The HOL Light manual (1.1)

John Harrison

University of Cambridge Computer Laboratory
New Museums Site
Pembroke Street
Cambridge CB2 3QG
ENGLAND
jrh@cl.cam.ac.uk
http://www.cl.cam.ac.uk/users/jrh

21 April 2000

Legal notice

Preface

HOL Light is a relatively new version of the HOL theorem prover (Gordon and Melham 1993). The whole implementation, even the axiomatization of the logic, has been re-engineered and simplified. Compared with other versions of HOL, it is relatively small and clean, and makes modest demands on the machine it is run on. The material that follows is not only a tutorial on the use of HOL Light and its interaction language, but also provides a detailed discussion of the implementation.

HOL Light proves theorems in a system of classical higher order logic based on polymorphic simple type theory. All proof proceeds by the application of low-level primitive rules, maintaining a high degree of reliability. However, a suite of derived rules for proving various useful theorems automatically is provided, as is a full programming language in which users can implement their own derived rules. A number of useful mathematical theories, e.g. real analysis, are already available.

To become an expert user of HOL Light, it is necessary to know something about programming in CAML Light, which is the implementation and interaction language. However, for readers primarily interested in theorem proving, it's no doubt somewhat dispiriting to spend a long time studying functional programming before even beginning to prove theorems. We have tried to minimize this problem in the organization that follows.

We begin with a short introductory chapter highlighting the basic features of CAML and HOL, including the basic mechanism of user interaction and the principles behind derived inference rules. Features of HOL and CAML are illustrated as we go, and most readers will be able to pick up the general ideas. This introduction is followed by the two larger Parts, comprising systematic introductions to CAML and HOL respectively. While these can be tackled in sequence, the impatient reader can read them in parallel, or even read the HOL part first and refer back to the CAML part as needed. (Indeed, there are a number of obvious parallels between CAML and the HOL logic, with both being an enriched version of lambda calculus, and both having a similar system of types. Reading these parts in parallel will show many similar concepts like currying and polymorphism in two different contexts.) Since HOL Light is aimed particularly at the enthusiast who wants to implement custom theorem-proving tools, a third Part gives an overview of the implementation, explaining the basic structure of the system and discussing various design decisions.

We hope that users interested in building custom theorem proving tools, or just in understanding the architecture of a modern theorem prover, will find something of interest in HOL Light and the present document. While we are writing primarily for those interested in theorem proving, the system might be considered interesting for two other reasons: it is a large application of (impure) functional programming, and it includes a systematic logical development of nontrivial mathematics from its very foundations à la *Principia Mathematica* (Whitehead and Russell 1910).

I do not assume that the reader is familiar with HOL or any similar system. Some knowledge of programming and of basic logic would be of great benefit, but not essential. However the present introduction is not comprehensive, and the serious user will need to spend time browsing through the source code.

Acknowledgements

HOL Light is one of a long line of 'HOL' theorem provers that have been released into the public domain for applications in academia, industry and government organizations.

Most of the important ideas behind the software, and many of the same functions and keywords, are taken from the original version of HOL, written by Mike Gordon. This in turn drew directly from the Edinburgh LCF project led by Robin Milner (and including Lockwood Morris, Malcolm Newey, Mike Gordon and Chris Wadsworth in the team), and some of the reengineering and rationalization of the system by Larry Paulson. Early versions of HOL were honed into successful tools by many people in the University of Cambridge and further afield, especially Tom Melham.

HOL Light began as a distillation of the simple core parts of HOL, which was done in collaboration with Konrad Slind, based on his hol90 system. The rest of the system was written gradually over the course of several years by John Harrison, and eventually practically all of the original hol90 code was rewritten. Although this is the first public release, several people have used the system, made helpful suggestions and pointed out bugs. Thanks to Donald Syme, B Karthikeyan, Michael Norrish and Mark Woodcock, among others.

Contents

Chapter 1

Introduction

In the following chapter we explain the key ideas behind HOL Light and cover the basics of interaction with the system. It is intended merely to give a brief taste, and readers wanting a more systematic introduction should study the subsequent chapters.

1.1 What is HOL Light?

There are many computer programs, e.g. as used in ordinary pocket calculators, for dealing with numerical problems like adding 2 and 2. Other programs, such as the computer algebra systems Maple[1] and Mathematica[2], can cope not just with particular numbers, but also with expressions involving variables. For example they can calculate that the derivative of x^2 with respect to x evaluated at the point x is $2x$.

These programs are usually thought of as calculating the answers to problems. But one can also look at them as systems that produce, on demand, mathematical theorems in a certain class. If we use the symbol $\vdash$ to indicate that an assertion is actually a true theorem of mathematics, we might say that these programs produce the following theorems, when given the appropriate left-hand sides:

$$\vdash 2 + 2 = 4$$

or

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

HOL Light is similar: it is a system for producing theorems on demand. Compared with calculators or computer algebra systems (CASs), it has two great advantages:

- HOL Light can produce theorems covering a wide mathematical range, e.g. involving infinite sets and so-called *quantifiers* like 'there exists some integer such that ...' or 'for any set of real numbers ...'. By contrast, calculators and CASs mainly produce unconditional equations with any variables implicitly regarded as universal.

- The theorems it produces can be relied on to be unambiguous in meaning and rigorously proven. By contrast, the exact readings of 'theorems' produced by

[1]Maple is a registered trademark of Waterloo Maple Software.
[2]Mathematica is a registered trademark of Wolfram Research Inc.

calculators and CASs are often open to doubt — even for something as trivial as explicit calculation involving approximations like $sin(0.7) = 0.6442176872$. Moreover, CASs often leave out essential sideconditions such as denominators of fractions being nonzero.

Needless to say, this greater power and reliability comes at a price.

- Only in limited problem domains can HOL Light produce its theorems completely automatically. In general, the user needs to describe a suitable mathematical proof in reasonable detail — HOL Light merely fills in some of the simpler gaps and checks that the user doesn't make mistakes.

- Whereas calculators and CASs are highly efficient and optimized for the typical problems, HOL Light derives its theorems via a uniform mechanism which tends to be less efficient in particular cases.

Like good calculators and CASs, HOL Light is programmable. This means that one can start with the available functions for proving certain theorems automatically, and produce new ones for particular tasks by implementing them in terms of the original ones. Similarly, a simple scientific calculator might have a built-in function to approximate sin, but none for evaluating, say, areas under the normal distribution curve — the user has to program the latter. Once this has been done, it can itself become a subroutine in more complex operations.

The majority of the HOL Light system is a tower of such functions. Right at the bottom, a very small set of primitive operations ultimately produce all theorems. In terms of these, more convenient higher-level functions are defined, these are themselves used to build up additional layers, and so on. Any user can build up this tower further. Because theorems are ultimately produced by the primitive rules, errors in higher-level functions cannot lead to false 'theorems' being produced; this explains the claim that HOL Light is relatively reliable. (A similar claim cannot be made for ordinary calculators since the answers are often approximate, and it's hard to analyze how the inaccuracy builds up.)

This approach to theorem proving, using programmability to build up from a small and reliable logical core, originated with the Edinburgh LCF project (Gordon, Milner, and Wadsworth 1979). For the approach to be palatable, the programming language must be well suited to the task, and as part of the LCF project a completely new programming language called ML[3] was developed. ML has since taken on a life of its own and is currently being widely touted as a general-purpose language. It is a higher-order functional programming language, featuring a novel polymorphic type system (Milner 1978) and a simple but useful exception mechanism as well as some traditional imperative features.

The version of ML used in HOL Light is CAML Light (Weis and Leroy 1993). This language and an excellent lightweight interpreter for it have been developed by a team at INRIA Rocquencourt in Paris. HOL Light has no separate user interface: the user actually works inside the CAML interpreter with all the HOL Light infrastructure loaded in.

HOL Light is the latest in a line of theorem provers going back to the mid-eighties, using the LCF approach to implement a theorem prover for classical Higher Order Logic (hence the name HOL). Previous versions have included HOL88, hol90, ICL ProofPower, and more recently hol98. HOL Light is intended to be a more simple and elegant version targeted at users who really want to understand how the

[3]ML for metalanguage; following Tarski (1936) and Carnap (1937), it has become usual to enforce a strict separation between the 'object language' under consideration and the 'metalanguage' used to talk about it. For example in a course in Russian given in English, Russian is the object language and English the metalanguage.

system works, or who want to build their own application-specific theorem proving tools.

1.2 Getting started

After starting up CAML and loading HOL, the user is confronted with CAML Light's prompt ('`#`'). CAML Light is expecting the user to type something in, and it will then evaluate it and print the result. CAML will only act after the user terminates the input with a double semicolon ('`;;`') and newline. For example, one can use CAML like a pocket calculator:

```
#2 + 2;;
it : int = 4
```

The user enters the expression `2 + 2`, and CAML evaluates it and prints the answer, `4`. It also prints out the *type* of the expression, namely `int` (short for integer, i.e. whole number). We will explain CAML's types in more detail later. CAML also abbreviates the result by '`it`', to save the user retyping. For example, one can now do:

```
#it + 3;;
it : int = 7
```

Instead of using the default name `it`, which is overwritten every time a new expression is evaluated, one can bind an expression to a name by using `let`. For example, after the following interaction, `x` has the value `4`, at least until another '`let x = ...`' overwrites it.

```
#let x = 2 + 2;;
x : int = 4
```

The above was only intended as an introduction to interaction with CAML. We are really interested in manipulating not numbers but logical entities like theorems. In fact, there are three key logical notions in HOL Light, each with a corresponding ML type: types (`hol_type`), terms (`term`) and theorems (`thm`). HOL Light is, at its core, a system for manipulating these objects. (Note the object-meta distinction here: one has an ML (meta) type of data structures representing HOL (object) types.)

A HOL *term* represents a mathematical assertion like $x + 1 = y$ or just some mathematical expression like $x + 1$. Every term has a *type*, indicating what sort of mathematical entity it is, e.g. a boolean value (true or false), a real number, a set of real functions etc. For example, $x + 1$ has type `num` indicating that it is a natural number, while $x + 1 = y$ has type `bool` indicating that it is either true or false. A HOL theorem simply asserts that some boolean-typed term is valid, or at least, follows from a finite list of assumptions.

Terms and types are represented by ML data structures that we describe in more detail below. However, it is tiresome to describe *particular* terms and types, especially large ones, by creating such data structures explicitly. Instead, HOL has parsers and printers that allow types and terms to be represented in something closer to familiar mathematical notation, subject to the limitations of ASCII. Terms are entered rather like strings, enclosed within backquotes:

```
#`x + 1`;;
it : term = `x + 1`
#`x + y <= z`;;
it : term = `x + y <= z`
```

This however hides quite a lot of processing. Quotations are expanded (by a front-end filter separate from CAML proper) into a call of a term parser and type inferencer. This not only analyzes the syntactic structure of the term but works out types for the term as a whole and all its subterms. For example, it knows that the constant `1` has type `num`, and that the left and right arguments of `+` must have the same type, which is also the type of the result. Hence it decides that `x` and the term as a whole must also have type `num`. If the user tries to enter a term that cannot be typed, e.g. `` `(1 <= 2) + 3` ``, the typechecker will fail. If, on the other hand, there is not enough type information to fix the types of all subterms, type variables are invented and a warning given:

```
#`x`;;
Warning: inventing type variables
it : term = `x`
```

The use can annotate the term or any subterms with types by writing a colon followed by a type, e.g.

```
#`x:num`;;
it : term = `x`
```

The parser does not allow the same variable to have different types in the same term.[4] It is possible to create such terms by hand using the functions described later, but is apt to look confusing. Note that identically-named variables with different types are treated as different. Types, rather than terms, can be entered by simply omitting the term, i.e. starting the quotation with a colon, e.g.

```
#`:bool`;;
it : hol_type = `:bool`
```

HOL types and terms are not actually ML abstract types (they could easily be made so by separately compiling the modules), but the user is expected to use the standard interface functions. These restrict formation to those that are well-formed and well-typed. So even using the basic constructors, it is impossible to create, for example, a term that adds a number and a boolean value. Theorems can also only be created by, at bottom, a small set of basic functions. One of these is the function `REFL` which takes a term `t` as an argument and returns a rather trivial theorem saying that `t` is equal to itself:

```
#REFL `x + 1`;;
it : thm = |- x + 1 = x + 1
```

HOL prints theorems using an ASCII approximation to the conventional 'turnstile' symbol $\vdash$. If a theorem has assumptions, these are printed to the left of the turnstile. For example, another primitive function `ASSUME` takes a term `p` of Boolean type and returns the theorem (once again rather trivial) that under the assumption that `p` holds, `p` holds:

```
#ASSUME `p:bool`;;
it : thm = p |- p
#ASSUME `1`;;
Uncaught exception: Failure "ASSUME: not a proposition"
```

[4]Or more precisely, in the same scope. Separately bound instances can have different types — see later.

While the user can enter any (typeable) term in quotations and have it elevated to a HOL term, it is not possible to do this with theorems. While there's a computable procedure for deciding if a term is well-typed, HOL has no way in general of deciding whether it is possible to construct a theorem from the primitive functions. However, there are some high-level functions that accept a term of a certain form and prove it automatically, turning it into a theorem. For example `ARITH_RULE` can prove many basic facts of natural number arithmetic:

```
#ARITH_RULE `2 * x < 2 * (x + 1)`;;
it : thm = |- 2 * x < 2 * (x + 1)
```

Note, however, that the theorem is still created under the surface by a (sometimes quite lengthy) series of applications of the primitive rules, maintaining the guarantee of reliability.

1.3 Derived rules

In general, an inference rule in HOL is simply any ML function that return a theorem or theorems (objects with ML type `thm`). Ones like `ARITH_RULE` that turn claims into theorems are particularly simple to use, but in general HOL inference rules may require other theorems as input. For example `MK_COMB` accepts two theorems as input, one saying that two functions (say f and g) are equal, the other saying two arguments (say x and y) are equal, and if the types match up correctly so it makes sense to apply f to x and g to y, `MK_COMB` returns a theorem saying that $f(x)$ and $g(y)$ are equal.

```
#let th1 = ASSUME `f:num->num = g`;;
th1 : thm = f = g |- f = g
#let th2 = ASSUME `m:num = n`;;
th2 : thm = m = n |- m = n
#MK_COMB(th1,th2);;
it : thm = f = g, m = n |- f m = g n
```

HOL rules can be separated into the *primitive* rules like `REFL`, `ASSUME` and `MK_COMB`, of which there are ten, and all the others, which are called *derived* rules, since they are built up from the primitives. A lot of HOL Light's source code is a systematic building up of a useful set of higher-level derived rules, and the use of the rules, primitive and derived, to prove useful mathematical theorems. Here is a very simple but genuine example, one of HOL Light's simplest inbuilt derived rules called `AP_TERM`. It accepts a term representing a function f and a theorem asserting that x and y are equal, and if the types match up, returns a theorem asserting that $f(x) = f(y)$:

```
#let AP_TERM tm th =
   MK_COMB(REFL tm,th);;
AP_TERM : term -> thm -> thm = <fun>
#AP_TERM `h:num->num` (ASSUME `m = 1`);;
it : thm = m = 1 |- h m = h 1
```

The definition of `AP_TERM` is a simple 2-line ML program, which first derives the trivial theorem that the function is equal to itself, using `REFL`, and then calls `MK_COMB` to get the final result. Note that this just expresses generically the way one would prove such a theorem given only the primitive rules to work with. A derived rule doesn't yield a single theorem, but rather a whole family of theorems depending on the input. It corresponds naturally to what a logician would think of as a 'derived rule'.

CAML Light, described in the next Part, is a full programming language, so one can perform essentially any kinds of inference one wants, provided it is reduced to the existing infrastructure of primitive and derived rules. Derived rules often have a recursive structure, passing over the input term and transforming it into an appropriate theorem. They may also do different things depending, for example, on the logical structure of the input, the names of variables, and so on. All this will be illustrated in more detail in what follows.

Further reading

The original textbook on Edinburgh LCF by Gordon, Milner, and Wadsworth (1979) introduces many of the basic ideas in HOL Light; see also the later book by Paulson (1987) on a re-engineered version 'Cambridge LCF'. The general approach to theorem-proving described above is, as emphasized by Gordon (1982), largely independent of the particular logic one works with, e.g. the original LCF (logic of computable functions), higher order logic, or first order set theory. The original HOL was born when Gordon used the Cambridge LCF system to implement classical higher order logic. There is a book by Gordon and Melham (1993) describing an early version of the system 'HOL88', while an interesting historical survey of the development of LCF and HOL is given by Gordon (2000). The original ML is also described in the early LCF publications. CAML Light has extensive on-line documentation and a book (in French) by Weis and Leroy (1993) devoted to it. Another ML version, Standard ML, is described by Paulson (1991).

Part I

CAML tutorial

Chapter 2

A taste of CAML

CAML Light feels rather different from common programming languages like C or FORTRAN. The major difference is that it is a *functional* rather than *imperative* language. While it does have imperative features, we won't make very great use of them. The following section explains the contrast; readers with no previous programming experience may choose to skip or just skim this material.

2.1 Imperative vs functional programming

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*. 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). 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 general 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

[1]Functional programming is often called 'applicative programming' since the basic mechanism is the *application* of functions to arguments.

$\sigma' = f(\sigma)$.[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, pure 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.

A 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 makes programs more comprehensible and debugging easier, since there is no danger of one part of a program unexpectedly affecting others. Moreover, functional programs may lend themselves well to parallel implementation, i.e. the computer can automatically farm out different subexpressions to different 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, CAML 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 when we do use assignments, and lose some of the preceding benefits, there are advantages in the more flexible use of functions that languages like CAML allow. Programs can often be expressed in a very concise and elegant style using higher-order functions (functions that operate on other functions). 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.

2.2 Basic use of CAML

We will use CAML in its interactive and interpretive mode. When it is started it presents its prompt (`#`):

```
>       Caml Light version 0.74

#
```

(In order to exit the system, simply type `ctrl/d` or `quit();;` at the prompt.) When CAML presents you with its prompt, you can type in expressions, terminated

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

by two successive semicolons, and it will evaluate them and print the result. In computing jargon, the CAML system sits in a read-eval-print loop: it repeatedly reads an expression, evaluates it, and prints the result. For example, CAML can be used as a simple calculator:

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

The system not only returns the answer, but also the *type* of the expression, which it has inferred automatically. (We will have more to say about CAML's types in a later section.) 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 CAML is a functional language, expressions are allowed to be functions. Functions can be written in CAML using the syntax `fun x -> t[x]` for the function that maps an argument `x` to `t[x]`, the latter being any expression involving `x`. Such an expression involving '`fun x -> ...`' is said to be a *function abstraction*. For example we can define the successor function:

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

Again, the type of the expression, this time `int -> int`, meaning a function from integers to integers, 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.[3] In normal mathematical notation, application of a function f to an argument x is written $f(x)$. In CAML, the parentheses can be omitted unless they are needed to enforce grouping, e.g.

```
#(fun x -> x + 1) 1 * 2;;
it : int = 4
#(fun x -> x + 1) (1 * 2);;
it : int = 3
#((fun x -> x + 1) 1) * 2;;
it : int = 4
```

Every function in CAML takes just a single argument. However there are two ways of getting the effect of functions of more than one argument. One way is to have a single argument but of a more complex type, such as pairs (see later) of integers. The other is to use 'currying' (after the logical Haskell Curry), where the function takes one argument and yields another function that takes the second argument, and so on. For example, a curried function of two arguments that adds the arguments together can be written and used as follows:

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

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

Note that the function has type `int -> int -> int`, meaning `int -> (int -> int)`. When applied to one argument, `1`, it yields another function, which takes the second argument and maps it to the corresponding sum. Currying is used a lot in functional programming, since it allows functions to be used quite flexibly. Some other syntactic conventions support it; for example, without parentheses to enforce grouping, function application associates to the left, i.e. $f\ g\ x$ means $(f\ g)(x)$ not $f(g(x))$. We can write the above example more succinctly as:

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

2.3 Bindings and declarations

A nontrivial functional program is a very complex expression, and it is of course not convenient to evaluate it all in one go. Instead, useful subexpressions can be evaluated and bound to names using `let`. (In fact, a filter in front of CAML Light, part of HOL Light, automatically binds the last anonymous expression evaluated to the special name `it`, hence its appearance above.) For example:

```
#let successor = fun x -> x + 1;;
successor : int -> int = <fun>
#successor 5;;
it : int = 6
```

Declarations can be made local to the evaluation of an expression, so they are invisible afterwards, using `in`. For example:

```
#let suc = fun x -> x + 1 in
  suc(suc 1);;
it : int = 3
#suc 1;;
Toplevel input:
>let it = suc 1;;
>         ^^^
The value identifier suc is unbound.
```

The arguments to functions can be written on the left of the equation, which most people find more natural:

```
#let successor x = x + 1;;
successor : int -> int = <fun>
#successor 5;;
it : int = 6
```

Functions can be *recursive*, i.e. defined in terms of themselves. To achieve this, simply include the keyword `rec`. For example, the factorial $n! = 1 \times 2 \times \cdots \times (n-1) \times n$ can be evaluated as follows: evaluate $(n-1)!$ recursively, then multiply by n:

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

By using `and`, one 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:

```
#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;;
it : bool = true
#odd 14;;
it : bool = false
```

If declarations do not include the `rec` keyword, then any instance of the name currently being bound on the right is taken to be the *previous* value. For example:

```
#let successor n = successor(successor n);;
successor : int -> int = <fun>
#successor 2;;
it : int = 4
#successor 5;;
it : int = 7
```

The old binding is now overwritten. But 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 scoping rules usual in programming languages. 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 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.[4] 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;;
it : int = 1
```

2.4 Evaluation rules

In essence, CAML is quite simple to understand, since it just evaluates expressions. However there are subtle questions over the precise order of evaluation. For example, consider the following recursive function:

```
#let rec f x = f(x + 1);;
f : int -> 'a = <fun>
#f 2;;
Interrupted.
```

Evaluation of `f 2` looped indefinitely, until interrupted by ctrl/c. Now suppose we use `f` in another expression, but in a way that doesn't require `f` to be evaluated on any arguments:

```
#(fun x -> 1) (f 2);;
Interrupted.
```

Even so, an indefinite loop results. The reason is that according to CAML's evaluation rules, all arguments to a function are evaluated before being inserted in the function body. This strategy is called *eager*, in contrast to cleverer *lazy* approaches that try to avoid evaluating subexpressions until they are definitely needed (and then no more than once).

CAML 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. By contrast, in CAML, we always first evaluate the arguments to functions and only then inserts them in the body — this is simple and efficient, and is easy to implement using standard compiler technology.

The second reason for preferring eager evaluation is that CAML is not a *pure* functional language, but includes imperative features (variables, assignments etc.). Therefore the order of evaluation of subexpressions can make a big difference. If lazy evaluation is used, it seems to become difficult for the programmer to visualize,

[4]The first version of LISP used *dynamic* binding, where a rebinding of a variable propagated to earlier uses of the variable. This was in fact originally regarded as a bug, but soon programmers started to appreciate its convenience. 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`.

in a nontrivial program, exactly when each subexpression gets evaluated. In the eager CAML system, one just needs to remember the simple evaluation rules. To be explicit, they are as follows:

- Constants (e.g. predefined values and functions like `1` and `+`) evaluate to themselves.
- Evaluation stops immediately at expressions of the form `fun x -> ...`, and does not look inside them. This only happens when such an expression is applied to an argument.
- When evaluating an application `s t`, then *first* both `s` and `t` are evaluated.[5] Then, assuming that the evaluated form of `s` is a function `fun x -> ...`, the body is evaluated with each instance of `x` replaced by the evaluated form of `t`. If the evaluated form of `s` is a built-in function like `+`, the appropriate evaluation is performed.
- When evaluating `if E1 then E2 else E3`, *first* `E1` is evaluated, and depending on whether it yields true or false, either `E2` or `E3` respectively (and not the other) is evaluated.

One can regard `let x = E1 in E2` as an abbreviation for `(fun x -> E2) E1`, and the above evaluation rules then give the right answer: `E1` is evaluated, and then the evaluated form replaces each `x` in `E1`, which is then itself evaluated. Let us see some examples of evaluating expressions:

```
  (fun x -> (fun y -> y + y) x) (2 + 2)
= (fun x -> (fun y -> y + y) x) 4
= (fun y -> y + y) 4
= 4 + 4
= 8
```

Note that the subterm `(fun y -> y + y) x` is *not* reduced, since it is inside the function abstraction '`fun x -> ...`'. However, terms that are reducible and *not* so enclosed in both function and argument get reduced before the function application itself is evaluated, e.g. the second step in the following:

```
  ((fun f x -> f x) (fun y -> y + y)) (2 + 2)
= ((fun f x -> f x) (fun y -> y + y)) 4
= (fun x -> (fun y -> y + y) x) 4
= (fun y -> y + y) 4
= 4 + 4
= 8
```

The fact that CAML does not evaluate under function abstractions is of crucial importance to advanced programmers. It gives precise control over the evaluation of expressions, and can be used to mimic many of the helpful cases of lazy evaluation, or sometimes to force earlier evaluation of expressions by moving them outside `fun x -> ...`.

2.5 Types and polymorphism

Some functions do not have a fixed type. For example, the identity function that returns its argument unchanged doesn't care whether its argument is an integer, a

[5] CAML Light actually evaluates t first.

boolean, or another function. Therefore, it is said to have *polymorphic* type, and CAML displays a type involving *type variables*. These can later be set to some particular type when it is used, different instances with different types.

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

CAML 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. CAML always assigns the most general type possible for an expression, without specializing it unnecessarily, using an algorithm due to Milner (1978). For example, the following is a more complex definition of an identity function; the reader may wish to study it to see why CAML gives all these expressions the types it does,[6] and why `I'` acts as an identity function. Note that in contrast to most programming languages, CAML allows the prime character in variable names, reflecting its background in logic and mathematics where variables like x' are common.

```
#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>
#let I' = S K K;;
I' : '_a -> '_a = <fun>
#I' 2;;
it : int = 2
```

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);;
```

Because of CAML's automatic type inference, the programmer need never enter a type. At least, CAML will already allocate as general a type as possible to an

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

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 CAML 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`:

```
#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>
```

2.6 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;;
it : 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;;
it : bool = true
```

Why these restrictions? Aren't functions supposed to be first-class objects in CAML? 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

[7]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).

illustration of why this might be so. It is still an open problem whether the 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. This is the approach taken by Standard ML. 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

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. A detailed and polemical advocacy of the functional style is given by Backus (1978), the main inventor of FORTRAN. A good elementary introduction to CAML Light and functional programming is Mauny (1995). Paulson (1991) is another good textbook, though based on Standard ML.

[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.

Chapter 3

Further CAML

In this chapter, we consolidate the previous examples by specifying the basic facilities of CAML 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 CAML'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 CAML 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 CAML'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 CAML 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. HOL Light runs a filter in front of CAML to expand backquotes into calls of term and type parser and typechecker. In order to make this happen when loading a file, use `loadt` instead of `include`.

In large programs, it is often helpful to include comments. In CAML, 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 *) *);;
```

3.1 Basic datatypes and operations

CAML 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.

The above values like `()`, `false`, `7` and `"caml"` are all to be regarded as fixed constants. 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:

[1] We will see later how to use an alternative type of integers with unlimited precision.

```
#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;;
it : int = 3
##infix "o";;
#let add3' = successor o successor o successor;;
add3' : int -> int = <fun>
#add3' 0;;
it : 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;;
it : int -> int = <fun>
#(prefix o) successor successor;;
it : 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:

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

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.

3.2 Syntax of CAML phrases

Expressions in CAML 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]

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

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

[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.

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;;
it : 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.

3.3 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>
```

We can apply `funpow` just to the first argument, and this encodes a natural number as a function that takes a function as an argument then iterates it the appropriate number of times, a so-called *Church numeral*.[3] Since functions aren't printed, we can't actually look at the expression representing a Church numeral:

```
#funpow 6;;
it : ('_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);;
it : int = 32
```

We can define some of the arithmetic operations on Church numerals. Understanding these definitions thoroughly is a good exercise.

[3] The basic idea was used earlier by Wittgenstein (1922), 6.021.

```
#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;;
it : int = 12
#test mul 2 10;;
it : int = 20
#test exp 2 10;;
it : int = 1024
```

The above is not a very efficient way of performing arithmetic operations. CAML does not have a built-in 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 $x^{2n+1} = (x^n)^2 x$ 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;;
it : int = 1024
#2 exp 20;;
it : 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;;
it : int = 4
#gcd 7 159;;
it : int = 1
#gcd 24 60;;
it : int = 12
```

Rather than using the `rec` keyword every time we declare a recursive function, eccentrics might prefer to define a recursion operator `Rec`, and thereafter use that, e.g.

```
#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 function abstraction '`fun x -> ...`' in the definition was essential, otherwise the expression `Rec f` goes into an infinite recursion when evaluated, 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
```

3.4 Type definitions

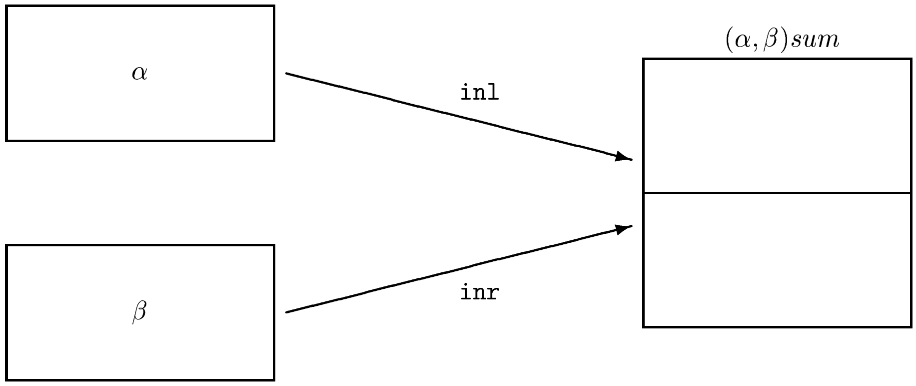
CAML has facilities for declaring new type constructors, so that composite types can be built up out of existing ones. In fact, CAML goes further and allows a composite type to be built up not only out of preexisting types but also from the composite type itself. Such types, naturally enough, are said to be *recursive*, even if they don't avail themselves of the chance to use the type being defined in the definition. 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 CAML system and apply them to objects:

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

3.4.1 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 function expression, e.g.

```
#fun (inl n) -> n > 6
   | (inr b) -> b;;
it : (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 CAML system then issues a warning:

```
#fun (inr b) -> b;;
Toplevel input:
>fun (inr b) -> b;;
>^^^^^^^^^^^^^^^^^
Warning: this matching is not exhaustive.
it : ('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 CAML, 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;;
it : int = 0
#(fun true -> 1 | false -> 0) (4 < 3);;
it : 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;;
it : int -> bool = <fun>
#(fun 0 -> true | n -> false) 0;;
it : bool = true
#(fun 0 -> true | n -> false) 1;;
it : 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.
it : 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 function 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.

3.4.2 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;;
it : 'a list = Nil
#Cons;;
it : '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;;
it : 'a list = Nil
#Cons(1,Nil);;
it : int list = Cons (1, Nil)
#Cons(1,Cons(2,Nil));;
it : int list = Cons (1, Cons (2, Nil))
#Cons(1,Cons(2,Cons(3,Nil)));;
it : 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, CAML 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:

```
#[];;
it : 'a list = []
#1::[];;
it : int list = [1]
#1::2::[];;
it : int list = [1; 2]
#1::2::3::[];;
it : 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];;
it : int = 1
#tl [1;2;3];;
it : 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];;
it : 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 [];;
it : int = 0
#length [5;3;1];;
it : 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);;
```

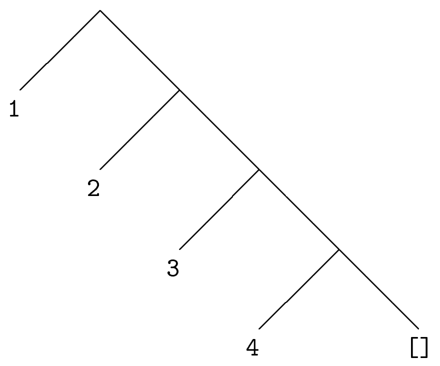
This latter style of function definition is more usual in many languages, notably LISP, but the direct use of pattern matching is often more elegant.

Some other classic list functions are appending (joining together) two lists, mapping a function over a list (i.e. applying it to each element) and reversing a list. We can define all these by recursion:

```
#let rec append l1 l2 =
   match l1 with
     [] -> l2
   | (h::t) -> h::(append t l2);;
append : ’a list -> ’a list -> ’a list = <fun>
#append [1;2;3] [4;5];;
it : int list = [1; 2; 3; 4; 5]
#let rec map f =
   fun [] -> []
     | (h::t) -> (f h)::(map f t);;
map : (’a -> ’b) -> ’a list -> ’b list = <fun>
#map (fun x -> 2 * x) [1;2;3];;
it : int list = [2; 4; 6]
#let rec rev =
   fun [] -> []
     | (h::t) -> append (rev t) [h];;
#rev [1;2;3;4];;
it : int list = [4; 3; 2; 1]
```

3.4.3 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));;
it : 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. Recursive types similar to these are used in HOL Light to define logical entities like terms.

3.4.4 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.[4] 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 a recursion operator without any explicit use of recursion, by using K as a kind of type cast.[5] The use of `let` is only used for the sake of efficiency, but we *do* need the extra argument `z` in order to prevent looping under CAML'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;;
it : int = 720
```

[4]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 $i(C) \in C \equiv i(C) \notin C$, 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 $\{s \mid s \notin j(s)\}$.

[5]Readers familiar with untyped λ-calculus may note that if the Ks are deleted, this is essentially the usual definition of the Y combinator.

Thus, recursive types are a powerful addition to the language.

Chapter 4

Effective CAML

In this chapter, we discuss some of the techniques and tricks that CAML 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.

4.1 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*. 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. 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 CAML 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];;
it : int = 15
#sum [];;
it : int = 0
#sum [1;1;1;1];;
it : 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];;
it : 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];;
it : bool = true
#forall (fun x -> x < 3) [1;2;3];;
it : 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.

4.2 Writing efficient code

Here we accumulate some common tricks of the trade, which can often make CAML 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.

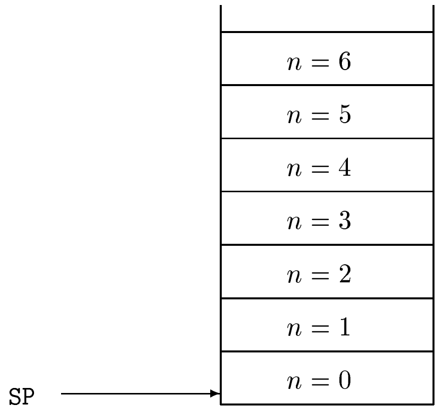
4.2.1 Tail recursion and accumulators

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 CAML 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

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

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;;
it : 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*.

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.

4.2.2 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 CAML 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 use, 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")));;
it : 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 CAML 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;;
```

4.2.3 Forcing evaluation

We have emphasized that, since CAML does not evaluate underneath function abstractions, one can use such constructs 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 function abstractions. 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 function abstraction. 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 it is here in CAML Light.

We might note, in passing, that if functions are implemented by plugging together combinators, with fewer explicit function abstractions, 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.

4.3 Imperative features

CAML has a fairly full complement of imperative features. We will not spend much time on the imperative style of programming, 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 HOL Light, and some knowledge of what is available will stand the reader in good stead for writing practical CAML code.

4.3.1 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 it 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"];;
it : string = "hi"
#headstring [];;
it : 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.

4.3.2 References and arrays

CAML 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 CAML 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 formal parameters of a function 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 CAML, 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;;
it : int = 1
#x := 2;;
it : unit = ()
#!x;;
it : int = 2
#x := !x + !x;;
it : unit = ()
#x;;
it : int ref = ref 4
#!x;;
it : 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;;
it : 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 CAML. 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:

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

```
#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;;
it : int = 0
#vect_assign v 1 10;;
it : unit = ()
#v;;
it : int vect = [|0; 10; 0; 0; 0|]
#vect_item v 1;;
it : int = 10
```

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

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

4.3.3 Sequencing

There is no need for an explicit sequencing operation in CAML, 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 CAML by using a single semicolon:

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

4.3.4 Interaction with the type system

While polymorphism works very well for the pure functional core of CAML, 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];;
it : unit = ()
```

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

```
#l;;
it : 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 occur 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;;
it : '_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 CAML 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.

Further reading

Hints and tips for practical programming can be found in many functional programming books, e.g. Paulson (1991). Methods used by language implementations to perform garbage collection are discussed in depth by Jones and Lins (1996).

Part II

HOL tutorial

Chapter 5

Primitive basis of HOL Light

The introductory chapter gave a brief introduction to the key ideas behind HOL and simple interaction with the system. Here we explain more systematically how mathematical and logical assertions are represented in HOL, and list all the primitive ways of producing theorems.

We should distinguish carefully between *abstract* and *concrete* syntax. The abstract syntax of a term, which HOL deals with internally, is a tree-like CAML data structure indicating how the term is built up from its components. While this is convenient to manipulate, humans are more used to representing terms by a linear sequence of characters, the concrete syntax. HOL's quotation parser automatically translates the concrete syntax into the abstract syntax, and its prettyprinter performs an inverse mapping back to concrete syntax. For simple use of HOL, it is not necessary to think much about the distinction, still less to understand details of the abstract syntax. However, we think it is best to cover this early, since it shows how simple the underlying structures really are. The present chapter can be read as an abstract description of the HOL logic, without considering the actual implementation in CAML. However when we discuss concrete syntax, we are implicitly talking about that accepted by HOL's parser.

5.1 Terms

HOL's logic is based on λ-calculus, a formalism invented by Alonzo Church. In HOL, as in λ-calculus, terms are built up starting just from *constants* and *variables* using *application* and *abstraction*. All mathematical and logical assertions are represented in this uniform way.

Constants and variables are probably familiar to the reader from an informal understanding of mathematics. They are used as the building-blocks of terms. Variables can have any name, e.g. n, x, p. Constants, e.g. [] (the empty list), $\top$ (true) and $\bot$ (false), are intended to be abbreviations for other terms, and except for a couple of primitive ones such as equality itself, need to have been defined before they can be used in terms. We will see below how the user can define new constants.

Application is application of a function to an argument, an operation used constantly in mathematics. The customary concrete syntax for the application of a function f to an argument t is $f(t)$. HOL, following lambda-calculus convention, allows the parentheses to be omitted, unless they are needed because t is itself a compound term. For example, $f(g(x))$ needs at least the outer pair of parentheses, as HOL's parser interprets $f\ g\ x$ to mean $(f(g))(x)$, for reasons explained shortly.

Abstraction is in a precise sense a converse operation to application. Given a variable x and a term t, which may or may not contain x, one can construct the so-called *lambda-abstraction* $\lambda x.\, t$, which means 'the function of x that yields t'. (In HOL's ASCII concrete syntax the backslash is used, e.g. `\x. t`.) For example, $\lambda x.\, x + 1$ is the function that adds one to its argument. Abstractions are not often seen in informal mathematics, but they have at least two merits. First, they allow one to write anonymous function-valued expressions without naming them (occasionally one sees $x \mapsto t[x]$ used for this purpose), and since our logic is avowedly higher order, it's desirable to place functions on an equal footing with first-order objects in this way. Secondly, they make variable dependencies and binding explicit; by contrast in informal mathematics one often writes $f(x)$ in situations where one really means $\lambda x.\, f(x)$.

We should give some idea of how ordinary mathematical and logical vocabulary (like $x + 1$ above) is represented in this simple term structure. The basic idea is quite simple. Fixed operations that one wants to use have constants corresponding to them. For example, the negation of a real number is represented by a constant `--`, and so $-x$ is represented by the application of the constant `--` to the variable `x`. Exactly the same idea is used for logical operations like negation ('not'), so $\neg p$ ('not p') is represented by the application of the logical negation constant `~` to the term p, whatever it may be.

Application makes no special provision for functions of more than one argument, such as addition. The trick used is known as *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 trick is to make the operation take its arguments 'one at a time'. For example, rather than considering addition as a function $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$, consider it as a function $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$. It accepts a single argument a, and yields a new function of one argument that adds a to its argument. This intermediate function is applied to the second argument, say b, and yields the final result $a + b$. In other words, what we write as $a + b$ is represented by HOL as $(+\ a)(b)$. (Certain operations like $+$ are written *infix* in the concrete syntax, for the sake of familiarity. But the use of currying is independent of this.)

This approach is used for many multiple-argument functions in HOL. However, there is also a pairing operation ',', once again written infix in the concrete syntax, that can also be used to form pairs of terms into new terms. Of course, this itself has to be curried, but all other functions can be written in 'uncurried' form to take a tuple as its argument. Thus, what is written in the concrete syntax as $f(x, y)$ is actually represented in HOL as $f((,\ x)(y))$.

Operations that bind variables are common in mathematics. For example, in $lim_{x\to\infty}\frac{1}{x}$, the variable x is *bound* by a variable-binding operation lim, and serves merely to connect different parts of the term. It can be renamed consistently, e.g. $lim_{y\to\infty}\frac{1}{y}$. By contrast, the inner term $\frac{1}{x}$ on its own depends on the value of x, and here x is said to be *free*. Some other examples of bound variables in mathematics and logic are the variable x in the set abstraction $\{x \mid P\}$ ('the set of all x such that P'), the variable n in $\Sigma_{n=1}^{N} n$ ('the sum of all n from 1 up to N') and the variable z in $\forall z.\, P$ ('for all z, P holds'). All these variable-binding operations are represented in HOL using special constants but with the actual variable-binding implemented by lambda-abstraction. For example, there is a constant `liminf` ('limit at infinity') and one then represents $lim_{x\to\infty}\frac{1}{x}$ by `liminf`$(\lambda x.\, \frac{1}{x})$, or expanding the body completely, `liminf`$(\lambda x.\, (/\ 1)(x))$. This means one should think of `liminf` as a function from real functions to reals, i.e. $(\mathbb{R} \to \mathbb{R}) \to \mathbb{R}$. Similarly, the logical assertion $\forall x.\, P$ is represented using the constant `!` as `!`$(\lambda x.\, P)$.

It is well-known that there is a 1-1 correspondence between sets of elements (drawn from some global 'universe' set U), and predicates or 'characteristic functions' $U \to 2$, where 2 is some 2-element set of truth values. In HOL, there is no separate notion of 'set': they are identified with predicates, i.e. Boolean-valued functions. Thus, one can simply write $s\ x$ instead of $x \in s$, though the latter is also possible using the infix constant `IN`, e.g. `x IN s`. It is thus normal and often convenient to slip between thinking of truth-functions as predicates or as sets, even within the same term.

5.2 Types

Application and abstraction are converse in the precise sense that $(\lambda x.\ t)(x)$ is equal to t, and there is a primitive HOL rule to make this inference and produce the theorem $\vdash (\lambda x.\ t)(x) = t$. More generally, HOL is capable of proving that $\vdash (\lambda x.\ t)(s) = t[s/x]$ where the right-hand side denotes the appropriate (see later) replacement of each instance of x in t by s. For example, $(\lambda x.\ 1 + x)(y) = 1 + y$. Unfortunately, even these banalities would allow one to get inconsistencies without further restrictions. For example, using the logical negation operation, we can derive the Russell paradox about the set of all sets that do not contain themselves (think of $P\ x$ as $x \in P$ if preferred):

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

In other words, something is equal to its own logical negation! The problem seems to arise because no proper distinction of levels is made: x is treated both as a predicate and the argument to a predicate. Even if it didn't lead to inconsistency, one might argue that it looks a bit strange. Normally one likes to have a clear idea of what sort of mathematical object a term denotes — our explanation of currying, for example, leaned on the idea that addition is thought of as a function $\mathbb{R} \to (\mathbb{R} \to \mathbb{R})$.

Accordingly, Church (1940) augmented λ-calculus with a theory of types, simplifying Russell's system from Principia Mathematica (Whitehead and Russell 1910) and giving what is often called 'simple type theory'. HOL follows this system quite closely. Every term has a unique *type* which is either one of the basic types or the result of applying a type constructor to other types. The only basic type in HOL is initially the type of booleans `bool` and the only type operator is the function space constructor $\to$. (Many others are added later, as we shall see.) HOL extends Church's system by allowing also 'type variables' which give a form of polymorphism. Constants with polymorphic type are generic, and can have various types resulting from fixing the names of the type variables. For example, the equality relation has type $\alpha \to \alpha \to \mathit{bool}$ where α is a type variable. This means it can be used with any types, even if they themselves involve type variables, replacing α.

Just as in typed programming languages, functions may only be applied to arguments of the right type; only a function of type $f : \gamma \to \ldots$ may be applied to an argument of type γ.

For familiarity, types are written in a concrete syntax with some type constructors like $\to$ written infix. Just as with constant and variable terms, type variables and type constants are not distinguished syntactically: HOL's parser assumes that everything whose name corresponds to a constant is a constant, and every other identifier is a variable. However, it's customary to use names beginning with an uppercase letter for type variables, e.g. `A` and `State`. Examples of HOL types then, include *bool* and $A \to \mathit{bool}$ (where A is a type variable). We write $t : \gamma$ to indicate that a term t has type γ. Readers familiar with set theory may like to think of types as sets within which the objects denoted by the terms live, so $t : \gamma$ can be read as $t \in \gamma$. Note that the use of the colon is already standard in set theory when used for function spaces, i.e. one typically writes $f : A \to B$ rather than $f \in A \to B$.

5.3 Primitive inference rules

The HOL formal system allows the deduction of arbitrary *sequents* of the form $\phi_1, \ldots, \phi_n \vdash \psi$ (read as 'if ϕ_1 and ...and ϕ_n then ψ') where the terms involved have type *bool*. (Where there are no assumptions it is customary to write just $\vdash \psi$.) There are no additional logical constants involved in the basic deductive system. The derivable sequents are those that can be generated by the following inference rules. Each rule is written with the conclusion below a line and the hypotheses above, and with the standard name for the inference rule, corresponding in fact to a CAML identifier in HOL, at the right.

$$\frac{}{\vdash t = t}\ \texttt{REFL}$$

This rule says that equality is reflexive.

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash t = u}{\Gamma \cup \Delta \vdash s = u}\ \texttt{TRANS}$$

This rule says that equality is transitive. It is of course necessary to include in the conclusion theorem any assumption that may have played a role in deducing the top two theorems.

$$\frac{\Gamma \vdash s = t \quad \Delta \vdash u = v}{\Gamma \cup \Delta \vdash s(u) = t(v)}\ \texttt{MK\_COMB}$$

This says that equal functions applied to equal arguments give equal results. We have assumed without comment that the types agree, e.g. $s : \sigma \to \tau$, $t : \sigma \to \tau$, $u : \sigma$ and $v : \sigma$.

$$\frac{\Gamma \vdash s = t}{\Gamma \vdash (\lambda x.\, s) = (\lambda x.\, t)}\ \texttt{ABS}$$

This rule requires that x is not a free variable in any of the assumptions Γ. It says that if, without using any special properties of x, we deduced that two expressions involving x are equal, then the functions that take x to those values are equal.

$$\frac{}{\vdash (\lambda x.\, t)x = t}\ \texttt{BETA}$$

This expresses the fact that combination and abstraction are converse operations, i.e. 'the function that takes an argument x to t', applied to an argument x, gives t.

$$\frac{}{\{p\} \vdash p}\ \texttt{ASSUME}$$

This says simply that from any p we can deduce p. Of course, p must have type *bool*.

$$\frac{\Gamma \vdash p = q \quad \Delta \vdash p}{\Gamma \cup \Delta \vdash q}\ \texttt{EQ\_MP}$$

This connects equality with deduction, saying that if p and q are equal, and we can deduce p, then we can deduce q (from the appropriately combined assumptions).

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{(\Gamma - \{q\}) \cup (\Delta - \{p\}) \vdash p = q}\ \texttt{DEDUCT\_ANTISYM\_RULE}$$

This rule also connects equality and deduction, effectively saying that equality on the boolean type represents logical equivalence. Ignoring extra hypotheses for a moment, it says that if we can deduce p from q and q from p, then p and q are equal, under the accumulated assumptions.

$$\frac{\Gamma[x_1,\ldots,x_n] \vdash p[x_1,\ldots,x_n]}{\Gamma[t_1,\ldots,t_n] \vdash p[t_1,\ldots,t_n]}\ \texttt{INST}$$

This rule expresses the fact that variables are to be interpreted as schematic, i.e. if p is true for variables $x_1,\ldots,x_n$, then we can replace those variables by any terms of the same type and still get something true. Note that the substitution is also applied to all hypotheses.

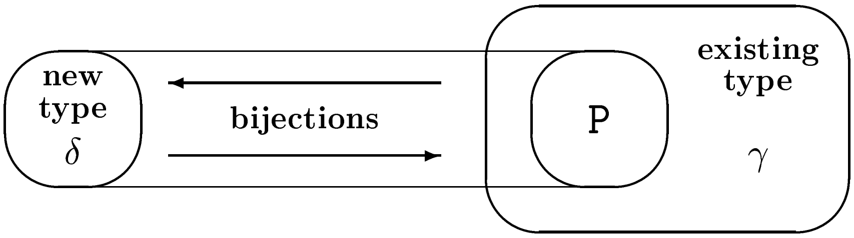
$$\frac{\Gamma[\alpha_1,\ldots,\alpha_n] \vdash p[\alpha_1,\ldots,\alpha_n]}{\Gamma[\gamma_1,\ldots,\gamma_n] \vdash p[\gamma_1,\ldots,\gamma_n]}\ \texttt{INST\_TYPE}$$

This is the same, but for substitution of type variables rather than term variables.

5.4 Definitions

All theorems in HOL are deduced using just the above rules, starting from a small set of *axioms*, which we will discuss shortly. Mathematics in HOL is derived just from these very basic axioms. However there is a special rule of definition, which allows the addition of new constants and corresponding new axioms provided they are purely definitional in character.[1] If $t : \tau$ is any term without free (term or type) variables, and $c : \tau$ an unused constant, then $c : \tau$ may be added to the stock of constants, and the axiom $\vdash c = t$ included as a theorem.

One can also define new types and type constructors in HOL. Given any subset of a type γ, marked out by its characteristic predicate $P : \gamma \to bool$, then given a theorem asserting that P is nonempty, one can define a new type δ (or type operator if γ contains type variables) in bijection with this set.



Both these definitional principles give a way of producing new mathematical theories without compromising soundness: one can easily prove that these principles are consistency-preserving. Effectively, constant definitions could be avoided simply by writing the definitional expansion out in full, while type definitions could be avoided by incorporating appropriate set constraints into theorems: rather than saying $\forall x : \delta.\ \ldots$ one could say $\forall y : \gamma.\ P(y) \Rightarrow \ldots$, with the appropriate isomorphic mappings.[2]

[1] From a logical point of view, we may say that HOL is actually an evolving sequence of logical systems, each a conservative extension of previous ones.

[2] In general, the logical core of HOL is reasonably *intuitionistic*, with classical principles introduced later as axioms. However the above definitional principle jars slightly with this since one of the type bijections is a *total* function $\gamma \to \delta$. This is at least weakly nonconstructive, allowing us for example to pass from $p \Rightarrow \exists x.\ q[x]$ to $\exists x.\ p \Rightarrow q[x]$.

5.5 Derived rules

HOL's logic is then built up by including constants for the usual logical operations. An attractive feature is that these do not need to be postulated: it has been known since Henkin (1963) how to define all logical constants in terms of equality, at least from a classical point of view. We do things in an 'intuitionistic' manner, giving useful deductive rules before we later assert the Law of the Excluded Middle, i.e. that every Boolean term is either true or false. While it is more typical (Prawitz 1965) to take a few additional logical constants such as $\forall$ and $\Rightarrow$ as primitive, our approach is very similar to the usual definitions of the internal logic of a topos; see e.g. Lambek and Scott (1986).

We will now show how all the logical constants are defined. These are $\top$ (true), $\wedge$ (and), $\Rightarrow$ (implies), $\forall$ (for all), $\exists$ (there exists), $\vee$ (or), $\bot$ (false) $\neg$ (not) and $\exists!$ (there exists a unique). Recall that what we write as $\forall x. P[x]$ is a syntactic sugaring of $\forall(\lambda x. P[x])$. Using this technique, quantifiers and the Hilbert ε operator can be used as if they bound variables, but with all binding implemented in terms of λ-calculus. There are several examples in this book.

$$
\begin{array}{rcl}
\top & = & (\lambda x.\, x) = (\lambda x.\, x) \\
\wedge & = & \lambda p.\, \lambda q.\, (\lambda f.\, f\ p\ q) = (\lambda f.\, f\ \top\ \top) \\
\Rightarrow & = & \lambda p.\, \lambda q.\, p \wedge q = p \\
\forall & = & \lambda P.\, P = \lambda x.\, \top \\
\exists & = & \lambda P.\, \forall Q.\, (\forall x.\, P(x) \Rightarrow Q) \Rightarrow Q \\
\vee & = & \lambda p.\, \lambda q.\, \forall r.\, (p \Rightarrow r) \Rightarrow (q \Rightarrow r) \Rightarrow r \\
\bot & = & \forall P.\, P \\
\neg & = & \lambda t.\, t \Rightarrow \bot \\
\exists! & = & \lambda P.\, \exists P \wedge \forall x.\, \forall y.\, P\ x \wedge P\ y \Rightarrow (x = y)
\end{array}
$$

While these might look puzzling at first sight, a little thought will convince the reader that they express what is intended. For example $\forall x.\, P[x]$, or without sugaring $\forall(\lambda x.\, P[x])$, says that for any a, $P[a]$, or equivalently $(\lambda x.\, P[x])\ a$, is true. This is exactly the same as saying that $\lambda x.\, P[x]$ is a constant function that always returns $\top$ (true), which is how $\forall$ is defined.

From the above definitions and the primitive rules, it is then possible to define derived inference rules that give convenient ways of manipulating logical formulas without explicitly taking everything back to the definitions. Because HOL is a programmable system in the LCF style, these can all be encapsulated as CAML functions that look to the user the same as primitive rules.

5.6 Classical axioms

That concludes the logic proper, and in fact quite a bit of interesting mathematics, e.g. infinitary inductive definitions, can be developed just from that basis (Harrison 1995). But for general use we adopt three more axioms.

- First, there is an axiom of extensionality, which we encode as an η-conversion theorem: $\vdash (\lambda x.\, t\ x) = t$.

- Secondly, we introduce one new primitive logical constant ε, of polymorphic type $(\alpha \to \mathit{bool}) \to \alpha$, the so-called Hilbert choice operator. It is accompanied by a new axiom giving the basic property of ε, namely that it picks out something satisfying P whenever there is something to pick:

$$\vdash \forall x.\, P(x) \Rightarrow P(\varepsilon x.\, P(x))$$

The intuitive reading of $\varepsilon x.\, P(x)$ is 'some x such that $P(x)$', which is an invaluable idiom when expressing some mathematical assertions. (Note that if there isn't anything satisfying $P(x)$, then $\varepsilon x.\, P(x)$ is still well-defined, but one can't prove any interesting properties of it.) However the above axiom isn't just an innocent convenience: it is a form of the Axiom of (global) Choice; since P can contain other variables, one can pass from $\forall x.\, \exists y.\, P[x, y]$ to $\forall x.\, P[x, \varepsilon y.\, P[x, y]]$. Rather surprisingly, it also makes the logic classical, i.e. allows us to prove the theorem $\vdash \forall p.\, p \vee \neg p$; see Beeson (1984) for the proof we use.

- Finally we introduce a new type `ind` of 'individuals', and add an axiom of infinity, asserting that the type *ind* is infinite. The Dedekind/Peirce definition of 'infinite' is used:

$$\begin{array}{ll} \vdash \exists f : ind \rightarrow ind. & (\forall x_1, x_2.\, (f(x_1) = f(x_2)) \Rightarrow (x_1 = x_2)) \wedge \\ & \neg(\forall y.\, \exists x.\, y = f(x)) \end{array}$$

  That is, we assert the existence of a function from the type of individuals to itself that is injective but not surjective. Such a mapping is impossible if the type is finite, since it would entail that it can be put into 1-1 correspondence with a proper subset of itself.

From that simple foundation, all the HOL mathematics and applications, including those described here, is developed by definitional extension.

Chapter 6

Implementation in CAML

The above description of HOL's logical basics abstracted away somewhat from its actual realization in CAML. However it has a fairly direct realization as three CAML types to represent HOL types, terms and theorems. (Note the object-meta distinction here: one has a CAML type of data structures representing HOL types.) These CAML types are all treated as abstract, with members only being created via special interface functions.[1] This guards against construction of meaningless types (e.g. using undefined type constructors), ill-typed terms, and theorems that have not been proved using the primitive rules.

6.1 Types

Each HOL type is either a type variables, or a type constructor applied to other types. Primitive types like `bool` are treated as nullary constructors, i.e. constructors with no arguments. We will now show some of the most useful CAML functions for manipulating types.

`get_type_arity :string -> int` finds the arity of the appropriately-named type constructor. If there is no type constructor with that name, it fails. For example:

```
#get_type_arity "bool";;
it : int = 0
#get_type_arity "fun";;
it : int = 2
#get_type_arity "con";;
Uncaught exception: Failure "find"
```

The rest of the functions divide nicely into three groups: those for creating HOL types, those for breaking them apart, and those for testing their structure.

`mk_vartype :string -> hol_type` creates a type variable with the requested name. This is permissible even if there is also a type constant of that name, but can look confusing. For example:

```
#mk_vartype "A";;
it : hol_type = `:A`
#mk_vartype "bool";;
it : hol_type = `:bool`
```

[1] This could actually be enforced by the CAML system by separately compiling the modules.

`mk_type :string * hol_type list -> hol_type` creates a composite type given the name of a type constructor and a list of component types of the right length. It fails if the name is not that of a constructor, or if the constructor's arity doesn't match the length of the list.

```
#mk_type("bool",[mk_vartype "A"]);;
Uncaught exception: Failure "mk_type: wrong number of arguments to bool"
#mk_type("bool",[]);;
it : hol_type = `:bool`
#mk_type("fun",[it; it]);;
it : hol_type = `:bool->bool`
```

`dest_vartype :hol_type -> string` reverses the effect of `mk_vartype`, i.e. takes a type variable and returns its name. It fails if the type isn't a type variable.

```
#dest_vartype `:A`;;
it : string = "A"
#dest_vartype `:bool`;;
Uncaught exception: Failure "dest_vartype: type constructor not a variable"
#dest_vartype (mk_vartype "bool");;
it : string = "bool"
```

Analogously, `dest_type :hol_type -> string * hol_type list` reverses the effect of `mk_type`, and fails if given a type variable.

```
#dest_type `:bool`;;
it : string * hol_type list = "bool", []
#dest_type `:A`;;
Uncaught exception: Failure "dest_type: type variable not a constructor"
#dest_type `:bool->bool`;;
it : string * hol_type list = "fun", [`:bool`; `:bool`]
```

The functions `is_type :hol_type -> bool` and `is_vartype :hol_type -> bool` test whether a HOL type is a composite type or a type variable respectively.

6.2 Terms

The CAML function `get_const_type :string -> hol_type` finds the type of the appropriately-named constant, or fails if there is no constant of that name. Some constants have *polymorphic* type, meaning a type including type variables. Such a constant can have any type that arises from replacing the component type variables consistently by other types. For example the equality constant is a curried operator of two arguments, but the types of the arguments are arbitrary, provided they are the same:

```
#get_const_type "=";;
it : hol_type = `:A->(A->bool)`
```

In such cases, the type returned by `get_const_type` is a *most general* type, and can be specialized by setting type variables appropriately. In general, terms feature instances of polymorphic constants. The type of an arbitrary term can by found using `type_of :term -> hol_type`, e.g.

```
#type_of `x:A`;;
it : hol_type = `:A`
#type_of `x = x`;;
Warning: inventing type variables
it : hol_type = `:bool`
```

By analogy with HOL types, the rest of the functions divide nicely into those for creating HOL terms, those for breaking them apart, and those for testing their structure.

`mk_var :string * hol_type -> term` creates a HOL variable with the chosen name and type.

```
#mk_var("x",mk_vartype "A");;
it : term = `x`
#type_of it;;
it : hol_type = `:A`
#mk_var("p",`:bool`);;
it : term = `p`
```

`mk_const :string * (hol_type * hol_type) list -> term` is the analogous constructor for HOL constants, but it's a bit more complicated to use. The second argument indicates not the desired type, but rather a list of settings for the type variables in order to attain that type. For example:

```
#mk_const("=",[]);;
it : term = `(=)`
#type_of it;;
it : hol_type = `:A->(A->bool)`
#mk_const("=",[`:bool`,`:A`]);;
it : term = `(=)`
#type_of it;;
it : hol_type = `:bool->(bool->bool)`
```

There is an alternative function `mk_mconst :string * hol_type -> term` which works out the instantiations itself. However it is not part of the logical core, relying as it does on higher-level functions to match up types. It will fail if the desired type cannot be realized:

```
#mk_mconst("=",`:bool->bool->bool`);;
it : term = `(=)`
#type_of it;;
it : hol_type = `:bool->(bool->bool)`
#mk_mconst("=",`:A->B->C`);;
Uncaught exception: Failure "mk_const: generic type cannot be instantiated"
```

`mk_comb : term * term -> term` creates an application; it is given two terms, one a function and one an argument, and tries to create the corresponding application term, failing if the types don't match up.

```
#mk_comb(`P:A->bool`,`x:A`);;
it : term = `P x`
#mk_comb(`P:A->bool`,`x:B`);;
Uncaught exception: Failure "mk_comb: types do not agree"
```

`mk_abs :term * term -> term` creates an abstraction term, given a variable to abstract over and the term to act as body. It fails if the first term argument isn't a variable.

```
#mk_abs(`x:A`,`x:A`);;
it : term = `\x. x`
#mk_abs(it,it);;
Uncaught exception: Failure "mk_abs: not a variable"
```

There are now analogous destructor functions `dest_var`, `dest_const`, `dest_comb` and `dest_abs` that act as inverses to the above. Strictly speaking `dest_const` is an inverse to `mk_mconst`, since it returns the constant name and type, not the instantiation list. Similarly, there are discriminator functions `is_var`, `is_const`, `is_abs` and `is_comb` to test whether a term is in each class.

```
#dest_comb `~p`;;
it : term * term = `(~)`, `p`
#dest_comb `\p. ~p`;;
Uncaught exception: Failure "dest_comb: not a combination"
#dest_abs `\p. ~p`;;
it : term * term = `p`, `~p`
#is_var `x:A`;;
it : bool = true
#is_var `~p`;;
it : bool = false
```

As well as the primitive syntax operations on terms, there are various derived ones, which avoid the need to reduce everything right down to the basic operations above. For example, `rator` and `rand` (the names established lambda-calculus jargon) take respectively the operator and operand of an application, i.e. return respectively `f` and `x` when applied to a term `f x`. They can be implemented just by applying `dest_comb` to get a pair of terms, then applying the CAML functions `fst` or `snd`:

```
#let rator tm = fst(dest_comb tm);;
rator : term -> term = <fun>
#let rand tm = snd(dest_comb tm);;
rand : term -> term = <fun>
#rand `SUC 2`;;
it : term = `2`
#rator `1 + 2`;;
it : term = `(+) 1`
#rand `1 + 2`;;
it : term = `2`
```

There are also derived functions to create, break apart and test for equations:

```
#dest_eq `x = 1`;;
it : term * term = `x`, `1`
#is_eq `x = y + 3`;;
it : bool = true
#is_eq `x <= y + 3`;;
it : bool = false
#is_eq `p = q`;;
it : bool = true
#mk_eq(`T`,`F`);;
it : term = `T = F`
```

Similarly, when the other constants are defined, they often have a corresponding set of functions to create, test, and destroy them. For example, `mk_imp` creates an implication `p ==> q`, `dest_conj` breaks apart a conjunction `p /\ q`, and `is_disj` tests if a term is a disjunction `p \/ q`.

6.3 Theorems

HOL theorems can be taken apart into a list of assumptions and a conclusion using the function `dest_thm :thm -> term list * term`. The hypotheses and conclusion can be grabbed separately using `hyp` and `concl`. However they can only be created by using one of the primitive rules, making a term or type definition, or finally asserting an axiom. The last of these is only done three times for the basic mathematical axioms, and thereafter HOL users are discouraged from adding new axioms, as this does not maintain the guarantee of consistency. The primitive inference rules were listed earlier, and their CAML realizations are simply CAML functions returning something of type `thm`. For example:

```
#BETA `(\p. ~p) p`;;
it : thm = |- (\p. ~p) p = ~p
#INST [`q:bool`,`p:bool`] it;;
it : thm = |- (\p. ~p) q = ~q
#TRANS (ASSUME `p:bool = q`) (ASSUME `q:bool = r`);;
it : thm = p = q, q = r |- p = r
#dest_thm it;;
it : term list * term = [`p = q`; `q = r`], `p = r`
```

New definitions are made using the function `new_definition`, which takes an equational term '`c = t`', where `c` is a variable. The system introduces a new constant called `c` and returns the theorem `|- c = t` for the new constant. For example:

```
#new_definition `true = T`;;
it : thm = |- true = T
```

Later on, more convenient derived definitional principles are built on top of this — even `new_definition` is bound to a more powerful derived function that can, for example, accept function definitions in the form '`f x1 ... xn = ...`'.

The primitive function for performing type definitions is `new_basic_type_definition`. The user gives the desired name for the new type and for the bijections that map between the old and new types, and finally a theorem asserting that the chosen subset of the existing type contains some object. For example, we can define a new type `single` in bijection with the 1-element subset of `bool` containing just `T`. The appropriate predicate is the function that asks of its argument `x` whether it is equal to `t`, i.e. `\x. x = T`:

```
#let th1 = BETA_CONV `(\x. x = T) T`;;
th1 : thm = |- (\x. x = T) T = T = T
#let th2 = EQ_MP (SYM th1) (REFL `T`);;
th2 : thm = |- (\x. x = T) T
#new_basic_type_definition "single" ("mk_single","abs_single") th2;;
it : thm * thm =
 |- mk_single (abs_single a) = a,
 |- (\x. x = T) r = abs_single (mk_single r) = r
```

Two theorems are returned as an ML pair, which together imply that the chosen bijections map 1-1 between the new type and the chosen subset of the old one.

6.4 Some predefined constants

HOL has a large number of constants predefined. The most basic of these are the logical operators whose definitions were given in passing above. Here is a table showing the conventional logical symbols, HOL's ASCII approximation, and the English reading. In the concrete syntax, they bind according to their order in the above table, negation being strongest and the variable-binding operations weakest.

$\bot$	`F`	Falsity
$\top$	`T`	Truth
$\neg$	`~`	Not
$\wedge$	`/\`	And
$\vee$	`\/`	Or
$\Rightarrow$	`==>`	Implies
$\equiv$	`=`	If and only if
$\forall$	`!`	For all
$\exists$	`?`	There exists
$\exists!$	`?!`	There exists a unique
ε	`@`	Some ... such that
λ	`\`	The function taking ... to

Readers are no doubt used to writing symbols like `+` rather than the word 'plus', but may well find these analogous logical operations less familiar. However, it's worth spending some time getting accustomed to them, since they are needed to understand most HOL terms. Here are a few examples:

- `T` says 'truth holds'.
- `F ==> p` says 'if falsity holds, so does any p'.
- `!x. x > 0 = (?y. x = y + 1)` says 'for all x, x is greater than zero if and only if there exists a y such that $x = y + 1$'.
- `x >= y /\ u > v ==> x + u > y + v` says 'if x is greater than or equal to y and u is greater than v, then $x + u$ is greater than $y + v$'.
- `p /\ q ==> q \/ r` says 'if p and q are true, then either q or r is true'.
- `~(p = ~p)` says 'it is always false that p holds if and only if p does not hold'.
- `(m * n = 0) = (m = 0) \/ (n = 0)` says 'mn is zero if and only if either m is zero or n is zero'.
- `(\x. x + 1) 3 = 4` says that the function mapping any x to $x + 1$, when applied to the argument 3, is equal to 4.
- `(?!x. P x) ==> !a. P(a) = (a = @x. P x)` says 'if there is a unique x satisfying P, then for all a, P holds of a if and only if a is equal to some canonical x satisfying P'.
- `!P. (!n. (!m. m < n ==> P m) ==> P n) ==> !n. P n` expresses the principle of complete mathematical induction, i.e. 'for every predicate P over numbers, if for each n, whenever P holds for each smaller m, then P holds for n, then for every n, P holds'.

There are also a lot of constants defined in mathematical theories. Most of these should look familiar, and in any case are summarized in a later chapter. However, the following is a list of some of the less obvious ones, which may help the reader follow some of the examples below.

HOL notation	Standard symbol	Meaning
`SUC n`	$n+1$	Successor of n
`#`	(*none*)	Natural map $\mathbb{N} \to \mathbb{R}$ or $\mathbb{N} \to \mathbb{Z}$
`--x`	$-x$	Unary negation of x
`inv(x)`	x^{-1}	Multiplicative inverse of x
`abs(x)`	$\lvert x \rvert$	Absolute value of x
`m EXP n`	m^n	Natural m raised to natural power n
`x pow n`	x^n	Real x raised to natural number power n
`root n x`	$\sqrt[n]{x}$	Positive n^{th} root of x
`Sum(n,d) f`	$\Sigma_{i=n}^{n+d-1} f(i)$	Sum of d terms $f(i)$ starting with $f(n)$
`x IN s`	$x \in s$	x is a member of set s
`EMPTY`	$\emptyset$	The empty set
`UNIV`	*none*	Universe set for a type
`x INSERT s`	$\{x\} \cup s$	Set s with element x
`s DELETE x`	$s - \{x\}$	Set s without element x
`s UNION t`	$s \cup t$	Union of sets s and t
`s INTER t`	$s \cap t$	Intersection of sets s and t
`s DIFF t`	$s - t$	Difference of sets s and t
`UNIONS s`	$\bigcup s$	Union of all members of s
`INTERS s`	$\bigcap s$	Intersection of all members of s

Formally, naturals, integers and reals are all different types, hence the use of a mapping `#` between then. The usual arithmetic operations like `+` are overloaded, meaning that they are used for addition of reals, integers, and natural numbers. (The main exception is that `EXP` is used for natural numbers.) The next chapter explains the translation from the usual symbols to different constants under the surface.

Chapter 7

Parsing and printing

We have already used the automatic quotation parsers quite extensively, and it's time we looked at the relationship between the underlying representations and the surface syntax in more detail. Many convenient constructs are representing using some special constants inside HOL, and the parser and printer transform such internal representations into more palatable surface syntax. For example the conditional expression

```
if b then e1 else e2
```

is represented inside the logic using a constant `COND`:

```
CONS b e1 e2
```

Various other handy syntactic constructs are also dealt with in this way, e.g. abstractions over non-variables, and let-terms. For example

```
\(x,y,z). x + y + z
```

is represented by:

```
GABS (\f. !x y z. GEQ (f (x,y,z)) (x + y + z))
```

and

```
let x = 1 and y = 2 in x + y
```

is represented by:

```
LET (\x y. LET_END (x + y)) 1 2
```

Apart from special case like these, the parser-printer transformations are pretty straightforward. Identifiers may be declared infix, and given a precedence and associativity (right or left) using `parse_as_infix`. Here are a few genuine examples from the source code:

```
parse_as_infix("<",(12,"right"));;

parse_as_infix("+",(16,"right"));;

parse_as_infix("-",(18,"left"));;

parse_as_infix("IN",(11,"right"));;

parse_as_infix("UNION",(16,"right"));;
```

7.1 Overloading

The parser and printer allow front-end symbols to be overloaded, and tries to resolve ambiguities by exploiting type information. Before a symbol can be overloaded, it must be given a most general type, and any term it maps to must have a type that is an instance of this type. During typechecking, the overloaded symbol is given its most general type. If the typechecking process fixes the type sufficiently to disambiguate, then the appropriate target is picked. Otherwise some instance is defaulted, and typechecking repeated until all symbols have been resolved. For example, the addition symbol is made overloadable:

```
make_overloadable "+" `:A->A->A`;;
```

Now in order to make "+" overloaded to natural number, integer and real addition, we do:

```
overload_interface ("+",`(+):num->num->num`);

overload_interface ("+",`int_add:int->int->int`);

overload_interface ("+",`real_add:real->real->real`);
```

Now the symbol + will map to one of three terms in the underlying representation, decided according to type. The default chosen is always the most recently declared version, real addition after the above sequence. If the user wants to avoid any defaults, then type information sometimes needs to be supplied. All the following are unambiguous:

```
#`x + 1`;;
it : term = `x + 1`
#`x:int + y`;;
it : term = `x`
#`(x + y):real`;;
it : term = `x + y`
```

Instead of mapping a symbol to multiple targets, one can always choose just one. The function `override_interface` is similar to `overload_interface`, except that it removes any existing mappings for the symbol first. For example, the user who dislikes the use of equality to mean logical equivalence could remap HOL Light's interface as follows:

```
#parse_as_infix("<=>",(2,"right"));;
it : unit = ()
#override_interface ("<=>",`(=):bool->bool->bool`);;
it : unit = ()
#`x = F`;;
it : term = `x <=> F`
#`x <=> F`;;
it : term = `x <=> F`
```

Chapter 8

Conversions

A *conversion* in HOL is a derived rule of type `term -> thm` that when given a term `t`, always returns (assuming it doesn't fail) a theorem of the form `|- t = t'`. Conversions were introduced into Cambridge LCF by Paulson (1983), who showed that they gave a convenient and regular way of implementing many handy derived rules. Conversions can be considered as transforming a term into an equal one, and also giving a theorem to justify this equality. They are therefore useful as building-blocks for larger transformations, similarly justified.

HOL has a variety of built-in conversions, and they often have names ending in `CONV` as a reminder that they are conversions. Rather trivially, for example, the primitive inference rule `REFL` is a conversion, which takes a term `t` and returns a theorem `|- t = t`. If we think of conversions as transforming one term to another, `REFL` is a sort of 'identity' conversion. In fact, for this reason, it is given a new name `ALL_CONV`, since it is a conversion that always, trivially, works on any term. Its converse, in a sense, is a conversion `NO_CONV` which always fails:

```
#let (ALL_CONV:conv) = REFL;;
ALL_CONV : conv = <fun>
#let (NO_CONV:conv) = fun tm -> failwith "NO_CONV";;
NO_CONV : conv = <fun>
```

A slightly more interesting conversion is `BETA_CONV`, which performs a beta reduction step on terms of the form `(\x. ...) t`:

```
#BETA_CONV `(\x. x + 1) 2`;;
it : thm = |- (\x. x + 1) 2 = 2 + 1
```

There are also some conversions special to particular theories. For example there is a conversion `NUM_RED_CONV` to evaluate the result of an arithmetic operation on two numerals:

```
#NUM_RED_CONV `2 * 2`;;
it : thm = |- 2 * 2 = 4
#NUM_RED_CONV `2 EXP 10`;;
it : thm = |- 2 EXP 10 = 1024
#NUM_RED_CONV `100 DIV 7`;;
it : thm = |- 100 DIV 7 = 14
```

8.1 Conversionals

These conversions are building blocks. The mechanism for building them up is a suite of higher order functions called 'conversionals' or 'conversion combining operators'. These allow one to construct composite conversions in a user-defined way. For example, the conversional THENC, used infix, uses one conversion and then afterwards, another, e.g.

```
#(BETA_CONV THENC NUM_RED_CONV) `(\x. x + 1) 2`;;
it : thm = |- (\x. x + 1) 2 = 3
```

The conversional REPEATC allows one to use a conversion repeatedly until it fails (maybe zero times), e.g.

```
#REPEATC BETA_CONV `23`;;
it : thm = |- 23 = 23
#REPEATC BETA_CONV `(\x. x + 1)`;;
it : thm = |- (\x. x + 1) = (\x. x + 1)
#REPEATC BETA_CONV `(\x. x + 1) 2`;;
it : thm = |- (\x. x + 1) 2 = 2 + 1
#REPEATC BETA_CONV `(\x. (\y. x + y) 2) 1`;;
it : thm = |- (\x. (\y. x + y) 2) 1 = 1 + 2
```

8.2 Depth conversions

The conversions above are still only applied at the top level of a term. For example, the following fails because the beta-redex is deeper inside the term than expected:

```
#BETA_CONV `1 + (\x. x + 1) 2`;;
Uncaught exception: Failure "BETA_CONV: Not a beta-redex"
```

However there is an additional set of conversionals that apply the given conversion at depth inside the term. For example ONCE_DEPTH_CONV applies a conversion to the first applicable term(s) encountered in a top-down traversal of the term. No deeper terms are examined, but several terms can be converted provided they are disjoint:

```
#ONCE_DEPTH_CONV NUM_RED_CONV `1 + (2 + 3)`;;
it : thm = |- 1 + 2 + 3 = 1 + 5
#ONCE_DEPTH_CONV NUM_RED_CONV `(1 + 1) * (1 + 1)`;;
it : thm = |- (1 + 1) * (1 + 1) = 2 * 2
```

Conversions like NUM_RED_CONV can be used to reduce a term completely by applying it in a single bottom-up sweep. This is done by the conversional DEPTH_CONV, e.g.

```
#DEPTH_CONV NUM_RED_CONV `7 * (3 EXP 10) + 11`;;
it : thm = |- 7 * 3 EXP 10 + 11 = 413354
```

However, this isn't always what's needed; sometimes the act of applying a conversion at one level can create new applicable terms lower down; in this case DEPTH_CONV will not reexamine them. Two other conversionals, TOP_DEPTH_CONV and REDEPTH_CONV, will keep applying conversions as long as possible all over the term.

```
#DEPTH_CONV BETA_CONV `(\f x. f x) (\y. y + 1)`;;
it : thm = |- (\f x. f x) (\y. y + 1) = (\x. (\y. y + 1) x)
#REDEPTH_CONV BETA_CONV `(\f x. f x) (\y. y + 1)`;;
it : thm = |- (\f x. f x) (\y. y + 1) = (\x. x + 1)
#TOP_DEPTH_CONV BETA_CONV `(\f x. f x) (\y. y + 1)`;;
it : thm = |- (\f x. f x) (\y. y + 1) = (\x. x + 1)
#TOP_DEPTH_CONV NUM_RED_CONV `7 * (3 EXP 10) + 11`;;
it : thm = |- 7 * 3 EXP 10 + 11 = 413354
```

The difference is that the main sweeps are respectively top-down and bottom-up, which can lead to one or the other being preferable, mainly for efficiency reasons, in some situations. `TOP_DEPTH_CONV` is the default for HOL's rewriting, described in a later chapter.

The conversionals all have fairly straightforward definitions using HOL's primitive and derived equality rules. For example, `THENC` just needs to apply the conversion to a term, getting a theorem, then take the right-hand side of this theorem's conclusion, apply the second conversion to that and then link the equations together using the primitive inference rule `TRANS`. One could write an equivalent function as:

```
#let THENC' conv1 conv2 t =
   let th1 = conv1 t in
   let th2 = conv2 (rand(concl th1)) in
   TRANS th1 th2;;
THENC' : ('a -> thm) -> (term -> thm) -> 'a -> thm = <fun>
```

The depth conversionals can be implemented by a recursive traversal of the term, using primitive rules like `MK_COMB` to lift the equational theorems up to the whole term. In fact, the implementations are a bit more sophisticated because they are careful to avoid creating trivial equations unless needed.

Chapter 9

Derived rules

HOL has a variety of other derived rules that are not conversions, or at least aren't used much as in the previous chapter. Here we cover some of the most basic ones.

9.1 Logical rules

All the logical constants are defined; we have seen the definitions above. From the definitions, rules for manipulating them directly are derived, so for most purposes users can forget that they aren't primitives. Most of the rules are so-called introduction and elimination rules of natural deduction (Prawitz 1965).[1] For example, the introduction rule for conjunctions, `CONJ`, takes two theorems and gives a new theorem that results from conjoining ('anding') them, e.g.

```
#CONJ (REFL `1`) (ASSUME `x = 2`);;
it : thm = x = 2 |- (1 = 1) /\ (x = 2)
```

Conversely, the elimination rules `CONJUNCT1` and `CONJUNCT2` take a theorem with a conjunction as conclusion, and give new theorems for the left and right arms. `CONJ_PAIR` gives a pair of both, while `CONJUNCTS` repeatedly breaks down a conjunctive theorem into a list of theorems.

```
#let th1 = CONJ (REFL `T`) (ASSUME `p /\ q`);;
th1 : thm = p /\ q |- (T = T) /\ p /\ q
#let th2 = CONJ (REFL `1`) th1;;
th2 : thm = p /\ q |- (1 = 1) /\ (T = T) /\ p /\ q
#CONJ_PAIR th2;;
it : thm * thm = p /\ q |- 1 = 1, p /\ q |- (T = T) /\ p /\ q
#CONJUNCTS th2;;
it : thm list = [p /\ q |- 1 = 1; p /\ q |- T = T; p /\ q |- p; p /\ q |- q]
#CONJUNCT2 th1;;
it : thm = p /\ q |- p /\ q
```

Abstracting away a bit from the implementation in CAML, we can represent the rules in the usual form as:

$$\frac{\Gamma \vdash p \quad \Delta \vdash q}{\Gamma \cup \Delta \vdash p \land q}\ \texttt{CONJ}$$

[1] Although HOL uses a sequent presentation, the conventional derived rules are natural deduction rules, i.e. introduction and elimination on the right, rather than left and right introduction.

$$\frac{\Gamma \vdash p \wedge q}{\Gamma \vdash p} \text{ CONJUNCT1}$$

$$\frac{\Gamma \vdash p \wedge q}{\Gamma \vdash q} \text{ CONJUNCT2}$$

All the other defined constants come equipped with a similar suite of rules. In most cases the reader will be able to guess how the corresponding CAML function is used, and can experiment a little on the lines of the above examples.

$$\frac{\Gamma \vdash p}{\Gamma \vdash p = \top} \text{ EQT\_INTRO}$$

$$\frac{\Gamma \vdash p = \top}{\Gamma \vdash p} \text{ EQT\_ELIM}$$

$$\frac{\Gamma \vdash p \Rightarrow q \quad \Delta \vdash p}{\Gamma \cup \Delta \Rightarrow q} \text{ MP}$$

$$\frac{\Gamma \vdash q}{\Gamma - \{p\} \vdash p \Rightarrow q} \text{ DISCH}$$

$$\frac{\Gamma \vdash p \Rightarrow q}{\Gamma \cup \{p\} \vdash q} \text{ UNDISCH}$$

$$\frac{\Gamma \vdash \forall x.\, p}{\Gamma \vdash p[t/x]} \text{ SPEC}$$

Here $p[t/x]$ denotes the result of substituting t for all free instances of x in p. HOL automatically renames variables to avoid capture if necessary, by adding prime characters. (This happens in the primitive function `INST` that is used in the implementation.)

```
#let th1 = ASSUME `!x. x >= 0`;;
th1 : thm = !x. x >= 0 |- !x. x >= 0
#let th2 = SPEC `y + 1` th1;;
th2 : thm = !x. x >= 0 |- y + 1 >= 0
#let th3 = ASSUME `!x. ?y. y > x`;;
th3 : thm = !x. ?y. y > x |- !x. ?y. y > x
#let th4 = SPEC `y:num` th3;;
th4 : thm = !x. ?y. y > x |- ?y'. y' > y
```

Note that the naive result of substituting would be the incorrect `?y. y > y`.

$$\frac{\Gamma \vdash p}{\Gamma \vdash \forall x.\, p} \text{ GEN}$$

This rule will fail if the variable x is free in the assumptions Γ. Again, this restriction arises naturally out of one in the underlying primitives, in this case in `ABS`.

```
#let th1 = REFL `x:num`;;
th1 : thm = |- x = x
#let th2 = GEN `x:num` th1;;
th2 : thm = |- !x. x = x
#let th3 = GEN `y:num` th2;;
th3 : thm = |- !y x. x = x
#let th4 = ASSUME `x = 2`;;
th4 : thm = x = 2 |- x = 2
#let th5 = GEN `x:num` th4;;
Uncaught exception: Failure "GEN"
#let th5 = GEN `y:num` th4;;
th5 : thm = x = 2 |- !y. x = 2
```

$$\frac{\Gamma \vdash p[t/x]}{\Gamma \vdash \exists x.\, p}\ \texttt{EXISTS}$$

The ML invocations for this rule are relatively complicated; the function requires the user to specify the desired form of the result and the term t to choose. It could work out the latter for itself, but in general one can derive many existential theorems from the same starting point, e.g.

```
#let th1 = REFL `1`;;
th1 : thm = |- 1 = 1
#let th2 = EXISTS(`?x. x = 1`,`1`) th1;;
th2 : thm = |- ?x. x = 1
#let th3 = EXISTS(`?x:num. x = x`,`1`) th1;;
th3 : thm = |- ?x. x = x
#let th4 = EXISTS(`?x:num. 1 = 1`,`23`) th1;;
th4 : thm = |- ?x. 1 = 1
```

$$\frac{\Gamma \vdash q}{\Gamma - \{p\} \vdash (\exists x.\, p) \Rightarrow q}\ \texttt{CHOOSE}$$

This rule requires that x is not free in q nor in any of the Γ besides p.

$$\frac{\Gamma \vdash p}{\Gamma \vdash p \vee q}\ \texttt{DISJ1}$$

$$\frac{\Gamma \vdash q}{\Gamma \vdash p \vee q}\ \texttt{DISJ2}$$

$$\frac{\Gamma \vdash r \quad \Gamma' \vdash r \quad \Delta \vdash p \vee q}{(\Gamma - \{p\}) \cup (\Gamma' - \{q\}) \cup \Delta \vdash r}\ \texttt{DISJ\_CASES}$$

$$\frac{\Gamma \vdash \neg p}{\Gamma \vdash p \Rightarrow \bot}\ \texttt{NOT\_ELIM}$$

$$\frac{\Gamma \vdash p \Rightarrow \bot}{\Gamma \vdash \neg p}\ \texttt{NOT\_INTRO}$$

$$\frac{\Gamma \vdash p \equiv \bot}{\Gamma \vdash \neg p}\ \texttt{EQF\_ELIM}$$

$$\frac{\Gamma \vdash \neg p}{\Gamma \vdash p \equiv \bot}\ \texttt{EQF\_INTRO}$$

9.2 Rewriting and simplification

HOL has various rules and conversions at a somewhat higher level. Some of the most useful of these automatically work out how to instantiate variables to apply to the case in hand. For example, the above 'Modus Ponens' rule requires the theorems to match up exactly:[2]

```
#MP (ASSUME `x < 1 ==> x <= 1`) (ASSUME `x < 1`);;
it : thm = x > 1 ==> x >= 1, x >= 1 |- x > 1
#MP (ASSUME `y < 1 ==> y <= 1`) (ASSUME `x < 1`);;
Uncaught exception: Failure "MP: theorems do not agree"
```

A more powerful rule, MATCH_MP, tries to work out settings for free or universally quantified variables in the first theorem in order to make things match up. we can illustrate this using a built-in theorem LT_IMP_LE:

```
#let th1 = LT_IMP_LE;;
th1 : thm = |- !m n. m < n ==> m <= n
#MATCH_MP th1 (ASSUME `x < 1`);;
it : thm = x < 1 |- x <= 1
```

A similar rule, actually a conversion, is REWR_CONV. It takes an equation, perhaps universally quantified, and sets the variables if possible so that the left-hand side matches the proffered term, 'rewriting' it. Again, we will illustrate it using a built-in theorem:

```
#let th1 = NOT_LE;;
th1 : thm = |- !m n. ~(m <= n) = n < m
#REWR_CONV th1 `~(x + 1 <= x)`;;
it : thm = |- ~(x + 1 <= x) = x < x + 1
```

Since it is a conversion, REWR_CONV can be combined with various depth conversions to rewrite repeatedly at various levels of a term. Built-in functions like REWRITE_CONV take a whole list of theorems, extract rewrites from them and repeatedly apply them to a term.[3] Moreover, they throw in a set of handy rewrites to get rid of trivial propositional clutter, e.g. reducing `p /\ p` to `p`. They are one of the workhorses in typical HOL proofs. If the additional propositional simplifications are not required, prefix the name with PURE:

```
#PURE_REWRITE_CONV[NOT_LE; LT_REFL] `~(x < x) \/ q`;;
it : thm = |- ~(x < x) \/ q = ~F \/ q
#REWRITE_CONV[NOT_LE; LT_REFL] `~(x < x) \/ q`;;
it : thm = |- ~(x < x) \/ q = T
```

As in this example, one often rewrites Boolean terms. In cases where conversions are applied to Boolean terms, it's often handy to convert conversions to forward inference rules. This is done using CONV_RULE, whose definition is simply:

```
#let CONV_RULE conv th =
  EQ_MP (conv(concl th)) th;;
CONV_RULE : (term -> thm) -> thm -> thm = <fun>
#CONV_RULE(REWRITE_CONV[NOT_LE; LT_REFL]) (ASSUME `~(x < x) \/ q`);;
it : thm = ~(x < x) \/ q |- T
```

[2]Actually, only up to alpha-equivalence, i.e. renaming of bound variables.

[3]They do work by applying REWR_CONV at depth, but are optimized using term nets to avoid too many wasteful attempts to match theorems against subterms.

Some conversions are made into rules and given names, because they are used so often. For example:

```
#let BETA_RULE = CONV_RULE(REDEPTH_CONV BETA_CONV);;
BETA_RULE : thm -> thm = <fun>
#let REWRITE_RULE thl = CONV_RULE(REWRITE_CONV thl);;
REWRITE_RULE : thm list -> thm -> thm = <fun>
```

Still more powerful than rewriting is simplification. This allows the use of equations that are conditional, i.e. of the form $\vdash p \Rightarrow l = r$. After matching up l with the term if possible, setting the theorem to $\vdash p' \Rightarrow l' = r'$ the conversion is recursively applied to the hypothesis p', trying to reduce it to $\top$ and so eliminate it. This can often avoid tedious chains of straightforward logical reasoning. For example, in

```
#DIV_LT;;
it : thm = |- !m n. m < n ==> (m DIV n = 0)
#SIMP_CONV[DIV_LT; ARITH] `3 DIV 7 = 0`;;
it : thm = |- (3 DIV 7 = 0) = T
```

the built-in theorem `DIV_LT` is used as a rewrite, giving a hypothesis `3 < 7` which is then attacked by more simplification, this time using a set of rewrites to do basic arithmetic (described later). Simplification also accumulates context, so when traversing a term $p \Rightarrow q$ and descending to q, additional rewrites are derived from p, e.g.

```
#SIMP_CONV [] `p /\ q ==> p`;;
it : thm = |- p /\ q ==> p = T
```

The rewrite `p = T` is extracted from the context `p` and this is used to rewrite the consequent to `T`. The final result follows from an additional rewrite with the built-in simplification `p ==> T = T`.

9.3 Ordered rewriting

It is possible for rewriting and simplification to go into an infinite loop, e.g. applying two successive rewrites $\vdash s = t$ and $\vdash t = s$ alternately. However, HOL tries to avoid looping in some cases, ignoring rewrites that would loop:

```
#REWRITE_CONV[ASSUME `x = x + 1`] `x:num`;;
Warning: discarding looping rewrite
it : thm = |- x = x
```

Some rewrites are said to be *permutative*, meaning that the left hand side can be matched to the right hand side and vice versa. For example, there is a built-in theorem `ADD_SYM` asserting that addition of natural numbers is commutative, and several others:

```
#ADD_SYM;;
it : thm = |- !m n. m + n = n + m
#CONJ_SYM;;
it : thm = |- !t1 t2. t1 /\ t2 = t2 /\ t1
#INSERT_COMM;;
it : thm = |- !x y s. x INSERT y INSERT s = y INSERT x INSERT s
```

The HOL Light approach to permutative rewrite rules has long been used in the Boyer-Moore theorem prover, and more recently in Isabelle thanks to Tobias Nipkow. They are only applied if, after instantiation, the left-hand side is "larger" than the right according to some well-founded ordering. The basic building block is `ORDERED_REWR_CONV`. This calls `REWR_CONV`, but will then force failure unless in the resulting theorem $\Gamma \vdash s' = t'$ one has $t' > s'$ according to the given ordering $>$ on terms. In this way, one can rewrite freely with a theorem such as $\vdash x + y = y + x$ without fear of infinite looping.

However in conjunction with other rewrites, infinite looping can reappear. For example, rewriting with the above commutative law and the associative law $\vdash (x + y) + z = x + (y + z)$ one could still have an infinite rewrite:

$$
\begin{array}{rcl}
x + (y + z) & \longrightarrow & (y + z) + x \\
& \longrightarrow & y + (z + x) \\
& \longrightarrow & (z + x) + y \\
& \longrightarrow & z + (x + y) \\
& \longrightarrow & (x + y) + z \\
& \longrightarrow & x + (y + z)
\end{array}
$$

This, however, can be prevented by a suitable choice of ordering. In fact, given the right ordering, the associative and commutative laws together not only always terminate, but actually reduce AC combinations to their canonical form. Martin and Nipkow (1990) give a slightly tricky ordering that makes the associative and commutative laws alone give a normalizer. However a more obvious approach is to add a third theorem, easily derived from the other two: $\vdash x + (y + z) = y + (x + z)$. Now, suppose that the ordering has the following properties for any terms x, y and z:

$$
\begin{array}{rcll}
(x + y) + z & > & x + (y + z) & \\
x + y & > & y + x & \text{iff } x > y \\
x + (y + z) & > & y + (x + z) & \text{iff } x > y
\end{array}
$$

Such an ordering, if it is also monotonic (if $s > t$ then $u[s] > u[t]$) and total and is wellfounded on ground terms, is said to be *AC-compatible*. Intuitively it is clear that ordered rewriting with these theorems will canonicalize AC combinations by 'bubbling' terms in iterated additions to their proper place. Theorems in this class for some associative and commutative operators are built into HOL, e.g.

```
#ADD_AC;;
it : thm =
 |- (m + n = n + m) /\ ((m + n) + p = m + n + p) /\ (m + n + p = n + m + p)
#MULT_AC;;
it : thm =
 |- (m * n = n * m) /\ ((m * n) * p = m * n * p) /\ (m * n * p = n * m * p)
#REWRITE_CONV[ADD_AC; MULT_AC] `x * y + z * x + w * x + x * w =
                                x * w + x * z + y * x + x * w`;;
it : thm =
 |- (x * y + z * x + w * x + x * w = x * w + x * z + y * x + x * w) = T
```

Martin and Nipkow (1990) show that one can also add laws of left and right distributivity for $+$ and $*$, as well as idempotence laws $\vdash x+x = x$ and $\vdash x+(x+y) = x+y$ and get canonicalizers under these laws too. (For example, if $+$ is conjunction or disjunction.)

```
#CONJ_ACI;;
it : thm =
 |- (p /\ q = q /\ p) /\
    ((p /\ q) /\ r = p /\ q /\ r) /\
    (p /\ q /\ r = q /\ p /\ r) /\
    (p /\ p = p) /\
    (p /\ p /\ q = p /\ q)
#REWRITE_CONV[CONJ_ACI] `p /\ q /\ p /\ r /\ q = r /\ q /\ p`;;
it : thm = |- (p /\ q /\ p /\ r /\ q = r /\ q /\ p) = T
```

9.4 Higher order matching

HOL Light supports a limited form of higher order matching. This is immensely useful in order to express more general term transformations as rewrite rules. If only simple 'first order' matching is used, the scope of rewriting, matching modus ponens etc. is rather restricted. Even quite simple schematic theorems need to be instantiated manually — a very tedious task. For example, if we want to use the theorem:

```
#SKOLEM_THM;;
it : thm = |- !P. (!x. ?y. P x y) = (?y. !x. P x (y x))
```

to rewrite the term `!n. ?m. m EXP 2 <= n /\ n < (SUC m) EXP 2`, then simple rewriting won't work; one first needs to instantiate the theorem with

```
P = (\n m. m EXP 2 <= n /\ n < (SUC m) EXP 2)
```

then beta-reduce it, and only then rewrite with it. HOL Light will do this automatically in some situations. For example, it will perform the following rewrite, even though the term isn't literally an instance of the theorem's left hand side:

```
#NOT_FORALL_THM;;
it : thm = |- !P. ~(!x. P x) = (?x. ~P x)
#REWR_CONV NOT_FORALL_THM `~(!n. n < n - 1)`;;
it : thm = |- ~(!n. n < n - 1) = (?n. ~(n < n - 1))
```

The implementation of higher order matching aims to make matching as general as possible while keeping it deterministic. It allows higher order matches of $P\ x_1 \cdots x_n$ where P is an instantiable variable, but only if it can decide with certainty how to instantiate the x_i. Generally speaking, there are lots of possible higher order matches; for example the pattern $P(x+y)$ can be matched against $(a+b)+(c+d)$ in several different ways, e.g. $x = a+b$, $y = c+d$ or $x = a$, $y = b$. In order to make the matches determinate, information is taken from corresponding variable bindings. For example there is no doubt about the matching of $\forall x.\ Px$ to $\forall n.\ n < n+1$, whereas with the bound variables removed one could have various alternatives, e.g. $P = \lambda x.\ n < x+1$ and $x = n$. Our allowable patterns correspond quite closely to higher order patterns, for which Miller (1991) proved even the *unification* (two-way matching) problem to be decidable and deterministic ('unitary'). We generalize higher order patterns in two ways. First, one need not simply have variables in the patterns, but can have arbitrary terms involving only these 'resolvable' variables. Thus one can match:

```
|- (!x. P(SUC x)) = !x. 0 < x ==> P x
```

with a term:

```
!n. (m / SUC n) * SUC n = m
```

We allow variables to be repeated in patterns (in the jargon, 'nonlinear' patterns); this does in theory introduce an element of nondeterminacy but this is resolved by always traversing the term to be matched top-down and picking the first match. For example:

```
|- (!x. P (SUC x) x) = !x. 0 < x ==> P x (PRE x)
```

matched against:

```
!n. (m / SUC n) * (n + 1) = m
```

will yield

```
|- (!n. (m / SUC n) * (n + 1) = m) =
   (!n. 0 < n ==> (m / n) * (PRE n + 1) = m)
```

rather than

```
|- (!n. (m / SUC n) * (n + 1) = m) =
   (!n. 0 < n ==> (m / SUC(PRE n)) * (PRE n + 1) = m)
```

Second, as well as binding instances, *first-order matchable* parts of the term are used to resolve more variables. The implementation reflects this: in a first pass, all first order parts are dealt with (in first order matches, all the term is dealt with). Then the new variable assignments are used to keep the overall match deterministic. For example:

```
|- C x y ==> P x y
```

(where `C` is a constant and so not eligible itself as a higher order pattern) will deterministically match:

```
C a b ==> (a + b = 27)
```

whereas the respective consequents could not be matched deterministically.

Note, by the way, that even beta-conversion can be implemented as a higher order rewrite rule, and hence conveniently thrown into a bunch of rewrites instead of being called separately.

```
#BETA_THM;;
it : thm = |- !f y. (\x. f x) y = f y
```

But note that rewrites with the following theorem go into an infinite loop at any beta-redex because of higher order matching!

```
#ETA_AX;;
it : thm = |- !t. (\x. t x) = t
```

9.5 Other rules

As well as these handy general-purpose rules, there are some special ones for particular theories, described later. For example, ARITH_RULE is useful for disposing of trivial facts of linear arithmetic over the natural numbers:

```
#ARITH_RULE `x < y ==> 4 * x + 3 < 4 * y`;;
it : thm = |- x < y ==> 4 * x + 3 < 4 * y
```

Another easy rule, TAUT, proves propositional tautologies automatically, e.g.

```
#TAUT `p /\ q ==> p`;;
it : thm = |- p /\ q ==> p
#TAUT `(p ==> q) \/ (q ==> p)`;;
it : thm = |- (p ==> q) \/ (q ==> p)
```

Chapter 10

Tactics

Rules give a way of performing proofs in a step-by-step, forward manner. However it's often more convenient to prove theorems in a backwards fashion, starting with the goal and reducing it to various subgoals until these can be solved. The tactic mechanism of HOL Light allows one to tackle proofs in a mixture of forward and backward steps. The user starts with a goal, which is roughly speaking, the theorem (sequent) that is desired: a list of assumptions and a conclusion.

A *tactic* takes a goal and reduces it to a list of subgoals. But it also keeps track of how to construct a proof of the main goal if the user succeeds in proving the subgoal; this is simply an ML function. So the user can keep applying tactics, and the forward proof is reconstructed by HOL. It's rather as if the machine automatically reverses the user's proof and converts it to the standard primitive inferences. The user can perform the proof via a mixture of forward and backward steps, as desired.

10.1 The goalstack

Proofs can be discovered interactively using the *goal stack*. This allows tactic steps to be performed, and if necessary retracted and corrected. The user sets up an initial goal using `g`, e.g.

```
#g `p /\ q ==> p`;;
it : goalstack = 1 subgoal (1 total)
`p /\ q ==> p`
```

It is then possible to apply a tactic to the current goal, e.g.

```
#e DISCH_TAC;;
it : goalstack = 1 subgoal (1 total)
`p`

 0 [`p /\ q`]
```

If the user makes a mistake, `b()` backs up to the previous level. The goal can be finished here by rewriting:

```
#e(ASM_REWRITE_TAC[]);;
it : goalstack = No subgoals
```

There are no subgoals; the proof is finished. To make HOL generate the desired theorem, use `top_thm()`:

```
#top_thm();;
it : thm = |- p /\ q ==> p
```

If a tactic splits a goal into more than one subgoal, they are presented one at a time. When one subgoal is solved the next unsolved one is presented. For example:

```
#g `p /\ q /\ r ==> q /\ p /\ r`;;
it : goalstack = 1 subgoal (1 total)
`p /\ q /\ r ==> q /\ p /\ r`

#e DISCH_TAC;;
it : goalstack = 1 subgoal (1 total)
`q /\ p /\ r`

 0 [`p /\ q /\ r`]

#e CONJ_TAC;;
it : goalstack = 2 subgoals (2 total)
`p /\ r`

 0 [`p /\ q /\ r`]
`q`

 0 [`p /\ q /\ r`]

#e(ASM_REWRITE_TAC[]);;
it : goalstack = 1 subgoal (1 total)
`p /\ r`

 0 [`p /\ q /\ r`]

#e CONJ_TAC;;
it : goalstack = 2 subgoals (2 total)
`r`

 0 [`p /\ q /\ r`]
`p`

 0 [`p /\ q /\ r`]

#e(ASM_REWRITE_TAC[]);;
it : goalstack = 1 subgoal (1 total)
`r`

 0 [`p /\ q /\ r`]

#e(ASM_REWRITE_TAC[]);;
it : goalstack = No subgoals
```

Effectively, the user is always at a point in the fringe of the partial proof tree. The position can be rotated by `n` goals by doing `r n`.

10.2 Basic tactics

The most basic tactics correspond to the basic logical derived rules, but working the other way round. We have seen some of them above. For example, where CONJ takes two theorems and gives their conjunction, the tactic CONJ_TAC breaks down a conjunctive goal and returns the two subgoals. Similarly DISJ1_TAC reduces a goal with conclusion $p \vee q$ to one with conclusion p, i.e. allows the user to decide to prove the first disjunct. Again, DISCH_TAC reverses the effect of the rule DISCH, i.e. it reduces a goal $\Gamma \vdash^? p \Rightarrow q$ to $\Gamma \cup \{p\} \vdash^? q$.

Tactics are especially useful for using rules like CHOOSE. If one has a theorem $\vdash \exists x.\, p$, then one can just put p into the assumptions of the goal using CHOOSE_TAC. Thereafter, it is as if one had picked some x such that p holds and can use it to solve the goal; HOL handles the appropriate application of CHOOSE.

The tactics MP_TAC and MATCH_MP are a bit tricker to understand, in that it's not quite so clear how they amount to reversals of MP and MATCH_MP. In fact their behaviour is quite different, going well beyond one performing matching and one not. Given a goal with conclusion q and a theorem that after matching is of the form $p \Rightarrow q$, then MATCH_MP_TAC reduces the goal to p. For example:

```
#g `0 <= SUC n`;;
it : goalstack = 1 subgoal (1 total)
`0 <= SUC n`

#e(MATCH_MP_TAC LT_IMP_LE);;
it : goalstack = 1 subgoal (1 total)
`0 < SUC n`
```

MP_TAC, on the other hand, simply places the theorem as an antecedent of the goal:

```
#g `0 <= SUC n`;;
it : goalstack = 1 subgoal (1 total)
`0 <= SUC n`

#e(MP_TAC LT_IMP_LE);;
it : goalstack = 1 subgoal (1 total)
`(!m n. m < n ==> m <= n) ==> 0 <= SUC n`
```

However this effect can be quite useful, since it's often more convenient to do things like rewrite on the conclusion of a goal, rather than the assumptions.

None of the tactics we have considered so far solves goals completely. The most primitive tactic that does is ACCEPT_TAC, which is used with a theorem with the same conclusion as the goal. A slightly more general version, MATCH_ACCEPT_TAC, will do some matching, e.g.

```
#g `x + 1 = 1 + x`;;
it : goalstack = 1 subgoal (1 total)
`x + 1 = 1 + x`

#e(MATCH_ACCEPT_TAC ADD_SYM);;
it : goalstack = No subgoals
```

Another group of tactics can be created from conversions, using CONV_TAC. This creates a tactic that applies the given conversion to the goal, e.g.

```
#g `(\x. x + 1) 2 = 3`;;
it : goalstack = 1 subgoal (1 total)
`(\x. x + 1) 2 = 3`

#e(CONV_TAC(ONCE_DEPTH_CONV BETA_CONV));;
it : goalstack = 1 subgoal (1 total)
`2 + 1 = 3`
```

If the conversion transforms the goal to `T`, the tactic mechanism accepts that as solving the goal, rather than presenting `T` as the subgoal, e.g.

```
#g `2 + 1 = 3`;;
it : goalstack = 1 subgoal (1 total)
`2 + 1 = 3`

#e(CONV_TAC NUM_REDUCE_CONV);;
it : goalstack = No subgoals
```

The rewriting conversions are also all used as tactics, e.g. `REWRITE_TAC`. The same names prefixed with `ASM_` also use the assumptions of the current goal as rewrites.

10.3 Tacticals

Just as basic conversions are built up into composite ones by conversionals, so tactics are built up via *tacticals*. For example the infix `THEN` executes two tactics in sequence. Once a proof has been found using the subgoal mechanism, it's common to plug all the steps into one tactic using `THEN`, e.g.

```
#g `!m n p. m * (n + p) = (m * n) + (m * p)`;;
it : goalstack = 1 subgoal (1 total)
`!m n p. m * (n + p) = m * n + m * p`

#e(GEN_TAC THEN
   INDUCT_TAC THEN
   ASM_REWRITE_TAC[ADD; MULT_CLAUSES; ADD_ASSOC]);;
it : goalstack = No subgoals
```

If the first tactic sequenced by `THEN` generates more than one subgoal, then the second tactic is applied to all of them. If different tactics are used for each subgoal, they can be put into a list and sequenced using `THENL`, e.g.

```
#g `p ==> p /\ (1 = 1)`;;
it : goalstack = 1 subgoal (1 total)
`p ==> p /\ (1 = 1)`

#e(DISCH_TAC THEN
   CONJ_TAC THENL
    [ASM_REWRITE_TAC[];
     ACCEPT_TAC (REFL `1`)]);;
it : goalstack = No subgoals
```

Tactics can be executed repeatedly by `REPEAT`, and there are various other useful tacticals.

If one uses THEN to compress a proof into a single large tactic, then one might as well dispense with the goal stack completely. One can simple get the theorem by applying prove to the goal and the tactic, e.g.

```
let LTE_ADD2 = prove
 (`!m n p q. m < p /\ n <= q ==> m + n < p + q`,
  ONCE_REWRITE_TAC[ADD_SYM; CONJ_SYM] THEN
  MATCH_ACCEPT_TAC LET_ADD2);;
```

10.4 Dealing with assumptions

Various tactics like DISCH_TAC push parts of the goal onto the assumption list. You can put any theorem there yourself using ASSUME_TAC. The problem then arises of identifying a particular assumption when it is needed. Often it is not necessary, but when required there are several alternatives. One can design a tactic that will succeed only on the desired assumption, and use FIRST_ASSUM. For example the tactic SUBST1_TAC expects and equational theorem as an argument and substitutes in the goal, so if there is only one equational assumption, FIRST_ASSUM SUBST1_TAC will use it. Alternatively, one can explicitly recreate the assumption as a theorem using ASSUME. Finally, it is possible to label things when putting them on the assumptions using LABEL_TAC instead of ASSUME_TAC. The appropriate assumption can then be used with USE_ASSUM.

10.5 Model elimination

Although proofs often need theory-specific reasoning tools, e.g. linear arithmetic, quite a lot of small parts of proofs can be finished off by a prover for pure logic. HOL Light provides a tactic MESON_TAC that can dispose of a lot of simple first order reasoning. It also has a limited ability to do higher order and equality reasoning. This prover is based on the Prolog Technology Theorem Prover (Stickel 1988), an implementation of model elimination (Loveland 1968). Such systems work by reducing to clausal form and then further to a set of pseudo-Horn clauses that can be used for Prolog-style backward search. The default search mode is one of our own invention — see (Harrison 1996) for more details and a comparison with other techniques. Here are a few examples of the HOL tactic in action:

```
#let LOS = prove
 (`(!x y z. P x y /\ P y z ==> P x z) /\
   (!x y z. Q x y /\ Q y z ==> Q x z) /\
   (!x y. P x y ==> P y x) /\
   (!(x:A) y. P x y \/ Q x y)
   ==> (!x y. P x y) \/ (!x y. Q x y)`,
MESON_TAC[]);;
LOS : thm =
 |- (!x y z. P x y /\ P y z ==> P x z) /\
    (!x y z. Q x y /\ Q y z ==> Q x z) /\
    (!x y. P x y ==> P y x) /\
    (!x y. P x y \/ Q x y)
    ==> (!x y. P x y) \/ (!x y. Q x y)
```

and[1]

[1] See message from Wishnu Prasetya to the info-hol mailing list on 18 October 1993, available on the Web as http://lal.cs.byu.edu/lal/holdoc/info-hol/15xx/1515.html.

```
#let WISHNU = prove
  (`(?!x. x=f(g x)) = (?!y. y=g(f y))`,
   MESON_TAC[]);;
WISHNU : thm = |- (?!x. x = f (g x)) = (?!y. y = g (f y))
```

The tactic accepts a list of theorems to use in the proof. `ASM_MESON_TAC` also uses the assumptions of the goal.

Chapter 11

Principles of definition

HOL's basic principles of definition are often quite inconvenient to use. The function `new_definition` is extended quite soon to permit definitions of functions with the arguments on the left, including pairs and tuples of arguments:

```
#let func = new_definition
  `func f x = f(x + 1) - 1`;;
func : thm = |- !f x. func f x = f (x + 1) - 1
#let add3 = new_definition
  `add3(x,y,z) = x + y + z`;;
add3 : thm = |- !x y z. add3 (x,y,z) = x + y + z
```

It's often convenient to make definitions recursively. HOL has some limited support for so-called *primitive recursive* definitions, which we examine below. General recursive functions can be defined using some of the theorems in the theory of well-foundedness described below, but HOL Light doesn't provide any handy functions for doing it elegantly. So one can't write down recursive functions with the abandon that one can in ML. This is inevitable to some extent, since all HOL functions are total and in general recursive definition schemes do not give well-defined or unique total functions. For example $f(n) = f(n) + 1$ has no solution, and neither (at least for functions $\mathbb{N} \to \mathbb{N}$) does $f(n) = f(n+1) + 1$, whereas $f(n) = f(n-1) + 1$ has many possible solutions.

11.1 Inductive definitions

What HOL does support in a more convenient way is the definition of inductive predicates (or sets). Inductive definitions are very common in mathematics, especially in the definition of formal languages used in mathematical logic and programming language semantics. Camilleri and Melham (1992) give some illustrative examples. Examples crop up in other parts of mathematics too, e.g. the definition of the Borel hierarchy of subsets of $\mathbb{R}$. A detailed discussion, from an advanced point of view, is given by Aczel (1991).

Inductive definitions define a set S by means of a set of *rules* of the form 'if ... then $t \in S$', where the hypothesis of the rule may make assertions about membership in S. These rules are customarily written with a horizontal line separating the hypotheses (if any) from the conclusion. For example, the set of even numbers E might be defined as a subset of the reals by:

$$\frac{}{0 \in E}$$

$$\frac{n \in E}{(n+2) \in E}$$

Read literally, such a definition merely places some constraints on the set E, asserting its 'closure' under the rules, and does not, in general, determine it uniquely. For example, the set of even numbers satisfies the above, but so does the set of natural numbers, the set of integers, the set of rational numbers and even the the whole set of real numbers! But implicit in writing a definition like this is that E is the *least* set which is closed under the rules. It is when understood in this sense that the above defines the even numbers.

This convention, however, needs to be justified by showing that there *is* a least set closed under the rules. A good try is to consider the set of *all* sets which are closed under the rules, and take their intersection. If only we knew that this intersection was closed under the rules, then it would certainly be the least such set. But in general we don't know that, as the following example illustrates:

$$\frac{n \notin E}{n \in E}$$

There are no sets at all closed under this rule! However it turns out that a simple syntactic restriction on the rules is enough to guarantee that the intersection is closed under the rules. Crudely speaking, the hypotheses must make only 'positive' assertions about membership in S. To state this precisely, observe that we can collect together all the rules in a single assertion of the form:

$$\forall x.\, P[S,x] \Rightarrow x \in S$$

The following example for the even numbers should be a suitable paradigm to indicate how:

$$\forall n.\, (n = 0 \vee \exists m.\, n = m + 2 \wedge m \in E) \Rightarrow n \in E$$

If we make the abbreviation $f(S) = \{x \mid P[S,x]\}$ the assertion can be written $f(S) \subseteq S$. Our earlier plan was to take the intersection of all subsets S which have this property, and hope that the intersection too is closed under the rules. A sufficient condition for this is given in the following fixpoint theorem due to Knaster (1927) and Tarski (1955) (which holds for an arbitrary complete lattice): if $f : \wp(X) \rightarrow \wp(X)$ is monotone, i.e. for any $S \subseteq X$ and $T \subseteq X$

$$S \subseteq T \Rightarrow f(S) \subseteq f(T)$$

then if we define

$$F = \bigcap \{S \subseteq X \mid f(S) \subseteq S\}$$

not only is $f(F) \subseteq F$ but F is actually a fixpoint of f, i.e. $f(F) = F$. HOL Light can take an inductive definition and generally manage to prove monotonicity for itself, providing the user with three useful theorems. The first says that the defined set is closed under the rules, the second that it is the least set closed under the rules, and the third gives a case analysis theorem saying that everything in the set is generated by applying the rules to something else in the set. For example, we can define finiteness of sets (or, viewed as a set, the set of all finite sets) as follows:

```
#let finite_RULES,finite_INDUCT,finite_CASES =
  new_inductive_definition
    `finite {} /\
     !x s. finite s ==> finite (x INSERT s)`;;
Warning: inventing type variables
finite_RULES : thm =
 |- finite EMPTY /\ (!x s. finite s ==> finite (x INSERT s))
finite_INDUCT : thm =
 |- !finite'. finite' EMPTY /\ (!x s. finite' s ==> finite' (x INSERT s))
              ==> (!a. finite a ==> finite' a)
finite_CASES : thm =
 |- !a. finite a = (a = EMPTY) \/ (?x s. (a = x INSERT s) /\ finite s)
```

HOL Light allows the user to define mutually inductive relations. For example here are predicates for evenness and oddity:

```
#let even_odd_RULES,even_odd_INDUCT,even_odd_CASES =
   new_inductive_definition
    `even 0 /\
     (!n. even(n) ==> odd(n + 1)) /\
     (!n. odd(n) ==> even(n + 1))`;;
even_odd_RULES : thm =
 |- even 0 /\ (!n. even n ==> odd (n + 1)) /\ (!n. odd n ==> even (n + 1))
even_odd_INDUCT : thm =
 |- !odd' even'.
        even' 0 /\
        (!n. even' n ==> odd' (n + 1)) /\
        (!n. odd' n ==> even' (n + 1))
        ==> (!a0. odd a0 ==> odd' a0) /\ (!a1. even a1 ==> even' a1)
even_odd_CASES : thm =
 |- (!a0. odd a0 = (?n. (a0 = n + 1) /\ even n)) /\
    (!a1. even a1 = (a1 = 0) \/ (?n. (a1 = n + 1) /\ odd n))
```

The induction theorem can be applied conveniently during backward proof using the tactical `RULE_INDUCT_THEN`, or in simple cases just with `MATCH_MP_TAC`.

11.2 Free recursive types

HOL Light's primitive type definition facility is rather awkward to work with. One of the most useful, and complicated, derived rules in HOL Light allows one to define recursive types much as in CAML, even using a similar syntax. There are some restrictions; for example a function space involving the type being defined cannot be used. However types can be defined mutually recursively and can involve instances of previously defined type constructors. The primitive function is `define_type`, and it always returns two theorems, the first a kind of induction theorem for the new type, the second a justification of definition by primitive recursion. For example we can define binary trees:

```
#let btree_INDUCT,btree_RECURSION = define_type
  "btree = Leaf A
         | Branch btree btree";;
btree_INDUCT : thm =
 |- !P. (!a. P (Leaf a)) /\ (!a0 a1. P a0 /\ P a1 ==> P (Branch a0 a1))
        ==> (!x. P x)
btree_RECURSION : thm =
 |- !f0 f1.
        ?fn. (!a. fn (Leaf a) = f0 a) /\
             (!a0 a1. fn (Branch a0 a1) = f1 a0 a1 (fn a0) (fn a1))
```

This defines a new type constructor `(A)btree`, since the definition contained a free type variable `A`. The recursion theorem can be used later to define functions by 'primitive recursion', i.e. defining a function on a type constructor in terms of the function on its arguments. For example here are functions to reflect a tree, i.e. swap left and right subtrees, and add up all the integers in an `(int)btree`:

```
#let reflect = new_recursive_definition btree_RECURSION
   `(reflect(Leaf x) = Leaf x) /\
    (reflect(Branch t1 t2) = Branch (reflect t2) (reflect t1))`;;
Warning: inventing type variables
reflect : thm =
 |- (reflect (Leaf x) = Leaf x) /\
    (reflect (Branch t1 t2) = Branch (reflect t2) (reflect t1))
#let addup = new_recursive_definition btree_RECURSION
  `(addup (Leaf n) = n) /\
   (addup (Branch t1 t2) = addup t1 + addup t2)`;;
addup : thm =
 |- (addup (Leaf n) = n) /\ (addup (Branch t1 t2) = addup t1 + addup t2)
```

The induction theorem can be used to prove theorems about objects of the new type. In simple cases one can just use `MATCH_MP_TAC`; for example:

```
#let ADDUP_REFLECT = prove
  (`!t. addup(reflect t) = addup t`,
   MATCH_MP_TAC btree_INDUCT THEN
   SIMP_TAC[addup; reflect; ADD_AC]);;
ADDUP_REFLECT : thm = |- !t. addup (reflect t) = addup t
```

Having defined a type constructor like `btree`, it can itself be used in the definition of new types. For example HOL Light already has a type of lists defined using the definition `list = NIL | CONS A list`, and we can create a type of finitely-branching trees like this:

```
#let xtree_INDUCTION,xtree_RECURSION = define_type
  "xtree = Lf A
         | Br (xtree list)";;
xtree_INDUCTION : thm =
 |- !P0 P1.
        (!a. P0 (Lf a)) /\
        (!a. P1 a ==> P0 (Br a)) /\
        P1 [] /\
        (!a0 a1. P0 a0 /\ P1 a1 ==> P1 (CONS a0 a1))
        ==> (!x0. P0 x0) /\ (!x1. P1 x1)
xtree_RECURSION : thm =
 |- !f0 f1 f2 f3.
        ?fn0 fn1.
            (!a. fn1 (Lf a) = f0 a) /\
            (!a. fn1 (Br a) = f1 a (fn0 a)) /\
            (fn0 [] = f2) /\
            (!a0 a1. fn0 (CONS a0 a1) = f3 a0 a1 (fn1 a0) (fn0 a1))
```

The induction and recursion theorems are as if the list constructor had been defined mutually recursively, but it uses the standard type of lists.

Chapter 12

Mathematical theories

To prove theorems in HOL Light, one needs a reasonable grasp of the theorem proving infrastructure. But equally important is knowing what has already been proved, and what the theorem one is after has been called. The following is not an exhaustive list, but gives some of the main theorems, grouped according to subject area. The following gives only a general overview; the reader should browse the source files to become more familiar with what is available.

12.1 Pairs

There is a type constructor `prod` that constructs Cartesian product types. In the concrete syntax of the type parser it is written as `#`. For example `num # num` is the type of pairs of natural numbers. Larger tuples can be built by iterating the pairing operation; the type constructor and the term function that constructs pairs (the infix comma) are both right associative. Destructors `FST` and `SND` are defined. Some of the main theorems about pairs are:

```
PAIR_EQ = |- !x y a b. (x,y = a,b) = (x = a) /\ (y = b)

PAIR_SURJECTIVE = |- !p. ?x y. p = x,y

FST = |- !x y. FST (x,y) = x

SND = |- !x y. SND (x,y) = y

PAIR = |- !x. FST x,SND x = x

pair_INDUCT = |- (!x y. P (x,y)) ==> (!p. P p)

pair_RECURSION = |- !PAIR'. ?fn. !a0 a1. fn (a0,a1) = PAIR' a0 a1
```

The last two are chosen as if pairs had been defined as a recursive type, though in fact they logically precede other recursive types.

12.2 Natural numbers

The type of natural numbers is carved out, using an inductive definition, from the infinite type `ind`. The Peano axioms are derived from the definition and the axioms of infinity. As with pairs, two theorems mimic those resulting from recursive type definitions, allowing proofs by induction and definitions by primitive recursion:

```
num_INDUCTION = |- !P. P 0 /\ (!n. P n ==> P (SUC n)) ==> (!n. P n)

num_RECURSION = |- !e f. ?fn. (fn 0 = e) /\ (!n. fn (SUC n) = f (fn n) n)
```

The latter is used to define most of the arithmetic operations, including the comparisons:

```
ADD = |- (!n. 0 + n = n) /\ (!m n. SUC m + n = SUC (m + n))

MULT = |- (!n. 0 * n = 0) /\ (!m n. SUC m * n = m * n + n)

EXP = |- (!m. m EXP 0 = 1) /\ (!m n. m EXP SUC n = m * m EXP n)

LE = |- (!m. m <= 0 = m = 0) /\
        (!m n. m <= SUC n = (m = SUC n) \/ m <= n)

LT = |- (!m. m < 0 = F) /\ (!m n. m < SUC n = (m = n) \/ m < n)

EVEN = |- (EVEN 0 = T) /\ (!n. EVEN (SUC n) = ~EVEN n)

ODD = |- (ODD 0 = F) /\ (!n. ODD (SUC n) = ~ODD n)
```

Numerals are prettyprinted forms of an internal binary representation using two constants:

```
BIT0 = |- BIT0 n = n + n

BIT1 = |- BIT1 n = SUC(n + n)
```

The rather artificial definition of the second is because multiplication (which uses numeral 1 in its definition) has not yet been defined. Now these constants are sufficient to express any number in binary. For example, we implement 37 as:

```
NUMERAL (BIT1 (BIT0 (BIT1 (BIT0 (BIT0 (BIT1 _0))))))
```

The reader may wonder why we use the constant `NUMERAL` at all, instead of just using `BIT0`, `BIT1` and `0`. The reason is that in that case one number becomes a subterm of another (e.g. 1 is a subterm of 2), which can lead to some surprising accidental rewrites. Besides, the `NUMERAL` constant is a useful tag for the prettyprinter.

The parser and printer transformations established, the theory of natural numbers can now be developed as usual. Most of the arithmetic operations are defined by primitive recursion, indicating a simple evaluation strategy for unary notation. For example one can evaluate $3 + 5$ as follows:

```
3 + 5
SUC 2 + 5
SUC (2 + 5)
SUC (SUC 1 + 5)
SUC (SUC (1 + 5))
SUC (SUC (SUC 0 + 5)))
SUC (SUC (SUC (0 + 5)))
SUC (SUC (SUC 5))
SUC (SUC 6)
SUC 7
8
```

But many of them have an almost equally direct strategy in terms of our binary notation.[1] For example the following theorems, easily proved, can be used directly as rewrite rules to perform arithmetic evaluation.

```
|- (!n. SUC (NUMERAL n) = NUMERAL (SUC n)) /\
   (SUC _0 = BIT1 _0) /\
   (!n. SUC (BIT0 n) = BIT1 n) /\
   (!n. SUC (BIT1 n) = BIT0 (SUC n))
```

or

```
|- (!m n. (NUMERAL m = NUMERAL n) = (m = n)) /\
   ((_0 = _0) = T) /\
   (!n. (BIT0 n = _0) = (n = _0)) /\
   (!n. (BIT1 n = _0) = F) /\
   (!n. (_0 = BIT0 n) = (_0 = n)) /\
   (!n. (_0 = BIT1 n) = F) /\
   (!m n. (BIT0 m = BIT0 n) = (m = n)) /\
   (!m n. (BIT0 m = BIT1 n) = F) /\
   (!m n. (BIT1 m = BIT0 n) = F) /\
   (!m n. (BIT1 m = BIT1 n) = (m = n))
```

Most arithmetic operations can be implemented as a set of rewrite rules like the above, and applied using the standard rewriting mechanism. A suite of such rewrites is collected together into a single rewrite rule `ARITH` that will evaluate most ground expressions using just the standard rewriting mechanism. For example:

```
#let conv = PURE_REWRITE_CONV[ARITH];;
conv : conv = <fun>
#conv `12345 * 12345`;;
it : thm = |- 12345 * 12345 = 152399025
```

However, a few operations are hard to evaluate efficiently with the standard rewriting mechanism; even `ARITH_SUB` is a bit inefficient, since the same condition is tested repeatedly. Therefore we also provide a full suite of conversions, and collect them together as `NUM_RED_CONV` and `NUM_REDUCE_CONV`.

12.3 Lists

A HOL recursive type of lists is defined, and various standard list combinators defined by recursion. These often have the same names as their CAML counterparts, but in upper case.

```
HD = |- HD (CONS h t) = h

TL = |- TL (CONS h t) = t

APPEND =
 |- (!l. APPEND [] l = l) /\
    (!h t l. APPEND (CONS h t) l = CONS h (APPEND t l))
```

[1] Another nice example, though we don't actually implement it, is the GCD function. Knuth (1969) gives a simple algorithm based on $gcd(2m, 2n) = 2gcd(m, n)$, $gcd(2m + 1, 2n) = gcd(2m + 1, n)$ and $gcd(2m+1, 2n+1) = gcd(m-n, 2n+1)$. This outperforms Euclid's method on machines where bitwise operations are relatively efficient; our in-logic implementation would surely exhibit the same characteristics even if our 'bits' are rather large!

```
REVERSE =
 |- (REVERSE [] = []) /\ (REVERSE (CONS x l) = APPEND (REVERSE l) [x])

LENGTH =
 |- (LENGTH [] = 0) /\ (!h t. LENGTH (CONS h t) = SUC (LENGTH t))

MAP = |- (!f. MAP f [] = []) /\
         (!f h t. MAP f (CONS h t) = CONS (f h) (MAP f t))

LAST = |- LAST (CONS h t) = (if t = [] then h else LAST t)

REPLICATE = |- (REPLICATE 0 x = []) /\
               (REPLICATE (SUC n) x = CONS x (REPLICATE n x))

NULL = |- (NULL [] = T) /\ (NULL (CONS h t) = F)

FORALL = |- (FORALL P [] = T) /\
            (FORALL P (CONS h t) = P h /\ FORALL P t)

EX = |- (EX P [] = F) /\ (EX P (CONS h t) = P h \/ EX P t)

ITLIST =
 |- (ITLIST f [] b = b) /\ (ITLIST f (CONS h t) b = f h (ITLIST f t b))

MEM = |- (MEM x [] = F) /\ (MEM x (CONS h t) = (x = h) \/ MEM x t)
```

A somewhat ad hoc collection of facts about these functions is collected, for example:

```
APPEND_ASSOC = |- !l m n. APPEND l (APPEND m n) = APPEND (APPEND l m) n

LENGTH_APPEND = |- !l m. LENGTH (APPEND l m) = LENGTH l + LENGTH m

LENGTH_MAP = |- !l f. LENGTH (MAP f l) = LENGTH l

REVERSE_REVERSE = |- !l. REVERSE (REVERSE l) = l

MAP_o = |- !f g l. MAP (g o f) l = MAP g (MAP f l)

NOT_EX = |- !P l. ~EX P l = FORALL (\x. ~P x) l
```

12.4 Well-founded relations

Wellfoundedness of a binary relation can be expressed in many equivalent ways. HOL Light includes a definition of wellfoundedness and a proof that it equivalent to several other important properties, like the admissibility of complete induction and wellfounded recursion. For example, the last theorem below, which also has a converse, says that one can define recursive functions provided the value of $f(x)$ is defined in terms of $f(y)$ for y below x in the wellfounded ordering.

```
WF =
 |- WF (<<) = (!P. (?x. P x) ==> (?x. P x /\ (!y. y << x ==> ~P y)))
```

```
WF_IND =
 |- WF (<<) = (!P. (!x. (!y. y << x ==> P y) ==> P x) ==> (!x. P x))

WF_DCHAIN = |- WF (<<) = ~(?s. !n. s (SUC n) << s n)

WF_REC =
 |- WF (<<)
    ==> (!H. (!f g x. (!z. z << x ==> (f z = g z)) ==> (H f x = H g x))
             ==> (?f. !x. f x = H f x))
```

12.5 Real numbers

HOL Light constructs the real numbers and then proves various properties of them. Algebraic trivialities include:

```
REAL_OF_NUM_SUB : thm = |- !m n. m <= n ==> (&n - &m = &(n - m))

REAL_ADD_RID : thm = |- !x. x + &0 = x

REAL_LT_IMP_LE : thm = |- !x y. x < y ==> x <= y

REAL_LT_LADD_IMP : thm = |- !x y z. y < z ==> x + y < x + z

REAL_LT_LNEG : thm = |- !x y. -- x < y = &0 < x + y

REAL_ABS_TRIANGLE : thm = |- !x y. abs (x + y) <= abs x + abs y

REAL_ABS_MUL : thm = |- !x y. abs (x * y) = abs x * abs y

REAL_INV_MUL : thm = |- !x y. inv (x * y) = inv x * inv y
```

Note that the inverse is defined with $0^{-1} = 0$. Most theorems not involving multiplication can be proved automatically using REAL_ARITH or the tactic form REAL_ARITH_TAC:

```
#REAL_ARITH `abs(x) < y ==> x < y`;;
it : thm = |- abs x < y ==> x < y
```

The key higher-order property of the reals asserts that any nonempty bounded set of reals has a least upper bound:

```
#REAL_COMPLETE;;
it : thm =
 |- !P. (?x. P x) /\ (?M. !x. P x ==> x <= M)
        ==> (?M. (!x. P x ==> x <= M) /\
                 (!M'. (!x. P x ==> x <= M') ==> M <= M'))
```

There is not much real analysis in the basic system, but there is a reasonable development included with the examples.

12.6 Integers

A theory of integers is also available, with theorems named by analogy with the reals, e.g. INT_LT_IMP_LE rather than REAL_LT_IMP_LE. Similarly, there is a decision procedure for linear integer arithmetic called INT_ARITH.

12.7 Sets

Sets in HOL Light are just predicates, but the usual set operations are defined:

```
EMPTY = |- EMPTY = (\x. F)

UNIV = |- UNIV = (\x. T)

UNION = |- !s t. s UNION t = {x | x IN s \/ x IN t}

UNIONS = |- !s. UNIONS s = {x | ?u. u IN s /\ x IN u}

INTER = |- !s t. s INTER t = {x | x IN s /\ x IN t}

INTERS = |- !s. INTERS s = {x | !u. u IN s ==> x IN u}

DIFF = |- !s t. s DIFF t = {x | x IN s /\ ~(x IN t)}

INSERT = |- x INSERT s = {y | y IN s \/ (y = x)}

DELETE = |- !s x. s DELETE x = {y | y IN s /\ ~(y = x)}

SUBSET = |- !s t. s SUBSET t = (!x. x IN s ==> x IN t)

PSUBSET = |- !s t. s PSUBSET t = s SUBSET t /\ ~(s = t)

DISJOINT = |- !s t. DISJOINT s t = s INTER t = EMPTY
```

The parser and printer support set enumerations and set abstractions. Trivial facts of set theory, which are just liftings of first order facts, can be proved automatically in a tactic framework using `SET_TAC`, e.g.

```
#prove
  (`x INSERT (s UNION t) = (x INSERT s) UNION (x INSERT t)`,SET_TAC[]);;
it : thm = |- x INSERT (s UNION t) = x INSERT s UNION x INSERT t
```

There are quite a lot of such theorems pre-proved. Some more interesting pre-proved theorems concern the finiteness and cardinality of sets, and in general the definition of function over finite sets by recursion:

```
CARD_CLAUSES =
 |- (CARD EMPTY = 0) /\
    (!x s.
         FINITE s
         ==> (CARD (x INSERT s) =
              if x IN s then CARD s else SUC (CARD s)))

HAS_SIZE = |- !s n. s HAS_SIZE n = FINITE s /\ (CARD s = n)

CARD_SUBSET_LE =
 |- !a b. FINITE b /\ a SUBSET b /\ CARD b <= CARD a ==> (a = b)

FINITE_RECURSION =
 |- !f b.
        (!x y s. ~(x = y) ==> (f x (f y s) = f y (f x s)))
```

```
==> (ITSET f EMPTY b = b) /\
    (!x s.
         FINITE s
         ==> (ITSET f (x INSERT s) b =
              if x IN s then ITSET f s b else f x (ITSET f s b)))
```

Chapter 13

Examples

A few examples are included in `Examples` directory. These just give some indication of how the system can be used. They aren't held up as particularly good examples exploiting HOL Light's facilities; indeed many of them are crudely ported from older versions of HOL. A few of them might be useful to some readers, but they are generally not polished or documented.

- `analysis.ml` is a development of elementary real analysis, e.g. sequences, series, limits, continuity, differentiation and integration.
- `lagrange.ml` shows how to prove some numerical identities using ordered rewriting and/or decision procedures.
- `mizar.ml` is a system for writing HOL proofs in a more readable declarative style based on Trybulec's Mizar system (Rudnicki 1992).
- `prog.ml` is a simple embedding of the semantics of a toy imperative programming language, derivation of weakest preconditions and Floyd-Hoare rules, and a tactic that performs verification condition generation on an annotated program.
- `rectypes.ml` defines a wide variety of (mutual, nested) recursive types.
- `reduct.ml` defines some basic notions for reductions, e.g. confluence, normalization, and proves a few theorems like Newman's Lemma. It requires `rstc.ml` to have been loaded first.
- `rstc.ml` defines various combinations of reflexive, symmetric and transitive closures of binary relations, and proves a comprehensive set of theorems about them.
- `transc.ml` defines and proves properties of the elementary transcendental functions like `exp`, `sin` and `ln`. It requires `analysis.ml` to have been loaded first.
- `wo.ml` proves some important version of the Axiom of Choice, e.g. the wellordering principle and Zorn's Lemma.

Appendix A

Compatibility with other HOLs

Here is a brief list of some of the major incompatibilities with other versions of HOL:

- CAML, the underlying ML is different from previous HOL versions, somewhere between 'Classic ML' and Standard ML.
- There is no theory mechanism; every theorem is just bound to an ML name. It is possible to save and load theorems via CAML primitives, but this is not recommended since it subverts the usual mechanisms for constructing elements of the `thm` type.
- Parsing status is orthogonal to whether an identifier is a constant or a variable. Parsing status is not indicated at the time constants are defined. To suppress special parse status, HOL Light requires the identifier to be put in parentheses like `(+)`, whereas other HOL versions use `$+`.
- Higher order matching is applied pervasively throughout the system, and in some cases this can lead to a different result from a first order match even when both succeed.
- All permutative rewrite rules are automatically ordered by the rewriting functions.
- Operator overloading is permitted in the surface syntax. There are however still some limitations on overloading of polymorphic operators. The interface map feature in previous HOLs has been abolished and operator overloading is used instead.
- Decision procedures for linear arithmetic are available for integers and reals as well as naturals.
- A comprehensive theory of wellfounded relations is provided, but no tools for automating general recursive definitions.
- The resolution tactics have been removed, or more accurately replaced by trivial ones that do not attempt multiple chaining.
- Goals have theorems as assumptions, rather than terms to be assumed. The tactic mechanism allows the use of instantiable metavariables, and assumptions may be labelled with names. The internal type of tactics has changed to reflect these changes.

- The names of many theorems, especially about natural numbers, are different. Some of the operations on natural numbers are defined differently.

- Various facilities are in the core system rather than loadable libraries, e.g. tautology checking, higher order matching, first order reasoning, quotient types, integers, reals and nested recursive types.

- The axiomatization of the logic is simpler and all 'derived rules' are genuinely derived. There is no separate boolean cases axiom, since it follows from the axiom of choice.

- The preferred concrete syntax for conditional expressions is 'if ... then ... else ...', although the old HOL syntax is still accepted.

- The internal encodings of paired abstractions and let-terms are different. The former is an instance of a more general method of allowing abstractions over arbitrary expressions.

- The term syntax uses a name-carrying representation like HOL88, rather than a de Bruijn representation as in hol90. It was felt that this would be more efficient on average, even if it makes a couple of primitive term operations like substitution tricky to get right.

Despite the above, readers familiar with older HOLs should find the system reasonably familiar. Many of the differences do not greatly affect day-to-day use of the system.

Bibliography

Aczel, P. (1991) An introduction to inductive definitions. In Barwise, J. and Keisler, H. (eds.), *Handbook of mathematical logic*, Volume 90 of *Studies in Logic and the Foundations of Mathematics*, pp. 739–782. North-Holland.

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.

Beeson, M. J. (1984) *Foundations of constructive mathematics: metamathematical studies*, Volume 3 of *Ergebnisse der Mathematik und ihrer Grenzgebiete*. Springer-Verlag.

Camilleri, J. and Melham, T. (1992) Reasoning with inductively defined relations in the HOL theorem prover. Technical Report 265, University of Cambridge Computer Laboratory, New Museums Site, Pembroke Street, Cambridge, CB2 3QG, UK.

Carnap, R. (1937) *The Logical Syntax of Language*. International library of psychology, philosophy and scientific method. Routledge & Kegan Paul. Translated from 'Logische Syntax der Sprache' by Amethe Smeaton (Countess von Zeppelin), with some new sections not in the German original.

Church, A. (1940) A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, **5**, 56–68.

Curry, H. B. (1930) Grundlagen der Kombinatorischen Logik. *American Journal of Mathematics*, **52**, 509–536, 789–834.

Davis, M. D., Sigal, R., and Weyuker, E. J. (1994) *Computability, complexity, and languages: fundamentals of theoretical computer science* (2nd ed.). 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.

Gordon, M. J. C. (1982) Representing a logic in the LCF metalanguage. In Néel, D. (ed.), *Tools and notions for program construction: an advanced course*, pp. 163–185. Cambridge University Press.

Gordon, M. J. C. (2000) From LCF to HOL: A short history. In Plotkin, G., Stirling, C., and Tofte, M. (eds.), *Proof, language, and interaction: essays in honour of Robin Milner*. MIT Press.

Gordon, M. J. C. and Melham, T. F. (1993) *Introduction to HOL: a theorem proving environment for higher order logic*. Cambridge University Press.

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.

Harrison, J. (1995) Inductive definitions: automation and application. In Windley, P. J., Schubert, T., and Alves-Foss, J. (eds.), *Higher Order Logic Theorem Proving and Its Applications: Proceedings of the 8th International Workshop*, Volume 971 of *Lecture Notes in Computer Science*, Aspen Grove, Utah, pp. 200–213. Springer-Verlag.

Harrison, J. (1996) Optimizing proof search in model elimination. In McRobbie, M. A. and Slaney, J. K. (eds.), *13th International Conference on Automated Deduction*, Volume 1104 of *Lecture Notes in Computer Science*, New Brunswick, NJ, pp. 313–327. Springer-Verlag.

Heijenoort, J. v. (ed.) (1967) *From Frege to Gödel: A Source Book in Mathematical Logic 1879–1931*. Harvard University Press.

Henkin, L. (1963) A theory of propositional types. *Fundamenta Mathematicae*, **52**, 323–344.

Jones, R. and Lins, R. (1996) *Garbage collection: algorithms for automatic dynamic memory management*. Wiley.

Knaster, B. (1927) Un théorème sur les fonctions d'ensembles. *Annales de la Société Polonaise de Mathématique*, **6**, 133–134. Volume published in 1928.

Knuth, D. E. (1969) *The Art of Computer Programming; Volume 2: Seminumerical Algorithms*. Addison-Wesley Series in Computer Science and Information processing. Addison-Wesley.

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`.

Lambek, J. and Scott, P. J. (1986) *Introduction to higher order categorical logic*, Volume 7 of *Cambridge studies in advanced mathematics*. Cambridge University Press.

Loveland, D. W. (1968) Mechanical theorem-proving by model elimination. *Journal of the ACM*, **15**, 236–251.

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.

Martin, U. and Nipkow, T. (1990) Ordered rewriting and confluence. In Stickel, M. E. (ed.), *10th International Conference on Automated Deduction*, Volume 449 of *Lecture Notes in Computer Science*, Kaiserslautern, Federal Republic of Germany, pp. 366–380. Springer-Verlag.

Mauny, M. (1995) Functional programming using CAML Light. Available on the Web from `http://pauillac.inria.fr/caml/tutorial/index.html`.

Miller, D. (1991) A logic programming language with lambda-abstraction, function variables and simple unification. *Journal of Logic and Computation*, **1**, 497–536.

Milner, R. (1978) A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences*, **17**, 348–375.

Paulson, L. C. (1983) A higher-order implementation of rewriting. *Science of Computer Programming*, **3**, 119–149.

Paulson, L. C. (1987) *Logic and computation: interactive proof with Cambridge LCF*, Volume 2 of *Cambridge Tracts in Theoretical Computer Science*. Cambridge University Press.

Paulson, L. C. (1991) *ML for the Working Programmer*. Cambridge University Press.

Prawitz, D. (1965) *Natural deduction; a proof-theoretical study*, Volume 3 of *Stockholm Studies in Philosophy*. Almqvist and Wiksells.

Raphael, B. (1966) The structure of programming languages. *Communications of the ACM*, **9**, 155–156.

Rudnicki, P. (1992) An overview of the MIZAR project. Available on the Web as `http://web.cs.ualberta.ca/~piotr/Mizar/MizarOverview.ps`.

Schönfinkel, M. (1924) Über die Bausteine der mathematischen Logik. *Mathematische Annalen*, **92**, 305–316. English translation, 'On the building blocks of mathematical logic' in Heijenoort (1967), pp. 357–366.

Stickel, M. E. (1988) A Prolog Technology Theorem Prover: Implementation by an extended Prolog compiler. *Journal of Automated Reasoning*, **4**, 353–380.

Tarski, A. (1936) Der Wahrheitsbegriff in den formalisierten Sprachen. *Studia Philosophica*, **1**, 261–405. English translation, 'The Concept of Truth in Formalized Languages', in Tarski (1956), pp. 152–278.

Tarski, A. (1955) A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, **5**, 285–309.

Tarski, A. (ed.) (1956) *Logic, Semantics and Metamathematics*. Clarendon Press.

Weis, P. and Leroy, X. (1993) *Le langage Caml*. InterEditions. See also the CAML Web page: `http://pauillac.inria.fr/caml/`.

Whitehead, A. N. and Russell, B. (1910) *Principia Mathematica (3 vols)*. Cambridge University 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.