereas $\lambda x.\ f(g\ x)$ will perform none at all until it receives its argument. On the other side of the coin, when we actually *want* to delay evaluation, we really need lambdas, so a purely combinatory version is impossible.

8.3 Imperative features

ML has a fairly full complement of imperative features. We will not spend much time on the imperative style of programming, since that is not the focus of this course, and we assume readers already have sufficient experience. Therefore, we treat these topics fairly quickly with few illustrative examples. However some imperative features are used in later examples, and some knowledge of what is available will stand the reader in good stead for writing practical ML code.

8.3.1 Exceptions

We have seen on occasion that certain evaluations fail, e.g. through a failure in pattern matching. There are other reasons for failure, e.g. attempts to divide by zero.

```
#1 / 0;;
Uncaught exception: Division_by_zero
```

In all these cases the compiler complains about an 'uncaught exception'. An exception is a kind of error indication, but they need not always be propagated to the top level. There is a type `exn` of *exceptions*, which is effectively a recursive type, though it is usually recursive only vacuously. Unlike with ordinary types, one can add new constructors for the type `exn` at any point in the program via an exception declaration, e.g.

```
#exception Died;;
Exception Died defined.
#exception Failed of string;;
Exception Failed defined.
```

While certain built-in operations generate (one usually says *raise*) exceptions, this can also be done explicitly using the `raise` construct, e.g.

```
#raise (Failed "I don't know why");;
Uncaught exception: Failed "I don't know why"
```

For example, we might invent our own exception to cover the case of taking the head of an empty list:

```
#exception Head_of_empty;;
Exception Head_of_empty defined.
#let hd = fun [] -> raise Head_of_empty
            | (h::t) -> h;;
hd : 'a list -> 'a = <fun>
#hd [];;
Uncaught exception: Head_of_empty
```

Normally exceptions propagate out to the top, but they can be 'caught' inside an outer expression by using `try ...with` followed by a series of patterns to match exceptions, e.g.

```
#let headstring sl =
   try hd sl
   with Head_of_empty -> ""
      | Failed s -> "Failure because "^s;;
headstring : string list -> string = <fun>
#headstring ["hi"; "there"];;
- : string = "hi"
#headstring [];;
- : string = ""
```

It is a matter of opinion whether exceptions are really an imperative feature. On one view, functions just return elements of a disjoint sum consisting of their visible return type and the type of exceptions, and all operations implicitly pass back exceptions. Another view is that exceptions are a highly non-local control flow perversion, analogous to `goto`.[2] Whatever the semantic view one takes, exceptions can often be quite useful.

[2] Perhaps more precisely, to C's `setjmp` and `longjmp`.

8.3.2 References and arrays

ML does have real assignable variables, and expressions can, as a side-effect, modify the values of these variables. They are explicitly accessed via *references* (pointers in C parlance) and the references themselves behave more like ordinary ML values. Actually this approach is quite common in C too. For example, if one wants so-called 'variable parameters' in C, where changes to the local variables propagate outside, the only way to do it is to pass a pointer, so that the function can dereference it. Similar techniques are often used where the function is to pass back composite data.

In ML, one sets up a new assignable memory cell with the initial contents `x` by writing `ref x`. (Initialization is compulsory.) This expression yields a reference (pointer) to the cell. Subsequent access to the contents of the cell requires an explicit dereference using the `!` operator, similar to unary `*` in C. The cell is assigned to using a conventional-looking assignment statement. For example:

```
#let x = ref 1;;
x : int ref = ref 1
#!x;;
- : int = 1
#x := 2;;
- : unit = ()
#!x;;
- : int = 2
#x := !x + !x;;
- : unit = ()
#x;;
- : int ref = ref 4
#!x;;
- : int = 4
```

Note that in most respects `ref` behaves like a type constructor, so one can pattern-match against it. Thus one could actually define an indirection operator like `!`:

```
#let contents_of (ref x) = x;;
contents_of : 'a ref -> 'a = <fun>
#contents_of x;;
- : int = 4
```

As well as being mutable, references are sometimes useful for creating explicitly shared data structures. One can easily create graph structures where numerous nodes contain a pointer to some single subgraph.

Apart from single cells, one can also use arrays in ML. In CAML these are called `vectors`. An array of elements of type α has type α *vect*. A fresh vector of size `n`, with each element initialized to `x` — once again the initialization is compulsory — is created using the following call:

```
#make_vect n x;;
```

One can then read element `m` of a vector `v` using:

```
#vect_item v m;;
```

and write value `y` to element `m` of `v` using:

```
#vect_assign v m y;;
```

These operations correspond to the expressions `v[m]` and `v[m] = y` in C. The elements of an array are numbered from zero. For example:

```
#let v = make_vect 5 0;;
v : int vect = [|0; 0; 0; 0; 0|]
#vect_item v 1;;
- : int = 0
#vect_assign v 1 10;;
- : unit = ()
#v;;
- : int vect = [|0; 10; 0; 0; 0|]
#vect_item v 1;;
- : int = 10
```

All reading and writing is constrained by bounds checking, e.g.

```
#vect_item v 5;;
Uncaught exception: Invalid_argument "vect_item"
```

8.3.3 Sequencing

There is no need for an explicit sequencing operation in ML, since the normal rules of evaluation allow one to impose an order. For example one can do:

```
#let _ =  x := !x + 1 in
 let _ =  x := !x + 1 in
 let _ =  x := !x + 1 in
 let _ =  x := !x + 1 in
 ();;
```

and the expressions are evaluated in the expected order. Here we use a special pattern _ which throws away the value, but we could use a dummy variable name instead. Nevertheless, it is more attractive to use the conventional notation for sequencing, and this is possible in ML by using a single semicolon:

```
#x := !x + 1;
 x := !x + 1;
 x := !x + 1;
 x := !x + 1;;
```

8.3.4 Interaction with the type system

While polymorphism works very well for the pure functional core of ML, it has unfortunate interactions with some imperative features. For example, consider the following:

```
#let l = ref [];;
```

Then `l` would seem to have polymorphic type α *list ref*. In accordance with the usual rules of let-polymorphism we should be able to use it with two different types, e.g. first

```
#l := [1];;
```

and then

```
#hd(!l) = true;;
```

But this isn't reasonable, because we would actually be writing something as an object of type `int` then reading it as an object of type `bool`. Consequently, some restriction on the usual rule of let polymorphism is called for where references are concerned. There have been many attempts to arrive at a sound but convenient restriction of the ML type system, some of them very complicated. Recently, different versions of ML seem to be converging on a relatively simple method, called the *value restriction*, due to Wright (1996), and CAML implements this restriction, with a twist regarding toplevel bindings. Indeed, the above sequence fails. But the intermediate behaviour is interesting. If we look at the first line we see:

```
#let l = ref [];;
l : '_a list ref = ref []
```

The underscore on the type variable indicates that `l` is not polymorphic in the usual sense; rather, it has a single fixed type, although that type is as yet undetermined. The second line works fine:

```
#l := [1];;
- : unit = ()
```

but if we now look at the type of `l`, we see that:

```
#l;;
- : int list ref = ref [1]
```

The pseudo-polymorphic type has now been fixed. Granted this, it is clear that the last line must fail:

```
#hd(!l) = true;;
Toplevel input:
>hd(!l) = true;;
>         ^^^^
This expression has type bool,
but is used with type int.
```

So far, this seems quite reasonable, but we haven't yet explained why the same underscored type variables oocur in apparently quite innocent purely functional expressions, and why, moreover, they often disappear on eta-expansion, e.g.

```
#let I x = x;;
I : 'a -> 'a = <fun>
#I o I;;
it : '_a -> '_a = <fun>
#let I2 = I o I in fun x -> I2 x;;
- : '_a -> '_a = <fun>
#fun x -> (I o I) x;;
it : 'a -> 'a = <fun>
```

Other techniques for polymorphic references often rely on encoding in the types the fact that an expression may involve references. This seems natural, but it can lead to the types of functions becoming cluttered with this special information. It is unattractive that the particular implementation of the function, e.g. imperative or functional, should be reflected in its type.

Wright's solution, on the other hand, uses just the basic syntax of the expression being let-bound, insisting that it is a so-called *value* before generalizing the type. What is really wanted is knowledge of whether the expression may cause side-effects when evaluated. However since this is undecidable in general, the simple syntactic criterion of its being or not being a value is used. Roughly speaking, an expression is a value if it admits no further evaluation according to the ML rules — this is why an expression can often be made into a value by performing a reverse eta conversion. Unfortunately this works against the techniques for forcing evaluation.

Exercises

1. Define the C combinator as:

```
#let C f x y = f y x;;
```

What does the following function do?

```
#fun f l1 l2 -> itlist (union o C map l2 o f) l1 [];;
```

2. What does this function do? Write a more efficient version.

```
#let rec upto n = if n < 1 then [] else append (upto (n-1)) [n];;
```

3. Define a function to calculate Fibonacci numbers as follows:

```
#let rec fib =
   fun 0 -> 0
     | 1 -> 1
     | n -> fib(n - 2) + fib(n - 1);;
```

   Why is this very inefficient? Write a similar but better version.

4. (*) Can you think of any uses for this exception or similar recursive ones?

```
#exception Recurse of exn;;
```

5. Implement a simple version of quicksort using arrays. The array is first partitioned about some pivot, then two recursive calls sort the left and right portions. Which recursive calls are tail calls? What is the worst-case space usage? How can this be improved dramatically by a simple change?

6. Prove that the two versions of `rev` that we have quoted, inefficient and efficient, always give the same answer.

Chapter 9

Examples

As we have said, ML was originally designed as a metalanguage for a computer theorem prover. However it is suitable for plenty of other applications, mostly from the same general field of 'symbolic computation'. In this chapter we will give some examples of characteristic uses of ML. It is not necessary that the reader should understand every detail of the particular examples, e.g. the analyses of real number approximations given below. However it is worth trying out these programs and similar examples for yourself, and doing some of the exercises. There is no better way of getting a feel for how ML is actually used.

9.1 Symbolic differentiation

The phrase 'symbolic computation' is a bit vague, but roughly speaking it covers applications where manipulation of mathematical *expressions*, in general containing variables, is emphasized at the expense of actual numerical calculation. There are several successful 'computer algebra systems' such as Axiom, Maple and Mathematica, which can do certain symbolic operations that are useful in mathematics, e.g. factorizing polynomials and differentiating and integrating expressions. (They are also capable of purely numerical work.) We will illustrate how this sort of application can be tackled using ML.

We use symbolic differentiation as our example, because there is an algorithm to do it which is fairly simple and well known. The reader is probably familiar with certain derivatives of basic operations, e.g $\frac{d}{dx} sin(x) = cos(x)$, as well as results such as the Chain Rule and Product Rule for calculating the derivatives of composite expressions in terms of the derivatives of the components. Just as one can use these systematically in order to find derivatives, it is possible to program a fairly straightforward algorithm to do it in ML.

9.1.1 First order terms

We will allow mathematical expressions in a fairly general form. They may be built up from variables and constants by the application of operators. These operators may be unary, binary, ternary, or in general n-ary. We represent this using the following recursive datatype:

```
#type term = Var of string
           | Const of string
           | Fn of string * (term list);;
Type term defined.
```

For example, the expression:

$$sin(x + y)/cos(x - exp(y)) - ln(1 + x)$$

is represented by:

```
Fn("-",[Fn("/",[Fn("sin",[Fn("+",[Var "x"; Var "y"])]);
                Fn("cos",[Fn("-",[Var "x"; Fn("exp",[Var "y"])])])]);
        Fn("ln",[Fn("+",[Const "1"; Var "x"])])]);;
```

9.1.2 Printing

Reading and writing expressions in their raw form is rather unpleasant. This is a general problem in all symbolic computation systems, and typically the system's interface contains a *parser* and *prettyprinter* to allow respectively input and output in a readable form. A detailed discussion of parsing is postponed, since it is worth a section of its own, but we will now write a very simple printer for expressions, so that at least we can see what's going on. We want the printer to support ordinary human conventions, i.e.

- Variables and constants are just written as their names.
- Ordinary n-ary functions applied to arguments are written by juxtaposing the function and a bracketed list of arguments, e.g. $f(x_1, \ldots, x_n)$.
- Infix binary functions like + are written in between their arguments.
- Brackets are used where necessary for disambiguation.
- Infix operators have a notion of precedence to reduce the need for brackets.

First we declare a list of infixes, which is a list of pairs: each operator name together with its precedence.

```
#let infixes = ["+",10; "-",10; "*",20; "/",20];;
```

It is common to use lists in this way to represent finite partial functions, since it allows more flexibility. They are commonly called *association lists*, and are very common in functional programming.[1] In order to convert the list into a partial function we use `assoc`:

```
#let rec assoc a ((x,y)::rest) = if a = x then y else assoc a rest;;
Toplevel input:
>let rec assoc a ((x,y)::rest) = if a = x then y else assoc a rest;;
>        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
assoc : 'a -> ('a * 'b) list -> 'b = <fun>
```

The compiler warns us that if the appropriate data is not found in the list, the function will fail. Now we can define a function to get the infix precedence of a function:

```
#let get_precedence s = assoc s infixes;;
get_precedence : string -> int = <fun>
#get_precedence "+";;
- : int = 10
#get_precedence "/";;
- : int = 20
#get_precedence "%";;
Uncaught exception: Match_failure ("", 6544, 6601)
```

Note that if we ever change this list of infixes, then any functions such as `get_precedence` that use it need to be redeclared. This is the main reason why many LISP programmers prefer dynamic binding. However we can make the set of infixes arbitrarily extensible by making it into a reference and dereferencing it on each application:

```
#let infixes = ref ["+",10; "-",10; "*",20; "/",20];;
infixes : (string * int) list ref =
  ref ["+", 10; "-", 10; "*", 20; "/", 20]
#let get_precedence s = assoc s (!infixes);;
get_precedence : string -> int = <fun>
#get_precedence "^";;
Uncaught exception: Match_failure ("", 6544, 6601)
#infixes := ("^",30)::(!infixes);;
- : unit = ()
#get_precedence "^";;
- : int = 30
```

[1] For more heavyweight applications, some alternative such as hash tables is much better, but for simple applications where the lists don't get too long, this kind of linear list is simple and adequate.

Note that by the ML evaluation rules, the dereferencing is only applied after the function `get_precedence` receives its argument, and hence results in the set of infixes at that time. We can also define a function to decide whether a function has any data associated with it. One way is simply to try the function `get_precedence` and see if it works:

```
#let is_infix s =
   try get_precedence s; true
   with _ -> false;;
```

An alternative is to use a general function `can` that finds out whether another function succeeds:

```
#let can f x = try f x; true with _ -> false;;
can : ('a -> 'b) -> 'a -> bool = <fun>
#let is_infix = can get_precedence;;
is_infix : string -> bool = <fun>
```

We will use the following functions that we have already considered:

```
#let hd(h::t) = h;;
#let tl(h::t) = t;;
#let rec length l = if l = [] then 0 else 1 + length(tl l);;
```

Without further ado, here is a function to convert a term into a string.

```
#let rec string_of_term prec =
   fun (Var s) -> s
     | (Const c) -> c
     | (Fn(f,args)) ->
         if length args = 2 & is_infix f then
           let prec' = get_precedence f in
           let s1 = string_of_term prec' (hd args)
           and s2 = string_of_term prec' (hd(tl args)) in
           let ss = s1^" "^f^" "^s2 in
           if prec' <= prec then "("^ss^")" else ss
         else
           f^"("^(string_of_terms args)^")"

 and string_of_terms t =
   match t with
       [] -> ""
     | [t] -> string_of_term 0 t
     | (h::t) -> (string_of_term 0 h)^","^(string_of_terms t);;
```

The first argument `prec` indicates the precedence level of the operator of which the currently considered expression is an immediate subterm. Now if the current expression has an infix binary operator at the top level, parentheses are needed if the precedence of this operator is lower, e.g. if we are considering the right hand subexpression of $x * (y + z)$. Actually we use parentheses even when they are equal, in order to make groupings clear. In the case of associative operators like $+$ this doesn't matter, but we must distinguish $x - (y - z)$ and $(x - y) - z$. (A more sophisticated scheme would give each operator an associativity.) The second, mutually recursive, function `string_of_terms` is used to print a comma-separated list of terms, as they occur for non-infix function applications of the form $f(t_1, \ldots, t_n)$. Let us see this in action:

```
#let t =
  Fn("-",[Fn("/",[Fn("sin",[Fn("+",[Var "x"; Var "y"])]);
                  Fn("cos",[Fn("-",[Var "x";
                                    Fn("exp",[Var "y"])])])]);
          Fn("ln",[Fn("+",[Const "1"; Var "x"])])]);;
t : term =
 Fn
  ("-",
   [Fn
     ("/",
      [Fn ("sin", [Fn ("+", [Var "x"; Var "y"])]);
       Fn ("cos", [Fn ("-", [Var "x"; Fn ("exp", [Var "y"])])])]);
    Fn ("ln", [Fn ("+", [Const "1"; Var "x"])])])
#string_of_term 0 t;;
- : string = "sin(x + y) / cos(x - exp(y)) - ln(1 + x)"
```

In fact, we don't need to convert a term to a string ourselves. CAML Light allows us to set up our own printer so that it is used to print anything of type `term` in the standard read-eval-print loop. This needs a few special commands to ensure that we interact properly with the toplevel loop:

```
##open "format";;
#let print_term s =
   open_hvbox 0;
   print_string("`"^(string_of_term 0 s)^"`");
   close_box();;
print_term : term -> unit = <fun>
#install_printer "print_term";;
- : unit = ()
```

Now compare the effect:

```
#let t =
    Fn("-",[Fn("/",[Fn("sin",[Fn("+",[Var "x"; Var "y"])]);
                    Fn("cos",[Fn("-",[Var "x";
                                      Fn("exp",[Var "y"])])])]);
            Fn("ln",[Fn("+",[Const "1"; Var "x"])])]);;
t : term = `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
#let x = t
x : term = `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
```

Once the new printer is installed, it is used whenever CAML wants to print something of type `term`, even if it is a composite datastructure such as a pair:

```
#(x,t);;
- : term * term =
 `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`,
 `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
```

or a list:

```
#[x; t; x];;
- : term list =
 [`sin(x + y) / cos(x - exp(y)) - ln(1 + x)`;
  `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`;
  `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`]
```

This printer is rather crude in that it will not break up large expressions intelligently across lines. The CAML library `format`, from which we used a few functions above, provides for a better approach. Instead of converting the term into a string then printing it in one piece, we can make separate printing calls for different parts of it, and intersperse these with some special function calls that help the printing mechanism to understand where to break the lines. Many of the principles used in this and similar 'prettyprinting engines' are due to Oppen (1980). We will not consider this in detail here — see the CAML manual for details.[2]

9.1.3 Derivatives

Now it is a fairly simple matter to define the derivative function. First let us recall the systematic method we are taught in school:

- If the expression is one of the standard functions applied to an argument that is the variable of differentiation, e.g. $sin(x)$, return the known derivative.

- If the expression is of the form $f(x) + g(x)$ then apply the rule for sums, returning $f'(x) + g'(x)$. Likewise for subtraction etc.

- If the expression is of the form $f(x) * g(x)$ then apply the product rule, i.e. return $f'(x) * g(x) + f(x) * g'(x)$.

- If the expression is one of the standard functions applied to a composite argument, say $f(g(x))$ then apply the Chain Rule and so give $g'(x) * f'(g(x))$

[2]There is also a tutorial, written by Pierre Weis, available online as part of the CAML page: '`http://pauillac.inria.fr/caml/FAQ/format-eng.html`'.

This is nothing but a recursive algorithm, though that terminology is seldom used in schools. We can program it in ML very directly:

```
#let rec differentiate x tm = match tm with
     Var y -> if y = x then Const "1" else Const "0"
   | Const c -> Const "0"
   | Fn("-",[t]) -> Fn("-",[differentiate x t])
   | Fn("+",[t1;t2]) -> Fn("+",[differentiate x t1;
                                differentiate x t2])
   | Fn("-",[t1;t2]) -> Fn("-",[differentiate x t1;
                                differentiate x t2])
   | Fn("*",[t1;t2]) ->
      Fn("+",[Fn("*",[differentiate x t1; t2]);
              Fn("*",[t1; differentiate x t2])])
   | Fn("inv",[t]) -> chain x t
      (Fn("-",[Fn("inv",[Fn("^",[t;Const "2"])])]))
   | Fn("^",[t;n]) -> chain x t
     (Fn("*",[n; Fn("^",[t; Fn("-",[n; Const "1"])])]))
   | Fn("exp",[t]) -> chain x t tm
   | Fn("ln",[t]) -> chain x t (Fn("inv",[t]))
   | Fn("sin",[t]) -> chain x t (Fn("cos",[t]))
   | Fn("cos",[t]) -> chain x t
      (Fn("-",[Fn("sin",[t])]))
   | Fn("/",[t1;t2]) -> differentiate x
      (Fn("*",[t1; Fn("inv",[t2])]))
   | Fn("tan",[t]) -> differentiate x
      (Fn("/",[Fn("sin",[t]); Fn("cos",[t])]))
 and chain x t u =  Fn("*",[differentiate x t; u]);;
```

The `chain` function just avoids repeating the chain rule construction for many operations. Apart from that, we just systematically apply rules for sums, products etc. and the known derivatives of standard functions. Of course, we could add other functions such as the hyperbolic functions and inverse trigonometric functions if desired. A couple of cases, namely division and tan, avoid a complicated definition by applying the differentiator to an alternative formulation.

9.1.4 Simplification

If we try the differentiator out on our running example, then it works quite well:

```
#t;;
- : term = `sin(x + y) / cos(x - exp(y)) - ln(1 + x)`
#differentiate "x" t;;
- : term =
 `(((1 + 0) * cos(x + y)) * inv(cos(x - exp(y))) +
  sin(x + y) * (((1 - 0 * exp(y)) * -(sin(x - exp(y)))) *
  -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 1) * inv(1 + x)`
#differentiate "y" t;;
- : term =
 `(((0 + 1) * cos(x + y)) * inv(cos(x - exp(y))) +
  sin(x + y) * (((0 - 1 * exp(y)) * -(sin(x - exp(y)))) *
  -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 0) * inv(1 + x)`
#differentiate "z" t;;
- : term =
 `(((0 + 0) * cos(x + y)) * inv(cos(x - exp(y))) +
  sin(x + y) * (((0 - 0 * exp(y)) * -(sin(x - exp(y)))) *
  -(inv(cos(x - exp(y)) ^ 2)))) - (0 + 0) * inv(1 + x)`
```

However, it fails to make various obvious simplifications such as $0 * x = 0$ and $x + 0 = x$. Some of these redundant expressions are created by the rather unsubtle differentiation strategy which, for instance, applies the chain rule to $f(t)$ even where t is just the variable of differentiation. However, some would be hard to avoid without making the differentiator much more complicated. Accordingly, let us define a separate simplification function that gets rid of some of these unnecessary expressions.

```
#let simp =
   fun (Fn("+",[Const "0"; t])) -> t
     | (Fn("+",[t; Const "0"])) -> t
     | (Fn("-",[t; Const "0"])) -> t
     | (Fn("-",[Const "0"; t])) -> Fn("-",[t])
     | (Fn("+",[t1; Fn("-",[t2])])) -> Fn("-",[t1; t2])
     | (Fn("*",[Const "0"; t])) -> Const "0"
     | (Fn("*",[t; Const "0"])) -> Const "0"
     | (Fn("*",[Const "1"; t])) -> t
     | (Fn("*",[t; Const "1"])) -> t
     | (Fn("*",[Fn("-",[t1]); Fn("-",[t2])])) -> Fn("*",[t1; t2])
     | (Fn("*",[Fn("-",[t1]); t2])) -> Fn("-",[Fn("*",[t1; t2])])
     | (Fn("*",[t1; Fn("-",[t2])])) -> Fn("-",[Fn("*",[t1; t2])])
     | (Fn("-",[Fn("-",[t])])) -> t
     | t -> t;;
```

This just applies the simplification at the top level of the term. We need to execute it in a bottom-up sweep. This will also bring negations upwards through products. In some cases it would be more efficient to simplify in a top-down sweep, e.g. $0 * t$ for a complicated expression t, but this would need repeating for terms like $(0 + 0) * 2$. The choice is rather like that between normal and applicative order reduction in lambda calculus.

```
#let rec dsimp =
   fun (Fn(fn,args)) -> simp(Fn(fn,map dsimp args))
     | t -> simp t;;
```

Now we get better results:

```
#dsimp(differentiate "x" t);;
- : term =
 `(cos(x + y) * inv(cos(x - exp(y))) +
  sin(x + y) * (sin(x - exp(y)) *
  inv(cos(x - exp(y)) ^ 2))) - inv(1 + x)`
#dsimp(differentiate "y" t);;
- : term =
 `cos(x + y) * inv(cos(x - exp(y))) -
  sin(x + y) * ((exp(y) * sin(x - exp(y))) *
  inv(cos(x - exp(y)) ^ 2))`
#dsimp(differentiate "z" t);;
- : term = `0`
```

In general, one can always add more and more sophisticated simplification. For example, consider:

```
#let t2 = Fn("tan",[Var "x"]);;
t2 : term = `tan(x)`
#differentiate "x" t2;;
- : term =
 `(1 * cos(x)) * inv(cos(x)) +
  sin(x) * ((1 * -(sin(x))) * -(inv(cos(x) ^ 2)))`
#dsimp(differentiate "x" t2);;
- : term = `cos(x) * inv(cos(x)) +
            sin(x) * (sin(x) * inv(cos(x) ^ 2))`
```

We would like to simplify `cos(x) * inv(cos(x))` simply to `1`, ignoring, as most commercial computer algebra systems do, the possibility that `cos(x)` might be zero. We can certainly add new clauses to the simplifier for that. Similarly, we might like to collect up terms in the second summand. However here we start to see that human psychology plays a role. Which of the following is preferable? It is hard for the machine to decide unaided.

```
sin(x) * (sin(x) * inv(cos(x) ^ 2))

sin(x) ^ 2 * inv(cos(x) ^ 2)

sin(x) ^ 2 / cos(x) ^ 2

(sin(x) / cos(x)) ^ 2

tan(x) ^ 2
```

And having chosen the last one, should it then convert

```
1 + tan(x) ^ 2
```

into

```
sec(x) ^ 2
```

Certainly, it is shorter and more conventional. However it relies on the fact that we know the definitions of these fairly obscure trig functions like `sec`. Whatever the machine decides to do, it is likely that some user will find it upsetting.

9.2 Parsing

A defect of our previous example was that the input had to be written explicitly in terms of type constructors. Here we will show how to write a parser for this situation, and make our code sufficiently general that it can easily be applied to similar formal languages. Generally speaking, the problem of parsing is as follows. A formal language is defined by a *grammar*, which is a set of *production rules* for each syntactic category. For a language of terms with two infix operators `+` and `*` and only alphanumeric variables and numeral (0, 1 etc.) constants, it might look like this:

$$
\begin{array}{rcl}
term & \longrightarrow & name\texttt{(}termlist\texttt{)} \\
 & | & name \\
 & | & \texttt{(}term\texttt{)} \\
 & | & numeral \\
 & | & \texttt{-}term \\
 & | & term \texttt{ + } term \\
 & | & term \texttt{ * } term \\
termlist & \longrightarrow & term\texttt{,}termlist \\
 & | & term
\end{array}
$$

We will assume for the moment that we know what names and numerals are, but we could have similar rules defining them in terms of single characters. Now this set of production rules gives us a way of generating the concrete, linear representation for all terms. Note that *name*, *numeral*, *term* and *termlist* are meant to be variables in the above, whereas elements like '`+`' and '`(`' written in bold type are actual characters. For example we have:

$$\begin{aligned}
term & \longrightarrow term \texttt{ * } term \\
& \longrightarrow term \texttt{ * (}term\texttt{)} \\
& \longrightarrow term \texttt{ * (}term \texttt{ + } term\texttt{)} \\
& \longrightarrow term \texttt{ * (}term \texttt{ + } term \texttt{ * } term\texttt{)} \\
& \longrightarrow numeral \texttt{ * (}name \texttt{ + } name \texttt{ * } name\texttt{)}
\end{aligned}$$

so, following similar rules for *name* and *numeral*, we can generate for example:

```
10 * (x + y * z)
```

The task of *parsing* is to reverse this process of applying production rules, i.e. to take a string and discover how it could have been generated by the production rules for a syntactic category. Typically, the output is a *parse tree*, i.e. a tree structure that shows clearly the sequence of applications of the rules.

One problem with parsing is that the grammar may be *ambiguous*, i.e. a given string can often be generated in several different ways. This is the case in our example above, for the string could also have been generated by:

$$\begin{aligned}
term & \longrightarrow term \texttt{ * } term \\
& \longrightarrow term \texttt{ * (}term\texttt{)} \\
& \longrightarrow term \texttt{ * (}term \texttt{ * } term\texttt{)} \\
& \longrightarrow term \texttt{ * (}term \texttt{ + } term \texttt{ * } term\texttt{)} \\
& \longrightarrow numeral \texttt{ * (}name \texttt{ + } name \texttt{ * } name\texttt{)}
\end{aligned}$$

The obvious way to fix this is to specify rules of precedence and associativity for infix operators. However, we can make the grammar unambiguous without introducing additional mechanisms at the cost of adding new syntactic categories:

$$\begin{aligned}
atom & \longrightarrow name\texttt{(}termlist\texttt{)} \\
& \quad\mid\quad name \\
& \quad\mid\quad numeral \\
& \quad\mid\quad \texttt{(}term\texttt{)} \\
& \quad\mid\quad \texttt{-}atom \\
mulexp & \longrightarrow atom \texttt{ * } mulexp \\
& \quad\mid\quad atom \\
term & \longrightarrow mulexp \texttt{ + } term \\
& \quad\mid\quad mulexp \\
termlist & \longrightarrow term\texttt{,}termlist \\
& \quad\mid\quad term
\end{aligned}$$

Note that this makes both infix operators right associative. For a more complicated example of this technique in action, look at the formal definition of expressions in the ANSI C Standard.

9.2.1 Recursive descent parsing

The production rules suggest a very simple way of going about parsing using a series of mutually recursive functions. The idea is to have a function for each syntactic category, and a recursive structure that reflects exactly the mutual recursion found in the grammar itself. For example, the procedure for parsing terms, say `term` will, on encountering a `-` symbol, make a recursive call to itself to parse the subterm, and on encountering a name followed by an opening parenthesis, will make a recursive call to `termlist`. This in itself will make at least one recursive call to `term`, and so on. Such a style of programming is particularly natural in a language like ML where recursion is the principal control mechanism.

In our ML implementation, we suppose that we have some input list of objects of arbitrary type α for the parser to consume. These might simply be characters, but in fact there is usually a separate level of *lexical analysis* which collects characters together into *tokens* such as '`x12`', '`:=`' and '`96`', so by the time we reach this level the input is a list of tokens. We avoid committing ourselves to any particular type. Similarly, we will allow the parser to construct objects of arbitrary type β from its input. These might for example be parse trees as members of a recursive datatype, or might simply be numbers if the parser is to take an expression and evaluate it. Now in general, a parser might not consume all its input, so we also make it output the remaining tokens. Therefore, a parser will have type

$$(\alpha)list \to \beta \times (\alpha)list$$

For example, when given the input characters `(x + y) * z` the function `atom` should process the characters `(x + y)` and leave the remaining characters `* z`. It might return a parse tree for the processed expression using our earlier recursive type, and hence we would have:

```
atom "(x + y) * z" = Fn("+",[Var "x"; Var "y"]),"* z"
```

Since any call of the function `atom` must be from within the function `mulexp`, this second function will use the result of `atom` and deal with the input that remains, making a second call to `atom` to process the expression '`z`'.

9.2.2 Parser combinators

Another reason why this technique works particularly well in ML is that we can define some useful combinators for plugging parsers together. In fact, by giving them infix status, we can make the ML parser program look quite similar in structure to the original grammar defining the language.

First we declare an exception to be used where parsing fails. Then we define an infix `++` that applies two parsers in sequence, pairing up their results, and and infix `||` which tries first one parser, then the other. We then define a many-fold version of `++` that repeats a parser as far as possible, putting the results into a list. Finally, the infix `>>` is used to modify the results of a parser according to a function.

The CAML rules for symbolic identifiers such as `++` make them infix automatically, so we suppress this during the definition using the `prefix` keyword. Their precedences are also automatic, following from the usual precedence of their first characters as arithmetic operations etc. That is, `++` is the strongest, `>>` falls in the middle, and `||` is the weakest; this is exactly what we want.

```
exception Noparse;;

let prefix || parser1 parser2 input =
  try parser1 input
  with Noparse -> parser2 input;;

let prefix ++ parser1 parser2 input =
  let result1,rest1 = parser1 input in
  let result2,rest2 = parser2 rest1 in
  (result1,result2),rest2;;

let rec many parser input =
  try let result,next = parser input in
      let results,rest = many parser next in
      (result::results),rest
  with Noparse -> [],input;;

let prefix >> parser treatment input =
  let result,rest = parser input in
  treatment(result),rest;;
```

We will, in what follows, use the following other functions. Most of these have been defined above, the main exception being `explode` which converts a string into a list of 1-elements strings. It uses the inbuilt functions `sub_string` and `string_length`; we do not describe them in detail but their function should be easy to grasp from the example.

```
let rec itlist f =
     fun [] b -> b
       | (h::t) b -> f h (itlist f t b);;

let uncurry f(x,y) = f x y;;

let K x y = x;;

let C f x y = f y x;;

let o f g x = f(g x);;
#infix "o";;

let explode s =
  let rec exap n l =
      if n < 0 then l else
      exap (n - 1) ((sub_string s n 1)::l) in
  exap (string_length s - 1) [];;
```

In order to get started, we define a few 'atomic' parsers. The function `some` accepts any item satisfying a predicate and returns it. The function `a` is similar, but demands a particular item, and finally `finished` just makes sure there are no items left to be parsed.

```
let some p =
  fun [] -> raise Noparse
    | (h::t) -> if p h then (h,t) else raise Noparse;;

let a tok = some (fun item -> item = tok);;

let finished input =
  if input = [] then 0,input else raise Noparse;;
```

9.2.3 Lexical analysis

Our parsing combinators can easily be used, in conjunction with a few simple character discrimination functions, to construct a lexical analyzer for our language of terms. We define a type of *tokens* (or lexemes), and the lexer translates the input string into a list of tokens. The 'other' cases cover symbolic identifiers; they are all straightforward as they consist of just a single character, rather than composite ones like '`:=`'.

```
type token = Name of string | Num of string | Other of string;;

let lex =
  let several p = many (some p) in
  let lowercase_letter s = "a" <= s & s <= "z" in
  let uppercase_letter s = "A" <= s & s <= "Z" in
  let letter s = lowercase_letter s or uppercase_letter s in
  let alpha s = letter s or s = "_" or s = "'" in
  let digit s = "0" <= s & s <= "9" in
  let alphanum s = alpha s or digit s in
  let space s = s = " " or s = "\n" or s = "\t" in
  let collect(h,t) = h^(itlist (prefix ^) t "") in
  let rawname =
     some alpha ++ several alphanum >> (Name o collect) in
  let rawnumeral =
     some digit ++ several digit >> (Num o collect) in
  let rawother = some (K true) >> Other in
  let token =
    (rawname || rawnumeral || rawother) ++ several space >> fst in
  let tokens = (several space ++ many token) >> snd in
  let alltokens = (tokens ++ finished) >> fst in
  fst o alltokens o explode;;
```

For example:

```
#lex "sin(x + y) * cos(2 * x + y)";;
 - : token list =
 [Name "sin"; Other "("; Name "x"; Other "+"; Name "y"; Other ")";
  Other "*"; Name "cos"; Other "("; Num "2"; Other "*"; Name "x";
  Other "+"; Name "y"; Other ")"]
```

9.2.4 Parsing terms

Since we are now working at the level of tokens, we define trivial parsers to accept tokens of (only) a certain kind:

```
let name =
  fun (Name s::rest) -> s,rest
    | _ -> raise Noparse;;

let numeral =
  fun (Num s::rest) -> s,rest
    | _ -> raise Noparse;;

let other =
  fun (Other s::rest) -> s,rest
    | _ -> raise Noparse;;
```

Now we can define a parser for terms, in a form very similar to the original grammar. The main difference is that each production rule has associated with it some sort of special action to take as a result of parsing.

```
let rec atom input
      = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
             >> (fun (((name,_),args),_) -> Fn(name,args))
      || name
             >> (fun s -> Var s)
      || numeral
             >> (fun s -> Const s)
      || a (Other "(") ++ term ++ a (Other ")")
             >> (snd o fst)
      || a (Other "-") ++ atom
             >> snd) input
  and mulexp input
      = (atom ++ a(Other "*") ++ mulexp
             >> (fun ((a,_),m) -> Fn("*",[a;m]))
      || atom) input
  and term input
      = (mulexp ++ a(Other "+") ++ term
             >> (fun ((a,_),m) -> Fn("+",[a;m]))
      || mulexp) input
  and termlist input
      = (term ++ a (Other ",") ++ termlist
             >> (fun ((h,_),t) -> h::t)
      || term
             >> (fun h -> [h])) input;;
```

Let us package everything up as a single parsing function:

```
let parser = fst o (term ++ finished >> fst) o lex;;
```

To see it in action, we try with and without the printer (see above) installed:

```
#parser "sin(x + y) * cos(2 * x + y)";;
- : term =
 Fn
  ("*",
   [Fn ("sin", [Fn ("+", [Var "x"; Var "y"])]);
    Fn ("cos", [Fn ("+", [Fn ("*", [Const "2"; Var "x"]);
                          Var "y"])])])
#install_printer "print_term";;
- : unit = ()
#parser "sin(x + y) * cos(2 * x + y)";;
- : term = `sin(x + y) * cos(2 * x + y)`
```

9.2.5 Automatic precedence parsing

In the above parser, we 'hardwired' a parser for the two infix operators. However it would be nicer, and consistent with our approach to printing, if changes to the set of infixes could propagate to the parser. Moreover, even with just two different binary operators, our production rules and consequent parser were already starting to look artificial; imagine if we had a dozen or so. What we would like is to automate the production of a layer of parsers, one for each operator, in order of precedence. This can easily be done. First, we define a general function that takes a parser for what, at this level, are regarded as atomic expressions, and produces a new parser for such expressions separated by a given operator. Note that this could also be defined using parsing combinators, but the function below is simple and more efficient.

```
let rec binop op parser input =
  let atom1,rest1 as result = parser input in
  if not rest1 = [] & hd rest1 = Other op then
    let atom2,rest2 = binop op parser (tl rest1) in
    Fn(op,[atom1; atom2]),rest2
  else result;;
```

Now we define functions that pick out the infix operator in our association list with (equal) lowest precedence, and delete it from the list. If we maintained the list already sorted in precedence order, we could simply take the head and tail of the list.

```
let findmin l = itlist
  (fun (_,pr1 as p1) (_,pr2 as p2) -> if pr1 <= pr2 then p1 else p2)
  (tl l) (hd l);;

let rec delete x (h::t) = if h = x then t else h::(delete x t);;
```

Now we can define the generic precedence parser:

```
let rec precedence ilist parser input =
  if ilist = [] then parser input else
  let opp = findmin ilist in
  let ilist' = delete opp ilist in
  binop (fst opp) (precedence ilist' parser) input;;
```

and hence can modify the main parser to be both simpler and more general:

```
let rec atom input
      = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
             >> (fun (((name,_),args),_) -> Fn(name,args))
      || name
             >> (fun s -> Var s)
      || numeral
             >> (fun s -> Const s)
      || a (Other "(") ++ term ++ a (Other ")")
             >> (snd o fst)
      || a (Other "-") ++ atom
             >> snd) input
  and term input = precedence (!infixes) atom input
  and termlist input
      = (term ++ a (Other ",") ++ termlist
             >> (fun ((h,_),t) -> h::t)
      || term
             >> (fun h -> [h])) input;;

let parser = fst o (term ++ finished >> fst) o lex;;
```

for example:

```
#parser "2 * sin(x)^2 + 2 * sin(y)^2 - 2";;
- : term =
 Fn
  ("+",
   [Fn ("*", [Const "2"; Fn ("^", [Fn ("sin", [Var "x"]);
                                   Const "2"])]);
    Fn
     ("-",
      [Fn ("*", [Const "2"; Fn ("^", [Fn ("sin", [Var "y"]);
                                      Const "2"])]);
       Const "2"])])
#install_printer "print_term";;
- : unit = ()
#parser "2 * sin(x)^2 + 2 * sin(y)^2 - 2";;
- : term = `2 * sin(x) ^ 2 + (2 * sin(y) ^ 2 - 2)`
```

9.2.6 Defects of our approach

Our approach to lexing and parsing is not particularly efficient. In fact, CAML and some other ML implementations include a generator for LR parsers similar to YACC (Yet Another Compiler Compiler) which is often used to generate parsers in C under Unix. Not only are these more general in the grammars that they can handle, but the resulting parsers are more efficient. However we believe the present approach is rather clear, adequate for most purposes, and a good introduction to programming with higher order functions.

The inefficiency of our approach can be reduced by avoiding a few particularly wasteful features. Note that if two different productions for the same syntactic category have a common prefix, then we should avoid parsing it more than once, unless it is guaranteed to be trivial. Our production rules for *term* have this property:

$$\begin{array}{rcl} term & \longrightarrow & name(termlist) \\ & | & name \\ & | & \cdots \end{array}$$

We carefully put the longer production first in our actual implementation, otherwise success in reading a name would cause the abandonment of attempts to read a parenthesized list of arguments. Nevertheless, because of the redundancy we have mentioned, this way of constructing the parser is wasteful. In this case it doesn't matter much, because the duplicated call only analyzes one token. However the case for *termlist* is more significant:

```
  let ...
  and termlist input
       = (term ++ a (Other ",") ++ termlist
              >> (fun ((h,_),t) -> h::t)
       || term
              >> (fun h -> [h])) input;;
```

Here we could reparse a whole term, which might be arbitrarily large. We can avoid this in several ways. One is to move the `||` alternation to after the initial term has been read. A convenient way to code it is to eschew explicit recursion in favour of `many`:

```
  let ...
  and termlist input
      = term ++ many (a (Other ",") ++ term >> snd)
              >> (fun (h,t) -> h::t) input;;
```

Therefore the final version of the parser is:

```
let rec atom input
       = (name ++ a (Other "(") ++ termlist ++ a (Other ")")
              >> (fun (((name,_),args),_) -> Fn(name,args))
       || name
              >> (fun s -> Var s)
       || numeral
              >> (fun s -> Const s)
       || a (Other "(") ++ term ++ a (Other ")")
              >> (snd o fst)
       || a (Other "-") ++ atom
              >> snd) input
  and term input = precedence (!infixes) atom input
  and termlist input
        = (term ++ many (a (Other ",") ++ term >> snd)
              >> (fun (h,t) -> h::t)) input;;
```

A general defect of our approach, and recursive descent parsing generally, is that so-called *left recursion* in productions causes problems. This is where a given category expands to a sequence including the category itself as the first item. For example, if we had wanted to make the addition operator left-associative in our earlier grammar, we could have used:

$$\begin{array}{rcl} term & \longrightarrow & term \; \texttt{+} \; mulexp \\ & | & mulexp \end{array}$$

The naive transcription into ML would loop indefinitely, calling `term` on the same subterm over and over. There are various slightly ad hoc ways of dealing with this problem. For example, one can convert this use of recursion, which in some ways is rather gratuitous, into the explicit use of repetition. We can do this using standard combinators like `many` to return a list of '*mulexp*'s and then run over the list building the tree left-associated.

A final problem is that we do not handle errors very gracefully. We always raise the exception `Noparse`, and this, when caught, initiates backtracking. However this is not always appropriate for typical grammars. At best, it can cause expensive reparsing, and can lead to baffling error indications at the top level. For example, on encountering `5 +`, we expect a term following the `+`, and if we don't find one, we should generate a comprehensible exception for the user, rather than raise `Noparse` and perhaps initiate several other pointless attempts to parse the expression in a different way.

9.3 Exact real arithmetic

Real arithmetic on computers is normally done via floating point approximations. In general, we can only manipulate a real number, either ourselves or inside a computer, via some sort of finite representation. Some question how numbers can be said to 'exist' if they have no finite representation. For example, Kronecker accepted integers and rationals because they can be written down explicitly, and even *algebraic* numbers[3] because they can be represented using the polynomials of which they are solutions. However he rejected transcendental numbers because apparently they could not be represented finitely. Allegedly he he greeted the famous proof by Lindemann (1882) that π is transcendental with the remark that this was 'interesting, except that π does not exist'.

However, given our modern perspective, we can say that after all many more numbers than Kronecker would have accepted *do* have a finite representation, namely the program, or in some more abstract sense the *rule*, used to calculate

[3]Algebraic numbers are those that are roots of polynomials with integer coefficients, e.g. $\sqrt{2}$, and transcendental numbers are ones that aren't.

them to greater and greater precision. For example, we can write a program that will produce for any argument n the first n digits of π. Alternatively it can produce a rational number r such that $|\pi - r| < 2^{-n}$. Whatever approach is taken to the successive approximation of a real number, the key point is that its representation, the program itself, is finite.

This works particularly nicely in a language like ML where higher order functions are available. What we have called a 'program' above will just be an ML function. We can actually represent the arithmetic operations on numbers as higher order functions that, given functions for approximating x and y, will produce new ones for approximating $x + y$, xy, $sin(x)$ and so on, for a wide range of functions. In an ordinary programming language, we would need to define a concrete representation for programs, e.g. 'Gödel numbering', and write an interpreter for this representation.[4]

9.3.1 Representation of real numbers

Our approach is to represent a real x by a function $f_x : \mathbb{N} \to \mathbb{Z}$ that for each $n \in \mathbb{N}$ returns an approximation of x to within 2^{-n}, appropriately scaled. In fact, we have:

$$|f_x(n) - 2^n x| < 1$$

This is of course equivalent to $|\frac{f_x(n)}{2^n} - x| < \frac{1}{2^n}$. We could return a rational approximation directly, but it would still be convenient for the rationals to have denominators that are all powers of some number, so that common denominators arising during addition don't get too big. Performing the scaling directly seems simpler, requiring only integer arithmetic. Our choice of the base 2 is largely arbitrary. A lower base is normally advantageous because it minimizes the *granularity* of the achievable accuracies. For example, if we used base 10, then even if we only need to increase the accuracy of an approximation slightly, we have to step up the requested accuracy by a factor of 10.

9.3.2 Arbitrary-precision integers

CAML's standard integers (type `int`) have a severely limited range, so first of all we need to set up a type of unlimited-precision integers. This is not so difficult to program, but is slightly tedious. Fortunately, the CAML release includes a library that gives a fast implementation of arbitrary precision integer (and in fact rational) arithmetic. A version of CAML Light with this library pre-loaded is installed on Thor. Assuming you have used the following path setting:

[4] Effectively, we are manipulating functions 'in extension' rather than 'in intension', i.e. not concerning ourselves with their representation.

```
PATH="$PATH:/home/jrh13/caml/bin"
export PATH
```

then the augmented version of CAML Light can be fired up using:

```
$ camllight my_little_caml
```

When the system returns its prompt, use the `#open` directive to make the library functions available:

```
##open "num";;
```

This library sets up a new type `num` of arbitrary precision rational numbers; we will just use the integer subset. CAML does not provide overloading, so it is necessary to use different symbols for the usual arithmetic operations over `num`.

Note that small constants of type `num` must be written as `Int k` rather than simply `k`. In fact `Int` is a type constructor for `num` used in the case when the number is a machine-representable integer. Larger ones use `Big_int`.

The unary negation on type `num` is written `minus_num`. Another handy unary operator is `abs_num`, which finds absolute values, i.e. given x returns $|x|$.

The usual binary operations are also available. The (truncating) division and modulus function, called `quo_num` and `mod_num`, do not have infix status. However most of the other binary operators are infixes, and the names are derived from the usual ones by adding a slash '/'. It is important to realize that in general, one must use `=/` to compare numbers for equality. This is because the constructors of the type `num` are, as always, distinct by definition, yet in terms of the underlying meaning they may overlap. For example we get different result from using truncating division and true division on 2^{30} and 2, even though they are numerically the same. One uses type constructor `Int` and the other `Ratio`.

```
#(Int 2 **/ Int 30) // Int 2 = quo_num (Int 2 **/ Int 30) (Int 2);;
it : bool = false
#(Int 2 **/ Int 30) // Int 2 =/ quo_num (Int 2 **/ Int 30) (Int 2);;
it : bool = true
```

Here is a full list of the main infix binary operators:

Operator	Type	Meaning
`**/`	`num -> num -> num`	Exponentiation
`*/`	`num -> num -> num`	Multiplication
`+/`	`num -> num -> num`	Addition
`-/`	`num -> num -> num`	Subtraction
`=/`	`num -> num -> bool`	Equality
`<>/`	`num -> num -> bool`	Inequality
`</`	`num -> num -> bool`	Less than
`<=/`	`num -> num -> bool`	Less than or equal
`>/`	`num -> num -> bool`	Greater than
`>=/`	`num -> num -> bool`	Greater than or equal

Let us see some further examples:

```
#Int 5 */ Int 14;;
it : num = Int 70
#Int 2 **/ Int 30;;
it : num = Big_int <abstr>
#(Int 2 **/ Int 30) // Int 2;;
it : num = Ratio <abstr>
#quo_num (Int 2 **/ Int 30) (Int 2);;
it : num = Int 536870912
```

Note that large numbers are not printed. However we can always convert one to a string using `string_of_num`:

```
#string_of_num(Int 2 ** Int 150);;
- : string = "1427247692705959881058285969449495136382746624"
```

This also has an inverse, called naturally enough `num_of_string`.

9.3.3 Basic operations

Recall that our real numbers are supposed to be (represented by) functions $\mathbb{Z} \to \mathbb{Z}$. In ML we will actually use `int -> num`, since the inbuilt type of integers is more than adequate for indexing the level of accuracy. Now we can define some operations on reals. The most basic operation, which gets us started, is to produce the real number corresponding to an integer. This is easy:

```
#let real_of_int k n = (Int 2 **/ Int n) */ Int k;;
real_of_int : int -> int -> num = <fun>
#real_of_int 23;;
- : int -> num = <fun>
```

It is obvious that for any k this obeys the desired approximation criterion:

$$|f_k(n) - 2^n k| = |2^n k - 2^n k| = 0 < 1$$

Now we can define the first nontrivial operation, that of unary negation:

```
let real_neg f n = minus_num(f n);;
```

The compiler generalizes the type more than intended, but this will not trouble us. It is almost as easy to see that the approximation criterion is preserved. If we know that for each n:

$$|f_x(n) - 2^n x| < 1$$

then we have for any n:

$$\begin{aligned}
|f_{-x}(n) - 2^n(-x)| &= |-f_x(n) - 2^n(-x)| \\
&= |-(f_x(n) - 2^n x)| \\
&= |f_x(n) - 2^n x| \\
&< 1
\end{aligned}$$

Similarly, we can define an 'absolute value' function on real numbers, using the corresponding function abs_num on numbers:

```
let real_abs f n = abs_num (f n);;
```

The correctness of this is again straightforward, using the fact that $||x|-|y|| \leq |x-y|$. Now consider the problem of adding two real numbers. Suppose x and y are represented by f_x and f_y respectively. We could define:

$$f_{x+y}(n) = f_x(n) + f_y(n)$$

However this gives no guarantee that the approximation criterion is maintained; we would have:

$$\begin{aligned}
|f_{x+y}(n) - 2^n(x+y)| &= |f_x(n) + f_y(n) - 2^n(x+y)| \\
&\leq |f_x(n) - 2^n x| + |f_y(n) - 2^n y|
\end{aligned}$$

We can guarantee that the sum on the right is less than 2, but not that it is less than 1 as required. Therefore, we need in this case to evaluate x and y to *greater* accuracy than required in the answer. Suppose we define:

$$f_{x+y}(n) = (f_x(n+1) + f_y(n+1))/2$$

Now we have:

$$\begin{aligned}
|f_{x+y}(n) - 2^n(x+y)| &= |(f_x(n+1) + f_y(n+1))/2 - 2^n(x+y)| \\
&\leq |f_x(n+1)/2 - 2^n x| + |f_y(n+1)/2 - 2^n y| \\
&= \frac{1}{2}|f_x(n+1) - 2^{n+1}x| + \frac{1}{2}|f_y(n+1) - 2^{n+1}y| \\
&< \frac{1}{2}1 + \frac{1}{2}1 = 1
\end{aligned}$$

Apparently this just gives the accuracy required. However we have implicitly used real mathematical division above. Since the function is supposed to yield an integer, we are obliged to round the quotient to an integer. If we just use quo_num, the error from rounding this might be almost 1, after which we could never guarantee the bound we want, however accurately we evaluate the arguments. However with a bit more care we can define a division function that always returns the integer closest to the true result (or one of them in the case of two equally close ones), so that the rounding error never exceeds $\frac{1}{2}$. This could be done directly in integer arithmetic, but the most straightforward coding is to use rational division followed by rounding to the nearest integer, as there are built-in functions for both these operations:

```
#let ndiv x y = round_num(x // y);;
ndiv : num -> num -> num = <fun>
##infix "ndiv";;
#(Int 23) ndiv (Int 5);;
- : num = Int 5
#(Int 22) ndiv (Int 5);;
- : num = Int 4
#(Int(-11)) ndiv (Int 4);;
- : num = Int -3
#(Int(-9)) ndiv (Int 4);;
- : num = Int -2
```

Now if we define:

$$f_{x+y}(n) = (f_x(n+2) + f_y(n+2)) \text{ ndiv } 4$$

everything works:

$$\begin{aligned}
|f_{x+y}(n) - 2^n(x+y)| &= |((f_x(n+2) + f_y(n+2)) \text{ ndiv } 4) - 2^n(x+y)| \\
&\leq \frac{1}{2} + |(f_x(n+2) + f_y(n+2))/4 - 2^n(x+y)| \\
&= \frac{1}{2} + \frac{1}{4}|(f_x(n+2) + f_y(n+2)) - 2^{n+2}(x+y)| \\
&\leq \frac{1}{2} + \frac{1}{4}|f_x(n+2) - 2^{n+2}x| + \frac{1}{4}|f_y(n+2) - 2^{n+2}y| \\
&< \frac{1}{2} + \frac{1}{4}1 + \frac{1}{4}1 \\
&= 1
\end{aligned}$$

Accordingly we make our definition:

```
let real_add f g n =
  (f(n + 2) +/ g(n + 2)) ndiv (Int 4);;
```

We can define subtraction similarly, but the simplest approach is to build it out of the functions that we already have:

```
#let real_sub f g = real_add f (real_neg g);;
real_sub : (num -> num) -> (num -> num) -> num -> num = <fun>
```

It is a bit trickier to define multiplication, inverses and division. However the cases where we multiply or divide by an integer are much easier and quite common. It is worthwhile implementing these separately as they can be made more efficient. We define:

$$f_{mx}(n) = (m f_x(n+p+1)) \text{ ndiv } 2^{p+1}$$

where p is chosen so that $2^p \geq |m|$. For correctness, we have:

$$\begin{aligned}
|f_{mx}(n) - 2^n(mx)| &\leq \frac{1}{2} + \left|\frac{m f_x(n+p+1)}{2^{p+1}} - 2^n(mx)\right| \\
&= \frac{1}{2} + \frac{|m|}{2^{p+1}}|f_x(n+p+1) - 2^{n+p+1}x| \\
&< \frac{1}{2} + \frac{|m|}{2^{p+1}} \\
&\leq \frac{1}{2} + \frac{1}{2}\frac{|m|}{2^p} \\
&\leq \frac{1}{2} + \frac{1}{2} = 1
\end{aligned}$$

In order to implement this, we need a function to find the appropriate p. The following is crude but adequate:

```
let log2 =
  let rec log2 x y =
    if x </ Int 1 then y
    else log2 (quo_num x (Int 2)) (y + 1) in
  fun x -> log2 (x -/ Int 1) 0;;
```

The implementation is simply:

```
let real_intmul m x n =
   let p = log2 (abs_num m) in
   let p1 = p + 1 in
   (m */ x(n + p1)) ndiv (Int 2 **/ Int p1);;
```

For division by an integer, we define:

$$f_{x/m}(n) = f_x(n) \text{ ndiv } m$$

For correctness, we can ignore the trivial cases when $m = 0$, which should never be used, and when $m = \pm 1$, since then the result is exact. Otherwise, we assume $|f_x(n) - 2^n x| < 1$, so $|f_x(n)/m - 2^n x/m| < \frac{1}{|m|} \leq \frac{1}{2}$, which, together with the fact that $|f_x(n) \text{ ndiv } m - f_x(n)/m| \leq \frac{1}{2}$, yields the result. So we have:

```
let real_intdiv m x n =
   x(n) ndiv (Int m);;
```

9.3.4 General multiplication

General multiplication is harder because the error in approximation for one number is multiplied by the magnitude of the second number. Therefore, before the final accuracies are calculated, a preliminary evaluation of the arguments is required to determine their approximate magnitude. We proceed as follows. Suppose that we want to evaluate $x + y$ to precision n. First we choose r and s so that $|r - s| \leq 1$ and $r + s = n + 2$. That is, both r and s are slightly more than half the required precision. We now evaluate $f_x(r)$ and $f_y(s)$, and select natural numbers p and q that are the corresponding 'binary logarithms', i.e. $|f_x(r)| \leq 2^p$ and $|f_y(s)| \leq 2^q$. If both p and q are zero, then it is easy to see that we may return just 0. Otherwise, remember that either $p > 0$ or $q > 0$ as we will need this later. Now set:

$$\begin{aligned}
k &= n + q - s + 3 = q + r + 1 \\
l &= n + p - r + 3 = p + s + 1 \\
m &= (k + l) - n = p + q + 4
\end{aligned}$$

We claim that $f_{xy}(n) = (f_x(k)f_y(l)) \text{ ndiv } 2^m$ has the right error behaviour, i.e. $|f_{xy}(n) - 2^n(xy)| < 1$. If we write:

$$\begin{aligned}
2^k x &= f_x(k) + \delta \\
2^l y &= f_y(l) + \epsilon
\end{aligned}$$

with $|\delta| < 1$ and $|\epsilon| < 1$, we have:

$$\begin{aligned}
|f_{xy}(n) - 2^n(xy)| &\leq \frac{1}{2} + |\frac{f_x(k)f_y(l)}{2^m} - 2^n(xy)| \\
&= \frac{1}{2} + 2^{-m}|f_x(k)f_y(l) - 2^{k+l}xy| \\
&= \frac{1}{2} + 2^{-m}|f_x(k)f_y(l) - (f_x(k) + \delta)(f_y(l) + \epsilon)| \\
&= \frac{1}{2} + 2^{-m}|\delta f_y(l) + \epsilon f_x(k) + \delta\epsilon| \\
&\leq \frac{1}{2} + 2^{-m}(|\delta f_y(l)| + |\epsilon f_x(k)| + |\delta\epsilon|) \\
&\leq \frac{1}{2} + 2^{-m}(|f_y(l)| + |f_x(k)| + |\delta\epsilon|) \\
&< \frac{1}{2} + 2^{-m}(|f_y(l)| + |f_x(k)| + 1)
\end{aligned}$$

Now we have $|f_x(r)| \leq 2^p$, so $|2^r x| < 2^p + 1$. Thus $|2^k x| < 2^{q+1}(2^p + 1)$, so $|f_x(k)| < 2^{q+1}(2^p + 1) + 1$, i.e. $|f_x(k)| \leq 2^{q+1}(2^p + 1)$. Similarly $|f_y(l)| \leq 2^{p+1}(2^q + 1)$. Consequently:

$$\begin{aligned} |f_y(l)| + |f_x(k)| + 1 &\leq 2^{p+1}(2^q+1) + 2^{q+1}(2^p+1) + 1 \\ &= 2^{p+q+1} + 2^{p+1} + 2^{p+q+1} + 2^{q+1} + 1 \\ &= 2^{p+q+2} + 2^{p+1} + 2^{q+1} + 1 \end{aligned}$$

Now for our error bound we require $|f_y(l)| + |f_x(k)| + 1 \leq 2^{m-1}$, or dividing by 2 and using the discreteness of the integers:

$$2^{p+q+1} + 2^p + 2^q < 2^{p+q+2}$$

We can write this as $(2^{p+q} + 2^p) + (2^{p+q} + 2^q) < 2^{p+q+1} + 2^{p+q+1}$, which is true because we have either $p > 0$ or $q > 0$. So at last we are justified in defining:

```
let real_mul x y n =
  let n2 = n + 2 in
  let r = n2 / 2 in
  let s = n2 - r in
  let xr = x(r)
  and ys = y(s) in
  let p = log2 xr
  and q = log2 ys in
  if p = 0 & q = 0 then Int 0 else
  let k = q + r + 1
  and l = p + s + 1
  and m = p + q + 4 in
  (x(k) */ y(l)) ndiv (Int 2 **/ Int m);;
```

9.3.5 Multiplicative inverse

Next we will define the multiplicative inverse function. In order to get any sort of upper bound on this, let alone a good approximation, we need to get a *lower* bound for the argument. In general, there is no better way than to keep evaluating the argument with greater and greater accuracy until we can bound it away from zero. We will use the following lemma to justify the correctness of our procedure:

Lemma 9.1 *If $2e \geq n + k + 1$, $|f_x(k)| \geq 2^e$ and $|f_x(k) - 2^k x| < 1$, where $f_x(k)$ is an integer and e, n and k are natural numbers, then if we define*

$$f_y(n) = 2^{n+k} \ ndiv \ f_x(k)$$

we have $|f_y(n) - 2^n x^{-1}| < 1$, i.e. the required bound.

Proof: *The proof is rather tedious and will not be given in full. We just sketch the necessary case splits. If $|f_x(k)| > 2^e$ then a straightforward analysis gives the result; the rounding in* `ndiv` *gives an error of at most $\frac{1}{2}$, and the remaining error is $< \frac{1}{2}$. If $|f_x(k)| = 2^e$ but $n + k \geq e$, then although the second component of the* *error may now be twice as much, i.e.* < 1*, there is no rounding error because* $f_x(k) = \pm 2^e$ *divides into* 2^{n+k} *exactly. (We use here the fact that* $2^e - 1 \leq 2^{e-1}$*, because since* $2e \geq n + k + 1$*,* e *cannot be zero.) Finally, if* $|f_x(k)| = 2^e$ *and* $n + k < e$*, we have* $|f_y(n) - 2^n \frac{1}{x}| < 1$ *because both* $|f_y(n)| \leq 1$ *and* $0 < |2^n \frac{1}{x}| < 1$*, and both these numbers have the same sign. Q.E.D.*

Now suppose we wish to find the inverse of x to accuracy n. First we evaluate $f_x(0)$. There are two cases to distinguish:

1. If $|f_x(0)| > 2^r$ for some natural number r, then choose the least natural number k (which may well be zero) such that $2r + k \geq n + 1$, and set $e = r + k$. We now return 2^{n+k} ndiv $f_x(k)$. It is easy to see that the conditions of the lemma are satisfied. Since $|f_x(0)| \geq 2^r + 1$ we have $|x| > 2^r$, and so $|2^k x| > 2^{r+k}$. This means $|f_x(k)| > 2^{r+k} - 1$, and as $f_x(k)$ is an integer, $|f_x(k)| \geq 2^{r+k} = 2^e$ as required. The condition that $2e \geq n = k + 1$ is easy to check. Note that if $r \geq n$ we can immediately deduce that $f_y(n) = 0$ is a valid approximation.

2. If $|f_x(0)| \leq 1$, then we call the function '`msd`' that returns the least p such that $|f_x(p)| > 1$. Note that this may in general fail to terminate if $x = 0$. Now we set $e = n + p + 1$ and $k = e + p$, and return 2^{n+k} ndiv $f_x(k)$. Once again the conditions for the lemma are satisfied. Since $|f_x(p)| \geq 2$, we have $|2^p x| > 1$, i.e. $|x| > \frac{1}{2^p}$. Hence $|2^k x| > 2^{k-p} = 2^e$, and so $|f_x(k)| > 2^e - 1$, i.e. $|f_x(k)| \geq 2^e$.

To implement this, we first define the `msd` function:

```
let msd =
  let rec msd n x =
    if abs_num(x(n)) >/ Int 1 then n else msd (n + 1) x in
  msd 0;;
```

and then translate the above mathematics into a simple program:

```
let real_inv x n =
  let x0 = x(0) in
  let k =
    if x0 >/ Int 1 then
      let r = log2 x0 - 1 in
      let k0 = n + 1 - 2 * r in
      if k0 < 0 then 0 else k0
    else
      let p = msd x in
      n + 2 * p + 1 in
  (Int 2 **/ Int (n + k)) ndiv (x(k));;
```

Now, of course, it is straightforward to define division:

```
let real_div x y = real_mul x (real_inv y);;
```

9.3.6 Ordering relations

The interesting things about all these functions is that they are in fact uncomputable in general. The essential point is that it is impossible to decide whether a given number is zero. We can keep approximating it to greater and greater accuracy, but if the approximations always return 0, we cannot tell whether it is going to do so indefinitely or whether the next step will give a nonzero answer.[5] If x is not zero, then the search will eventually terminate, but if x is zero, it will run forever.

Accepting this, it is not difficult to define the relational operators. To decide the ordering relation of x and y it suffices to find an n such that $|x_n - y_n| \geq 2$. For example, if $x_n \geq y_n + 2$ we have

$$2^n x > x_n - 1 \geq y_n + 1 > 2^n y$$

and so $x > y$. We first write a general procedure to perform this search, and then define all the orderings in terms of it. Note that the only way to arrive at the reflexive orderings is to justify the irreflexive version!

```
let separate =
  let rec separate n x y =
    let d = x(n) -/ y(n) in
    if abs_num(d) >/ Int 1 then d
    else separate (n + 1) x y in
separate 0;;

let real_gt x y = separate x y >/ Int 0;;
let real_ge x y = real_gt x y;;
let real_lt x y = separate x y </ Int 0;;
let real_le x y = real_lt x y;;
```

9.3.7 Caching

In order to try out all the above functions, we would like to be able to print out a decimal representation of some approximation to a real. This is not hard, using some more standard facilities in the CAML library. If d decimal places are desired, then we make n such that $2^n > 10^d$ and hence the accuracy corresponds at least to the number of digits printed.

```
let view x d =
  let n = 4 * d in
  let out = x(n) // (Int 2 **/ Int n) in
  approx_num_fix d out;;
```

[5] For a proof that it is uncomputable, simply wrap up the halting problem, by defining $f(n) = 1$ if a certain Turing machine has halted after no more than n iterations, and $f(n) = 0$ otherwise.

Now we can test out some simple examples, which seem to work:

```
#let x = real_of_int 3;;
x : int -> num = <fun>
#let xi = real_inv x;;
xi : int -> num = <fun>
#let wun = real_mul x xi;;
wun : int -> num = <fun>
#view x 20;;
it : string = "3.00000000000000000000"
#view xi 20;;
it : string = ".33333333333333333333"
#view wun 20;;
it : string = "1.00000000000000000000"
```

However there is a subtle and serious bug in our implementation, which shows itself if we try larger expressions. The problem is that subexpressions can be evaluated many times. Most obviously, this happens if they occur several times in the input expression. But even if they don't, the multiplication, and even more so, the inverse, require trial evaluations at differing accuracies. As the evaluation propagates down to the bottom level, there is often an exponential buildup of reevaluations. For example, the following is slow — it takes several seconds.

```
#let x1 = real_of_int 1 in
 let x2 = real_mul x1 x1 in
 let x3 = real_mul x2 x2 in
 let x4 = real_mul x3 x3 in
 let x5 = real_mul x4 x4 in
 let x6 = real_mul x5 x5 in
 let x7 = real_mul x6 x6 in
 view x7 10;;
 - : string = "+1.0000000000"
```

We can fix this problem using the idea of *caching* or *memo functions* (Michie 1968). We give each function a reference cell to remember the most accurate version already calculated. If there is a second request for the same accuracy, it can be returned immediately without further evaluation. What's more, we can always construct a lower-level approximation, say n, from a higher one, say $n+k$ with $k \geq 1$. If we know that $|f_x(n+k) - 2^{n+k}x| < 1$ then we have:

$$\begin{aligned}
|f_x(n+k) \text{ ndiv } 2^k - 2^n x| &\leq \frac{1}{2} + |\frac{f_x(n+k)}{2^k} - 2^n x| \\
&= \frac{1}{2} + \frac{1}{2^k}|f_x(n+k) - 2^{n+k}x| \\
&< \frac{1}{2} + \frac{1}{2^k} \\
&\leq 1
\end{aligned}$$

Hence we are always safe in returning $f_x(n+k)$ ndiv 2^k. We can implement this memo technique via a generic function `memo` to be inserted in each of the previous functions:

```
let real_of_int k = memo (fun n -> (Int 2 **/ Int n) */ Int k);;

let real_neg f = memo (fun n -> minus_num(f n));;

let real_abs f = memo (fun n -> abs_num (f n));;

let real_add f g = memo (fun n ->
   (f(n + 2) +/ g(n + 2)) ndiv (Int 4));;

let real_sub f g = real_add f (real_neg g);;

let real_intmul m x = memo (fun n ->
   let p = log2 (abs_num m) in
   let p1 = p + 1 in
   (m */ x(n + p1)) ndiv (Int 2 **/ Int p1));;

let real_intdiv m x = memo (fun n ->
   x(n) ndiv (Int m));;

let real_mul x y = memo (fun n ->
  let n2 = n + 2 in
  let r = n2 / 2 in
  let s = n2 - r in
  let xr = x(r)
  and ys = y(s) in
  let p = log2 xr
  and q = log2 ys in
  if p = 0 & q = 0 then Int 0 else
  let k = q + r + 1
  and l = p + s + 1
  and m = p + q + 4 in
  (x(k) */ y(l)) ndiv (Int 2 **/ Int m));;

let real_inv x = memo (fun n ->
  let x0 = x(0) in
  let k =
    if x0 >/ Int 1 then
      let r = log2 x0 - 1 in
      let k0 = n + 1 - 2 * r in
      if k0 < 0 then 0 else k0
    else
      let p = msd x in
      n + 2 * p + 1 in
  (Int 2 **/ Int (n + k)) ndiv (x(k)));;

let real_div x y = real_mul x (real_inv y);;
```

where

```
let memo f =
  let mem = ref (-1,Int 0) in
  fun n -> let (m,res) = !mem in
           if n <= m then
             if m = n then res
             else res ndiv (Int 2 **/ Int(m - n))
           else
             let res = f n in
             mem := (n,res); res;;
```

Now the above sequence of multiplications is instantaneous. Here are a few more examples:

```
#let pi1 = real_div (real_of_int 22) (real_of_int 7);;
pi1 : int -> num = <fun>
#view pi1 10;;
it : string = "+3.1428571429"
#let pi2 = real_div (real_of_int 355) (real_of_int 113);;
pi2 : int -> num = <fun>
#view pi2 10;;
it : string = "+3.1415929204"
#let pidiff = real_sub pi1 pi2;;
pidiff : int -> num = <fun>
#view pidiff 20;;
it : string = "+0.00126422250316055626"
#let ipidiff = real_inv pidiff;;
ipidiff : int -> num = <fun>
#view ipidiff 20;;
it : string = "+791.00000000000000000000"
```

Of course, everything we have done to date could be done with rational arithmetic. Actually, it may happen that our approach is more efficient in certain situations since we avoid calculating numbers that might have immense numerators and denominators when we just need an approximation. Nevertheless, the present approach really comes into its own when we want to define transcendental functions like *exp*, *sin* etc. We will not cover this in detail for lack of space, but it makes an interesting exercise. One approach is to use truncated Taylor series. Note that finite summations can be evaluated directly rather than by iterating the addition function; this leads to much better error behaviour.

9.4 Prolog and theorem proving

The language Prolog is popular in Artificial Intelligence research, and is used in various practical applications such as expert systems and intelligent databases. Here we will show how the main mechanism of Prolog, namely depth-first search through a database of rules using unification and backtracking, can be implemented in ML. We do not pretend that this is a full-blown implementation of

Prolog, but it gives an accurate picture of the general flavour of the language, and we will be able to run some simple examples in our system.

9.4.1 Prolog terms

Prolog code and data is represented using a uniform system of first order terms. We have already defined a type of terms for our mathematical expressions, and associated parsers and printers. Here we will use something similar, but not quite the same. First of all, it simplifies some of the code if we treat constants as nullary functions, i.e. functions that take an empty list of arguments. Accordingly we define:

```
  type term = Var of string
            | Fn of string * (term list);;
```

Where we would formerly have used `Const s`, we will now use `Fn(s,[])`. Note that we will treat functions of different arities (different numbers of arguments) as distinct, even if they have the same name. Thus there is no danger of our confusing constants with true functions.

9.4.2 Lexical analysis

In order to follow Prolog conventions, which include case sensitivity, we also modify lexical analysis. We will not attempt to conform exactly to the full details of Prolog, but one feature is very important: alphanumeric identifiers that begin with an *upper case* letter or an underscore are treated as variables, while other alphanumeric identifiers, along with numerals, are treated as constants. For example `X` and `Answer` are variables while `x` and `john` are constants. We will lump all symbolic identifiers together as constants too, but we will distinguish the punctuation symbols: left and right brackets, commas and semicolons. Non-punctuation symbols are collected together into the longest strings possible, so symbolic identifiers need not consist only of one character.

```
  type token = Variable of string
             | Constant of string
             | Punct of string;;
```

The lexer therefore looks like this:

```
let lex =
  let several p = many (some p) in
  let collect(h,t) = h^(itlist (prefix ^) t "") in
  let upper_alpha s = "A" <= s & s <= "Z" or s = "_"
  and lower_alpha s = "a" <= s & s <= "z" or "0" <= s & s <= "9"
  and punct s = s = "(" or s = ")" or s = "[" or s = "]"
                or s = "," or s = "."
  and space s = s = " " or s = "\n" or s = "\t" in
  let alphanumeric s = upper_alpha s or lower_alpha s in
  let symbolic s = not space s & not alphanumeric s & not punct s in
  let rawvariable =
    some upper_alpha ++ several alphanumeric >> (Variable o collect)
  and rawconstant =
    (some lower_alpha ++ several alphanumeric ||
     some symbolic ++ several symbolic) >> (Constant o collect)
  and rawpunct = some punct >>  Punct in
  let token =
    (rawvariable || rawconstant || rawpunct) ++
    several space >> fst in
  let tokens = (several space ++ many token) >> snd in
  let alltokens = (tokens ++ finished) >> fst in
  fst o alltokens o explode;;
```

For example:

```
#lex "add(X,Y,Z) :- X is Y+Z.";;
- : token list =
 [Constant "add"; Punct "("; Variable "X"; Punct ",";
  Variable "Y"; Punct ","; Variable "Z"; Punct ")";
  Constant ":-"; Variable "X"; Constant "is"; Variable "Y";
  Constant "+"; Variable "Z"; Punct "."]
```

9.4.3 Parsing

The basic parser is pretty much the same as before; the printer is exactly the same. The main modification to the printer is that we allow Prolog lists to be written in a more convenient notation. Prolog has '.' and 'nil' corresponding to ML's '`::`' and '`[]`', and we set up the parser so that lists can be written much as in ML, e.g. '`[1,2,3]`'. We also allow the Prolog notation '`[H|T]`' instead of '`cons(H,T)`', typically used for pattern-matching. After the basic functions:

```
let variable =
  fun (Variable s::rest) -> s,rest
    | _ -> raise Noparse;;

let constant =
  fun (Constant s::rest) -> s,rest
    | _ -> raise Noparse;;
```

we have a parser for terms and also for Prolog rules, of these forms:

$$term.$$
$$term \quad \text{:-} \quad term, \ldots, term.$$

The parsers are as follows:

```
let rec atom input
      = (constant ++ a (Punct "(") ++ termlist ++ a (Punct ")")
             >> (fun (((name,_),args),_) -> Fn(name,args))
      || constant
             >> (fun s -> Fn(s,[]))
      || variable
             >> (fun s -> Var s)
      || a (Punct "(") ++ term ++ a (Punct ")")
             >> (snd o fst)
      || a (Punct "[") ++ list
             >> snd) input
  and term input = precedence (!infixes) atom input
  and termlist input
      = (term ++ a (Punct ",") ++ termlist
             >> (fun ((h,_),t) -> h::t)
      || term
             >> (fun h -> [h])) input
  and list input
      = (term ++ (a (Constant "|") ++ term ++ a (Punct "]")
                     >> (snd o fst)
                 || a (Punct ",") ++ list
                      >> snd
                 || a (Punct "]")
                     >> (K (Fn("[]",[]))))
             >> (fun (h,t) -> Fn(".",[h; t]))
      || a (Punct "]")
             >> (K (Fn("[]",[])))) input
  and rule input
     = (term ++ (a (Punct ".")
                     >> (K [])
                || a (Constant ":-") ++ term ++
                   many (a (Punct ",") ++ term >> snd) ++
                   a (Punct ".")
                     >> (fun (((_,h),t),_) -> h::t))) input;;

let parse_term = fst o (term ++ finished >> fst) o lex;;

let parse_rules = fst o (many rule ++ finished >> fst) o lex;;
```

9.4.4 Unification

Prolog uses a set of rules to solve a current *goal* by trying to match one of the rules against the goal. A rule consisting of a single term can solve a goal immediately.

In the case of a rule *term* :- $term_1, \ldots, term_n$., if the goal matches *term*, then Prolog needs to solve each $term_i$ as a subgoal in order to finish the original goal.

However, goals and rules do not have to be exactly the same. Instead, Prolog assigns variables in both the goal and the rule to make them match up, a process known as *unification*. This means that we can end up proving a special case of the original goal, e.g. $P(f(X))$ instead of $P(Y)$. For example:

- To unify $f(g(X), Y)$ and $f(g(a), X)$ we can set $X = a$ and $Y = a$. Then both terms are $f(g(a), a)$.
- To unify $f(a, X, Y)$ and $f(X, a, Z)$ we can set $X = a$ and $Y = Z$, and then both terms are $f(a, a, Z)$.
- It is impossible to unify $f(X)$ and X.

In general, unifiers are not unique. For example, in the second example we could choose to set $Y = f(b)$ and $Z = f(b)$. However one can always choose one that is *most general*, i.e. any other unification can be reached from it by further instantiation (compare most general types in ML). In order to find it, roughly speaking, one need only descend the two terms recursively in parallel, and on finding a variable on either side, assigns it to whatever the term on the other side is. One also needs to check that the variable hasn't already been assigned to something else, and that it doesn't occur in the term being assigned to it (as in the last example above). A simple implementation of this idea follows. We maintain an association list giving instantiations already made, and we look each variable up to see if it is already assigned before proceeding. We use the existing instantiations as an accumulator.

```
let rec unify tm1 tm2 insts =
  match tm1 with
    Var(x) ->
      (try let tm1' = assoc x insts in
           unify tm1' tm2 insts
       with Not_found ->
           augment (x,tm2) insts)
  | Fn(f1,args1) ->
      match tm2 with
        Var(y) ->
          (try let tm2' = assoc y insts in
               unify tm1 tm2' insts
           with Not_found ->
               augment (y,tm1) insts)
      | Fn(f2,args2) ->
          if f1 = f2
          then itlist2 unify args1 args2 insts
          else raise (error "functions do not match");;
```

where the instantiation lists need to be augmented with some care to avoid the so-called 'occurs check' problem. We must disallow instantiation of X to a non-trivial term involving X, as in the third example above. Most real Prologs ignore this, either for (claimed) efficiency reasons or in order to allow weird cyclic datastructures instead of simple first order terms.

```
let rec occurs_in x =
  fun (Var y) -> x = y
    | (Fn(_,args)) -> exists (occurs_in x) args;;

let rec subst insts = fun
   (Var y) -> (try assoc y insts with Not_found -> tm)
 | (Fn(f,args)) -> Fn(f,map (subst insts) args);;

let raw_augment =
  let augment1 theta (x,s) =
    let s' = subst theta s in
    if occurs_in x s & not(s = Var(x))
    then raise (error "Occurs check")
    else (x,s') in
  fun p insts -> p::(map (augment1 [p]) insts);;

let augment (v,t) insts =
  let t' = subst insts t in match t' with
    Var(w) -> if w <= v then
                if w = v then insts
                else raw_augment (v,t') insts
              else raw_augment (w,Var(v)) insts
  | _ -> if occurs_in v t'
         then raise (error "Occurs check")
         else raw_augment (v,t') insts;;
```

9.4.5 Backtracking

Prolog proceeds by depth-first search, but it may backtrack: even if a rule unifies successfully, if all the remaining goals cannot be solved under the resulting instantiations, then another initial rule is tried. Thus we consider the whole list of goals rather than one at a time, to give the right control strategy.

```
let rec first f =
  fun [] -> raise (error "No rules applicable")
    | (h::t) -> try f h with error _ -> first f t;;

let rec expand n rules insts goals =
  first (fun rule ->
    if goals = [] then insts else
    let conc,asms =
      rename_rule (string_of_int n) rule in
    let insts' = unify conc (hd goals) insts in
    let local,global = partition
      (fun (v,_) -> occurs_in v conc or
              exists (occurs_in v) asms) insts' in
    let goals' = (map (subst local) asms) @
                 (tl goals) in
    expand (n + 1) rules global goals') rules;;
```

Here we use a function 'rename' to generate fresh variables for the rules each time:

```
let rec rename s =
  fun (Var v) -> Var("~"^v^s)
    | (Fn(f,args)) -> Fn(f,map (rename s) args);;

let rename_rule s (conc,asms) =
  (rename s conc,map (rename s) asms);;
```

Finally, we can package everything together in a function `prolog` that tries to solve a given goal using the given rules:

```
type outcome = No | Yes of (string * term) list;;

let prolog rules goal =
  try let insts = expand 0 rules [] [goal] in
      Yes(filter (fun (v,_) -> occurs_in v goal)
                 insts)
  with error _ -> No;;
```

This says either that the goal cannot be solved, or that it can be solved with the given instantiations. Note that we only return one answer in this latter case, but this is easy to change if desired.

9.4.6 Examples

We can use the Prolog interpreter just written to try some simple examples from Prolog textbooks. For example:

```
#let rules = parse_rules
  "male(albert).
   male(edward).
   female(alice).
   female(victoria).
   parents(edward,victoria,albert).
   parents(alice,victoria,albert).
   sister_of(X,Y) :-
     female(X),
     parents(X,M,F),
     parents(Y,M,F).";;
rules : (term * term list) list =
 [`male(albert)`, []; `male(edward)`, [];
  `female(alice)`, []; `female(victoria)`, [];
  `parents(edward,victoria,albert)`, [];
  `parents(alice,victoria,albert)`, [];
  `sister_of(X,Y)`,
    [`female(X)`; `parents(X,M,F)`; `parents(Y,M,F)`]]
#prolog rules ("sister_of(alice,edward)");;
- : outcome = Yes []
#prolog rules (parse_term "sister_of(alice,X)");;
- : outcome = Yes ["X", `edward`]
#prolog rules (parse_term "sister_of(X,Y)");;
- : outcome = Yes ["Y", `edward`; "X", `alice`]
```

The following are similar to some elementary ML list operations. Since Prolog is relational rather than functional, it is possible to use Prolog queries in a more flexible way, e.g. ask what arguments would give a certain result, rather than vice versa:

```
#let r = parse_rules
  "append([],L,L).
   append([H|T],L,[H|A]) :- append(T,L,A).";;
r : (term * term list) list =
 [`append([],L,L)`, [];
  `append(H . T,L,H . A)`, [`append(T,L,A)`]]
#prolog r (parse_term "append([1,2],[3],[1,2,3])");;
- : outcome = Yes []
#prolog r (parse_term "append([1,2],[3,4],X)");;
- : outcome = Yes ["X", `1 . (2 . (3 . (4 . [])))`]
#prolog r (parse_term "append([3,4],X,X)");;
- : outcome = No
#prolog r (parse_term "append([1,2],X,Y)");;
- : outcome = Yes ["Y", `1 . (2 . X)`]
```

In such cases, Prolog seems to be showing an impressive degree of intelligence. However under the surface it is just using a simple search strategy, and this can easily be thwarted. For example, the following loops indefinitely:

```
#prolog r (parse_term "append(X,[3,4],X)");;
```

9.4.7 Theorem proving

Prolog is acting as a simple theorem prover, using a database of logical facts (the rules) in order to prove a goal. However it is rather limited in the facts it can prove, partly because its depth-first search strategy is incomplete, and partly because it can only make logical deductions in certain patterns. It is possible to make Prolog-like systems that are more powerful, e.g. see Stickel (1988). In what follows, we will just show how to build a more capable theorem prover using essentially similar technology, including the unification code and an identical backtracking strategy.

In particular, unification is an effective way of deciding how to specialize universally quantified variables. For example, given the facts that $\forall X.\, p(X) \Rightarrow q(X)$ and $p(f(a))$, we can unify the two expressions involving p and thus discover that we need to set X to $f(a)$. By contrast, the very earliest theorem provers tried all possible terms built up from the available constants and functions (the 'Herbrand base').

Usually, depth-first search would go into an infinite loop, so we need to modify the Prolog strategy slightly. We will use *depth first iterative deepening*. This means that the search depth has a hard limit, and attempts to exceed it cause backtracking. However, if no proof is found at a given depth, the bound is increased and another attempt is made. Thus, first one searches for proofs of depth 1, and if that fails, searches for one of depth 2, then depth 3 and so on. For 'depth' one can use various parameters, e.g. the height or overall size of the search tree; we will use the number of unifiable variables introduced.

Manipulating formulas

We will simply use our standard first order terms to denote formulas, introducing new constants for the logical operators. Many of these are written infix.

Operator	Meaning
`~(p)`	not p
`p & q`	p and q
`p | q`	p or q
`p --> q`	p implies q
`p <-> q`	p if and only if q
forall(X,p)	for all X, p
exists(X,p)	there exists an X such that p

An alternative would be to introduce a separate type of formulas, but this would require separate parsing and printing support. We will avoid this, for the sake of simplicity.

Preprocessing formulas

It's convenient if the main part of the prover need not cope with implications and 'if and only if's. Therefore we first define a function that eliminates these in favour of the other connectives.

```
let rec proc tm =
  match tm with
    Fn("~",[t]) -> Fn("~",[proc t])
  | Fn("&",[t1; t2]) -> Fn("&",[proc t1; proc t2])
  | Fn("|",[t1; t2]) -> Fn("|",[proc t1; proc t2])
  | Fn("-->",[t1; t2]) ->
        proc (Fn("|",[Fn("~",[t1]); t2]))
  | Fn("<->",[t1; t2]) ->
        proc (Fn("&",[Fn("-->",[t1; t2]);
                      Fn("-->",[t2; t1])]))
  | Fn("forall",[x; t]) -> Fn("forall",[x; proc t])
  | Fn("exists",[x; t]) -> Fn("exists",[x; proc t])
  | t -> t;;
```

The next step is to push the negations down the formula, putting it into so-called 'negation normal form' (NNF). We define two mutually recursive functions that create NNF for a formula, and its negation.

```
let rec nnf_p tm =
  match tm with
    Fn("~",[t]) -> nnf_n t
  | Fn("&",[t1; t2]) -> Fn("&",[nnf_p t1; nnf_p t2])
  | Fn("|",[t1; t2]) -> Fn("|",[nnf_p t1; nnf_p t2])
  | Fn("forall",[x; t]) -> Fn("forall",[x; nnf_p t])
  | Fn("exists",[x; t]) -> Fn("exists",[x; nnf_p t])
  | t -> t

and nnf_n tm =
  match tm with
    Fn("~",[t]) -> nnf_p t
  | Fn("&",[t1; t2]) -> Fn("|",[nnf_n t1; nnf_n t2])
  | Fn("|",[t1; t2]) -> Fn("&",[nnf_n t1; nnf_n t2])
  | Fn("forall",[x; t]) -> Fn("exists",[x; nnf_n t])
  | Fn("exists",[x; t]) -> Fn("forall",[x; nnf_n t])
  | t -> Fn("~",[t]);;
```

We will convert the negation of the input formula into negation normal form, and the main prover will then try to derive a contradiction from it. This will suffice to prove the original formula.

The main prover

At each stage, the prover has a current formula, a list of formulas yet to be considered, and a list of literals. It is trying to reach a contradiction. The following strategy is employed:

- If the current formula is 'p & q', then consider 'p' and 'q' separately, i.e. make 'p' the current formula and add 'q' to the formulas to be considered.
- If the current formula is 'p | q', then try to get a contradiction from 'p' and then one from 'q'.
- If the current formula is 'forall(X,p)', invent a new variable to replace 'X': the right value can be discovered later by unification.
- If the current formula is 'exists(X,p)', invent a new constant to replace 'X'.
- Otherwise, the formula must be a literal, so try to unify it with a contradictory literal.
- If this fails, add it to the list of literals, and proceed with the next formula.

We desire a similar backtracking strategy to Prolog: only if the current instantiation allow all remaining goals to be solved do we accept it. We could use lists again, but instead we use *continuations*. A continuation is a function that is passed to another function and can be called from within it to 'perform the rest of the computation'. In our case, it takes a list of instantiations and tries to solve the remaining goals under these instantiations. Thus, rather than explicitly trying to solve all remaining goals, we simply try calling the continuation.

```
let rec prove fm unexp pl nl n cont i =
  if n < 0 then raise (error "No proof") else
  match fm with
    Fn("&",[p;q]) ->
        prove p (q::unexp) pl nl n cont i
  | Fn("|",[p;q]) ->
        prove p unexp pl nl n
        (prove q unexp pl nl n cont) i
  | Fn("forall",[Var x; p]) ->
        let v = mkvar() in
        prove (subst [x,Var v] p) (unexp@[fm]) pl nl (n - 1) cont i
  | Fn("exists",[Var x; p]) ->
        let v = mkvar() in
        prove (subst [x,Fn(v,[])] p) unexp pl nl (n - 1) cont i
  | Fn("~",[t]) ->
        (try first (fun t' -> let i' = unify t t' i in
                              cont i') pl
         with error _ ->
            prove (hd unexp) (tl unexp) pl (t::nl) n cont i)
  | t ->
        (try first (fun t' -> let i' = unify t t' i in
                              cont i') nl
         with error _ ->
            prove (hd unexp) (tl unexp) (t::pl) nl n cont i);;
```

We set up the final prover as follows:

```
let prover =
  let rec prove_iter n t =
    try let insts = prove t [] [] [] n I [] in
        let globinsts = filter
          (fun (v,_) -> occurs_in v t) insts in
        n,globinsts
    with error _ -> prove_iter (n + 1) t in
  fun t -> prove_iter 0 (nnf_n(proc(parse_term t)));;
```

This implements the iterative deepening strategy. It tries to find the proof with the fewest universal instantiations; if it succeeds, it returns the number required and any toplevel instantiations, throwing away instantiations for internally created variables.

Examples

Here are some simple examples from Pelletier (1986).

```
#let P1 = prover "p --> q <-> ~(q) --> ~(p)";;
P1 : int * (string * term) list = 0, []
#let P13 = prover "p | q & r <-> (p | q) & (p | r)";;
P13 : int * (string * term) list = 0, []
#let P16 = prover "(p --> q) | (q --> p)";;
P16 : int * (string * term) list = 0, []
#let P18 = prover "exists(Y,forall(X,p(Y)-->p(X)))";;
P18 : int * (string * term) list = 2, []
#let P19 = prover "exists(X,forall(Y,forall(Z,
                 (p(Y)-->q(Z))-->p(X)-->q(X))))";;
P19 : int * (string * term) list = 6, []
```

A bigger example is the following:

```
#let P55 = prover
  "lives(agatha) & lives(butler) & lives(charles) &
   (killed(agatha,agatha) | killed(butler,agatha) |
    killed(charles,agatha)) &
   (forall(X,forall(Y,
     killed(X,Y) --> hates(X,Y) & ~(richer(X,Y))))) &
   (forall(X,hates(agatha,X)
            --> ~(hates(charles,X)))) &
   (hates(agatha,agatha) & hates(agatha,charles)) &
   (forall(X,lives(X) & ~(richer(X,agatha))
            --> hates(butler,X))) &
   (forall(X,hates(agatha,X) --> hates(butler,X))) &
   (forall(X,~(hates(X,agatha)) | ~(hates(X,butler))
            | ~(hates(X,charles))))
   --> killed(agatha,agatha)";;
P55 : int * (string * term) list = 6, []
```

In fact, the prover can work out 'whodunit':

```
#let P55' = prover
  "lives(agatha) & lives(butler) & lives(charles) &
   (killed(agatha,agatha) | killed(butler,agatha) |
    killed(charles,agatha)) &
   (forall(X,forall(Y,
     killed(X,Y) --> hates(X,Y) & ~(richer(X,Y))))) &
   (forall(X,hates(agatha,X)
            --> ~(hates(charles,X)))) &
   (hates(agatha,agatha) & hates(agatha,charles)) &
   (forall(X,lives(X) & ~(richer(X,agatha))
            --> hates(butler,X))) &
   (forall(X,hates(agatha,X) --> hates(butler,X))) &
   (forall(X,~(hates(X,agatha)) | ~(hates(X,butler))
            | ~(hates(X,charles))))
   --> killed(X,agatha)";;
P55' : int * (string * term) list = 6, ["X", `agatha`]
```

Further reading

Symbolic differentiation is a classic application of functional languages. Other symbolic operations are major research issues in their own right — Davenport, Siret, and Tournier (1988) give a readable overview of what computer algebra systems can do and how they work. Paulson (1983) discusses the kind of simplification strategy we use here in a much more general setting.

Parsing with higher order functions is another popular example. It seems to have existed in the functional programming folklore for a long time; an early treatment is given by Burge (1975). Our presentation has been influenced by the treatments in Paulson (1991) and Reade (1989).

The first definition of 'computable real number' in our sense was in the original paper of Turing (1936). His approach was based on decimal expansions, but he needed to modify it (Turing 1937) for reasons explored in the exercises below. The approach to real arithmetic given here follows the work of Boehm, Cartwright, O'Donnel, and Riggle (1986). More recently a high-performance implementation in CAML Light has been written by Ménissier-Morain (1994), whose thesis (in French) contains detailed proofs of correctness for all the algorithms for elementary transcendental functions. For an alternative approach using 'linear fractional transformations', see Potts (1996).

Our approach to Prolog search and backtracking, either using lists or continuations, is fairly standard. For more on implementing Prolog, see for example Boizumault (1993), while for more on the actual use of the language, the classic text by Clocksin and Mellish (1987) is recommended. A detailed discussion of continuations in their many guises is given by Reynolds (1993). The unification algorithm given here is simple and purely functional, but rather inefficient. For faster imperative algorithms, see Martelli and Montanari (1982). Our theorem prover is based on leanT^AP (Beckert and Posegga 1995). For another important theorem proving method conveniently based on Prolog-style search, look at the Prolog Technology Theorem Prover (Stickel 1988).

Exercises

1. Modify the printer so that each operator has an associativity, used when printing iterated cases of the operator to avoid excessive parentheses.

2. The differentiation rules for inverses $1/g(x)$ and quotients $f(x)/g(x)$ are not valid when $g(x) = 0$. Several others for functions like ln are also true only under certain conditions. Actually most commercial computer algebra systems ignore this fact. However, try to do better, by writing a new version of the differentiator that not only produces the derivative but also a list of conditions that must be assumed for the derivative to be valid.

3. Try programming a simple procedure for indefinite integration. In general, it will not be able to do every integral, but try to make it understand a few basic principles.[6]

4. Read the documentation for the `format` library, and try to implement a printer that behaves better when the output is longer than one line.

5. What happens if the parser functions like `term`, `atom` etc. are all eta-reduced by deleting the word `input`? What happens if `precedence` is consistently eta-reduced by deleting `input`?

6. How well do precedence parsers generated by `precedence` treat distinct operators with the same precedence? How can the behaviour be improved? Can you extend it to allow a mix of left and right associative operators? For example, one really wants subtraction to associate to the left.

7. Rewrite the parser so that it gives helpful error messages when parsing fails.

8. Try representing a real number by the function that generates the first n digits in a decimal expansion. Can you implement the addition function?

9. (*) How can you retain a positional representation while making the basic operations computable?

[6] In fact there are algorithms that can integrate all algebraic expressions (not involving sin, ln etc.) — see Davenport (1981).

10. Implement multiprecision integer arithmetic operations yourself. Make your multiplication algorithm $O(n^{log_2 3})$ where n is the number of digits in the arguments.

11. Using `memo` as we have defined it, is it the case that whatever series of arguments a function is applied to, it will always give the same answer for the same argument? That is, are the functions true mathematical functions despite the internal use of references?

12. Write a parser-evaluator to read real-valued expressions and create the functional representation for the answer.

13. (*) Extend the real arithmetic routines to some functions like *exp* and *sin*.

14. Our preprocessing and negation normal form process can take exponential time in the length of the input in the case of many 'if and only if's. Modify it so that it is linear in this number.

15. Extend the initial transformation into negation normal form so that it gives Skolem normal form.[7]

16. (*) Implement a more efficient unification algorithm, e.g. following Martelli and Montanari (1982).

17. (*) Implement a version of the Prolog Technology Theorem Prover (Stickel 1988)

[7] Skolem normal form is covered in most elementary logic books, e.g. Enderton (1972).

Bibliography

Aagaard, M. and Leeser, M. (1994) Verifying a logic synthesis tool in Nuprl: A case study in software verification. In v. Bochmann, G. and Probst, D. K. (eds.), *Computer Aided Verification: Proceedings of the Fourth International Workshop, CAV'92*, Volume 663 of *Lecture Notes in Computer Science*, Montreal, Canada, pp. 69–81. Springer Verlag.

Abramsky, S. (1990) The lazy lambda-calculus. In Turner, D. A. (ed.), *Research Topics in Functional Programming*, Year of Programming series, pp. 65–116. Addison-Wesley.

Adel'son-Vel'skii, G. M. and Landis, E. M. (1962) An algorithm for the organization of information. *Soviet Mathematics Doklady*, **3**, 1259–1262.

Backus, J. (1978) Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. *Communications of the ACM*, **21**, 613–641.

Barendregt, H. P. (1984) *The Lambda Calculus: Its Syntax and Semantics*, Volume 103 of *Studies in Logic and the Foundations of Mathematics.* North-Holland.

Barwise, J. (1989) Mathematical proofs of computer correctness. *Notices of the American Mathematical Society*, **7**, 844–851.

Beckert, B. and Posegga, J. (1995) leanT^AP: Lean, tableau-based deduction. *Journal of Automated Reasoning*, **15**, 339–358. Also available on the Web from `http://i12www.ira.uka.de/~posegga/LeanTaP.ps.Z`.

Boehm, H. J., Cartwright, R., O'Donnel, M. J., and Riggle, M. (1986) Exact real arithmetic: a case study in higher order programming. In *Conference Record of the 1986 ACM Symposium on LISP and Functional Programming*, pp. 162–173. Association for Computing Machinery.

Boizumault, P. (1993) *The implementation of Prolog.* Princeton series in computer science. Princeton University Press. Translated from 'Prolog: l'implantation' by A. M. Djamboulian and J. Fattouh.

Boyer, R. S. and Moore, J. S. (1979) *A Computational Logic.* ACM Monograph Series. Academic Press.

Burge, W. H. (1975) *Recursive Programming Techniques.* Addison-Wesley.

Church, A. (1936) An unsolvable problem of elementary number-theory. *American Journal of Mathematics*, **58**, 345–363.

Church, A. (1940) A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, **5**, 56–68.

Church, A. (1941) *The calculi of lambda-conversion*, Volume 6 of *Annals of Mathematics Studies.* Princeton University Press.

Clocksin, W. F. and Mellish, C. S. (1987) *Programming in Prolog* (3rd ed.). Springer-Verlag.

Curry, H. B. (1930) Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, **52**, 509–536, 789–834.

Davenport, J. H. (1981) *On the integration of algebraic functions*, Volume 102 of *Lecture Notes in Computer Science.* Springer-Verlag.

Davenport, J. H., Siret, Y., and Tournier, E. (1988) *Computer algebra: systems and algorithms for algebraic computation.* Academic Press.

Davis, M. D., Sigal, R., and Weyuker, E. J. (1994) *Computability, complexity, and languages: fundamentals of theoretical computer science* (2nd ed.). Academic Press.

de Bruijn, N. G. (1972) Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, **34**, 381–392.

DeMillo, R., Lipton, R., and Perlis, A. (1979) Social processes and proofs of theorems and programs. *Communications of the ACM*, **22**, 271–280.

Dijkstra, E. W. (1976) *A Discipline of Programming.* Prentice-Hall.

Enderton, H. B. (1972) *A Mathematical Introduction to Logic.* Academic Press.

Frege, G. (1893) *Grundgesetze der Arithmetik begriffsschrift abgeleitet.* Jena. Partial English translation by Montgomery Furth in 'The basic laws of arithmetic. Exposition of the system', University of California Press, 1964.

Girard, J.-Y., Lafont, Y., and Taylor, P. (1989) *Proofs and Types*, Volume 7 of *Cambridge Tracts in Theoretical Computer Science.* Cambridge University Press.

Gordon, A. D. (1994) *Functional Programming and Input/Output.* Distinguished Dissertations in Computer Science. Cambridge University Press.

Gordon, M. J. C. (1988) *Programming Language Theory and its Implementation: applicative and imperative paradigms.* Prentice-Hall International Series in Computer Science. Prentice-Hall.

Gordon, M. J. C., Milner, R., and Wadsworth, C. P. (1979) *Edinburgh LCF: A Mechanised Logic of Computation*, Volume 78 of *Lecture Notes in Computer Science.* Springer-Verlag.

Henson, M. C. (1987) *Elements of functional languages.* Blackwell Scientific.

Hindley, J. R. and Seldin, J. P. (1986) *Introduction to Combinators and λ-Calculus*, Volume 1 of *London Mathematical Society Student Texts.* Cambridge University Press.

Hudak, P. (1989) Conception, evolution, and application of functional programming languages. *ACM Computing Surveys*, **21**, 359–411.

Huet, G. (1980) Confluent reductions: abstract properties and applications to term rewriting systems. *Journal of the ACM*, **27**, 797–821.

Kleene, S. C. (1935) A theory of positive integers in formal logic. *American Journal of Mathematics*, **57**, 153–173, 219–244.

Lagarias, J. (1985) The $3x + 1$ problem and its generalizations. *The American Mathematical Monthly*, **92**, 3–23. Available on the Web as `http://www.cecm.sfu.ca/organics/papers/lagarias/index.html`.

Landin, P. J. (1966) The next 700 programming languages. *Communications of the ACM*, **9**, 157–166.

Lindemann, F. (1882) Über die Zahl π. *Mathematische Annalen*, **120**, 213–225.

Mairson, H. G. (1990) Deciding ML typability is complete for deterministic exponential time. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages (POPL)*, San Francisco, pp. 382–401. Association for Computing Machinery.

Mairson, H. G. (1991) Outline of a proof theory of parametricity. In Hughes, J. (ed.), *1991 ACM Symposium on Functional Programming and Computer Architecture*, Volume 523 of *Lecture Notes in Computer Science*, Harvard University, pp. 313–327.

Martelli, A. and Montanari, U. (1982) An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, **4**, 258–282.

Mauny, M. (1995) Functional programming using CAML Light. Available on the Web from `http://pauillac.inria.fr/caml/tutorial/index.html`.

Ménissier-Morain, V. (1994) *Arithmétique exacte, conception, algorithmique et performances d'une implémentation informatique en précision arbitraire.* Thèse, Université Paris 7.

Michie, D. (1968) "Memo" functions and machine learning. *Nature*, **218**, 19–22.

Milner, R. (1978) A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, **17**, 348–375.

Mycroft, A. (1981) Abstract interpretation and optimising transformations of applicative programs. Technical report CST-15-81, Computer Science Department, Edinburgh University, King's Buildings, Mayfield Road, Edinburgh EH9 3JZ, UK.

Neumann, P. G. (1995) *Computer-related risks.* Addison-Wesley.

Oppen, D. (1980) Prettyprinting. *ACM Transactions on Programming Languages and Systems*, **2**, 465–483.

Paulson, L. C. (1983) A higher-order implementation of rewriting. *Science of Computer Programming*, **3**, 119–149.

Paulson, L. C. (1991) *ML for the Working Programmer.* Cambridge University Press.

Pelletier, F. J. (1986) Seventy-five problems for testing automatic theorem provers. *Journal of Automated Reasoning*, **2**, 191–216. Errata, JAR 4 (1988), 235–236.

Peterson, I. (1996) *Fatal Defect : Chasing Killer Computer Bugs.* Arrow.

Potts, P. (1996) Computable real arithmetic using linear fractional transformations. Unpublished draft for PhD thesis, available on the Web as `http://theory.doc.ic.ac.uk/~pjp/pub/phd/draft.ps.gz`.

Raphael, B. (1966) The structure of programming languages. *Communications of the ACM*, **9**, 155–156.

Reade, C. (1989) *Elements of Functional Programming.* Addison-Wesley.

Reynolds, J. C. (1993) The discoveries of continuations. *Lisp and Symbolic Computation*, **6**, 233–247.

Robinson, J. A. (1994) Logic, computers, Turing and von Neumann. In Furukawa, K., Michie, D., and Muggleton, S. (eds.), *Machine Intelligence 13*, pp. 1–35. Clarendon Press.

Schönfinkel, M. (1924) Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, **92**, 305–316. English translation, 'On the building blocks of mathematical logic' in van Heijenoort (1967), pp. 357–366.

Schwichtenberg, H. (1976) Definierbare Funktionen im λ-Kalkül mit Typen. *Arkhiv für mathematische Logik und Grundlagenforschung*, **17**, 113–114.

Stickel, M. E. (1988) A Prolog Technology Theorem Prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, **4**, 353–380.

Turing, A. M. (1936) On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society (2)*, **42**, 230–265.

Turing, A. M. (1937) Correction to Turing (1936). *Proceedings of the London Mathematical Society (2)*, **43**, 544–546.

van Heijenoort, J. (ed.) (1967) *From Frege to Gödel: A Source Book in Mathematical Logic 1879–1931*. Harvard University Press.

Whitehead, A. N. (1919) *An Introduction to Mathematics*. Williams and Norgate.

Whitehead, A. N. and Russell, B. (1910) *Principia Mathematica (3 vols)*. Cambridge University Press.

Winskel, G. (1993) *The formal semantics of programming languages: an introduction*. Foundations of computing. MIT Press.

Wittgenstein, L. (1922) *Tractatus Logico-Philosophicus*. Routledge & Kegan Paul.

Wright, A. (1996) Polymorphism for imperative languages without imperative types. Technical Report TR93-200, Rice University.