[For hol98 Taupo-3] April 13, 2000

The HOL System
TUTORIAL

University of Cambridge DSTO SRI International

Preface

This volume contains a tutorial on the HOL system. It is one of three documents making up the documentation for HOL:

(i) *TUTORIAL*: a tutorial introduction to HOL.

(ii) *DESCRIPTION*: a description of higher order logic, the ML programming language, and theorem proving methods in the HOL system;

(iii) *REFERENCE*: the reference documentation of the tools available in HOL.

These three documents will be referred to by the short names (in small slanted capitals) given above.

This document, *TUTORIAL*, is intended to be the first item read by new users of HOL. It provides a self-study introduction to the structure and use of the system. The tutorial is intended to give a 'hands-on' feel for the way HOL is used, but it does not systematically explain all the underlying principles (*DESCRIPTION*, explains these). After working through *TUTORIAL* the reader should be capable of using HOL for simple tasks, and should also be in a position to consult the other two documents.

Getting started

Chapter 1 explains how to get and install HOL. Once this is done, the potential HOL user should become familiar with the following subjects:

1. The programming meta-language ML, and how to interact with it through an editor.

2. The formal logic supported by the HOL system (higher order logic) and its manipulation via ML.

3. Forward proof and derived rules of inference.

4. Goal directed proof, tactics and tacticals.

Chapters 1–3 introduce the first two of these topics. Chapter 4 then develops an extended example (Euclid's proof of the infinitude of primes) to demonstrate how HOL is used to prove theorems. This example is intended to demonstrate HOL's capabilities and to explain some of the issues at a high level. Chapters 5 and 6 then describe forward and goal directed proof in much greater detail.

Chapter 7 consists of a worked example: the specification and verification of a simple sequential parity checker. The intention is to accomplish two things: (i) to present a complete piece of work with HOL; and (ii) to give an idea of what it is like to use the HOL system for a tricky proof.

Chapter 8 briefly discusses some of the examples distributed with hol98 in the `examples` directory.

TUTORIAL has been kept short so that new users of HOL can get going as fast as possible. Sometimes details have been simplified. It is recommended that as soon as a topic in *TUTORIAL* has been digested, the relevant bits of *DESCRIPTION* and *REFERENCE* be studied.

Acknowledgements

First edition

The three volumes *TUTORIAL*, *DESCRIPTION* and *REFERENCE* were produced at the Cambridge Research Center of SRI International with the support of DSTO Australia.

The HOL documentation project was managed by Mike Gordon, who also wrote parts of *DESCRIPTION* and *TUTORIAL* using material based on an early paper describing the HOL system[1] and *The ML Handbook*[2]. Other contributers to *DESCRIPTION* incude Avra Cohn, who contributed material on theorems, rules, conversions and tactics, and also composed the index (which was typeset by Juanito Camilleri); Tom Melham, who wrote the sections describing type definitions, the concrete type package and the 'resolution' tactics; and Andy Pitts, who devised the set-theoretic semantics of the HOL logic and wrote the material describing it.

The original document design used LaTeX macros supplied by Elsa Gunter, Tom Melham and Larry Paulson. The typesetting of all three volumes was managed by Tom Melham. The cover design is by Arnold Smith, who used a photograph of a 'snow watching lantern' taken by Avra Cohn (in whose garden the original object resides). John Van Tassel composed the LaTeX picture of the lantern.

Many people other than those listed above have contributed to the HOL documentation effort, either by providing material, or by sending lists of errors in the first edition. Thanks to everyone who helped, and thanks to DSTO and SRI for their generous support.

Later editions

The second edition of *REFERENCE* was a joint effort by the Cambridge HOL group.

The third edition of all three volumes represents a wide-ranging and still incomplete revision of material written for HOL88 so that it applies to the hol98 system a decade later. The third edition has been prepared by Konrad Slind and Michael Norrish.

[1]M.J.C. Gordon, 'HOL: a Proof Generating System for Higher Order Logic', in: *VLSI Specification, Verification and Synthesis*, edited by G. Birtwistle and P.A. Subrahmanyam, (Kluwer Academic Publishers, 1988), pp. 73–128.

[2]*The ML Handbook*, unpublished report from Inria by Guy Cousineau, Mike Gordon, Gérard Huet, Robin Milner, Larry Paulson and Chris Wadsworth.

Contents

Chapter 1

Getting and Installing HOL

This chapter describes how to get the HOL system and how to install it. It is generally assumed that some sort of Unix system is being used, but the instructions that follow should apply *mutatis mutandis* to other platforms. Unix is not a pre-requisite for using the system. HOL may be run PCs running Windows NT, and we are always interested in ports to other platforms.

1.1 Getting HOL

The HOL system can be downloaded from `http://www.ftp.cl.cam.ac.uk/ftp/hvg/`. This page on the World-wide Web contains a number of sub-directories. The system discussed in this tutorial is hol98, so the `hol98` sub-directory contains the appropriate source files. The naming scheme for hol98 releases is ⟨*name*⟩-⟨*number*⟩; the release described here is Taupo-3.

1.2 The `info-hol` mailing list

Phil Windley has started a mailing list: `info-hol@jaguar.cs.byu.edu` which he set up to serve as a forum for discussing HOL and disseminating news about it. If you wish to be on this list (which is recommended for all users of HOL), or know of other people who should be included, email to: `info-hol-request@jaguar.cs.byu.edu`.

1.3 Installing HOL

It is assumed that the HOL sources have been obtained and the `tar` file unpacked into a directory `hol98`.[1] The contents of this directory are likely to change over time, but it should contain the following:

[1]You may choose another name if you want; it is not important.

Principal Files on the HOL Distribution Directory

File name	*Description*	*File type*
`README`	Description of directory `hol`	Text
`COPYRIGHT`	A copyright notice	Text
`install.txt`	Installation instructions	Text
`tools`	Source code for building the system	Directory
`bin`	Directory for HOL executables	Directory
`sigobj`	Directory for ML object files	Directory
`src`	ML sources of HOL	Directory
`Manual`	Files for HOL system documentation	Directory
`help`	Help files for HOL system	Directory
`examples`	Example source files	Directory

The session in the box below shows a typical distribution directory. The HOL distribution has been placed on a PC running Linux in the directory `/local/scratch/mn200/hol98/`.

All sessions in this documentation will be displayed in boxes with a number in the top right hand corner. This number indicates whether the session is a new one (when the number will be *1*) or the continuation of a session started in an earlier box. Consecutively numbered boxes are assumed to be part of a single continuous session. The Unix prompt for the sessions is `$`, so lines beginning with this prompt were typed by the user. After entering the HOL system (see below), the user is prompted with `-` for an expression or command of the HOL meta-language ML; lines beginning with this are thus ML expressions or declarations. Lines not beginning with `$` or `-` are system output. Occasionally, system output will be replaced with a line containing . . . when it is of minimal interest. The meta-language ML is introduced in Chapter 2.

```
$ pwd                                                                   1
/local/scratch/mn200/hol98
$ ls -F
COPYRIGHT  bin/  examples/  install.txt  src/
README     doc/  help/      sigobj/      tools/
```

Now you will need to rebuild HOL from the sources.[2]

Before beginning you must have a current version of MoscowML. In particular, you must have version 1.44 or later. MoscowML is available on the web from

```
http://www.dina.kvl.dk/~sestoft/mosml.html
```

The first step of the installation process is to edit the file `tools/configure.sml`. At the top of the file are the parameters that need to be set.

[2]It is possible that pre-built systems may soon be available from the web-page mentioned above.

```
$ cd tools/                                                                2
$ head -30 configure.sml
(*---------------------------------------------------------------------------
             HOL98 configuration script

   First, edit the following user-settable parameters. Then execute this
   file by going

      mosml < configure.sml

 ---------------------------------------------------------------------------*)



(*---------------------------------------------------------------------------
          BEGIN user-settable parameters
 ---------------------------------------------------------------------------*)

val mosmldir = _
val holdir   = _
val OS       = _         (* Operating system; choices are:
                                "linux", "solaris", "unix", "winNT" *)

val CC       = "gcc";     (* C compiler (for building quote filter)       *)
val GNUMAKE  = "gnumake"; (* for robdd library                            *)

val DEPDIR   = ".HOLMK";  (* local dir. where Holmake dependencies kept   *)
val LN_S     = "ln -s";   (* only change if you are a HOL developer.      *)

(*---------------------------------------------------------------------------
          END user-settable parameters
 ---------------------------------------------------------------------------*)
```

The first three parameters *must* be set to sensible values. The mosmldir value must be the name of the directory where the MoscowML implementation is on your machine. The holdir value must be the name of the top-level directory listed in the first session above. The OS value should be one of the strings specified in the accompanying comment. All three strings must be enclosed in double quotes.

The next two values (CC and GNUMAKE) are needed for “optional” components of the system. The first gives a string suitable for invoking the system’s C compiler, and the second specifies a make program.

After editing, tools/configure.sml the lines above will look something like:

```
$ more configure.sml                                                          3
  ...
val mosmldir = "/usr/groups/hol/mosml.144";
val holdir   = "/local/scratch/mn200/hol98";
val OS       = "linux"       (* Operating system; choices are:
                                "linux", "solaris", "unix", "winNT" *)

val CC       = "gcc";     (* C compiler (for building quote filter)        *)
val GNUMAKE  = "gnumake"; (* for robdd library                             *)
  ...
$
```

Now, at either this level (in the `tools` directory) or at the level above, the `configure.sml` script must be piped into the MoscowML interpreter (called `mosml`). This process creates a suite of useful tools that will be used to build the full system. In particular, it will compile Holmake and build and put them in the `bin` directory.

```
$ /usr/groups/hol/mosml.144/bin/mosml < configure.sml                         4
Moscow ML version 1.44 (August 1999)
Enter ‘quit();’ to quit.
- > val mosmldir = "/usr/groups/hol/mosml.144" : string
- > val holdir = "/local/scratch/mn200/hol98" : string
- > val OS = "linux" : string
  val CC = "gcc" : string
  ...
Beginning configuration.
- Making bin/Holmake.
- Making bin/build.
- Making hol98-mode.el (for Emacs)
- Setting up the standard prelude.
- Setting up src/0/Globals.sml.
- Generating bin/hol.
- Attempting to compile quote filter ... successful.
Generating bin/hol.unquote.
- Setting up the robdd library Makefile.
- Setting up the muddy library Makefile.
-
Finished configuration!
-
$
```

The next step is to run the build program. This should result in a great deal of output as all of the system code is compiled and the theories built. Eventually, a HOL system[3] is produced in the `bin/` directory.

[3]Two executables are produced: hol and hol.unquote. The latter will be used for most examples in the *TUTORIAL*.

```
$ ../bin/build                                                      5
  ...
  ...
Uploading files to /local/scratch/mn200/Work/hol98/sigobj

Hol built successfully.
$
```

Chapter 2

Introduction to ML

This chapter is a brief introduction to the meta-language ML. The aim is just to give a feel for what it is like to interact with the language. A more detailed introduction can be found in numerous textbooks and web-pages; see for example the list of resources on the MoscowML home-page, or the `comp.lang.ml` FAQ.

2.1 How to interact with ML

ML is an interactive programming language like Lisp. At top level one can evaluate expressions and perform declarations. The former results in the expression's value and type being printed, the latter in a value being bound to a name.

A standard way to interact with ML is to configure the workstation screen so that there are two windows:

(i) An editor window into which ML commands are initially typed and recorded.

(ii) A shell window (or non-Unix equivalent) which is used to evaluate the commands.

A common way to achieve this is to work inside `Emacs` with a text window and a shell window.

After typing a command into the edit (text) window it can be transferred to the shell and evaluated in HOL by 'cut-and-paste'. In `Emacs` this is done by copying the text into a buffer and then 'yanking' it into the shell. The advantage of working via an editor is that if the command has an error, then the text can simply be edited and used again; it also records the commands in a file which can then be used again (via a batch load) later. In `Emacs`, the shell window also records the session, including both input from the user and the system's response. The sessions in this tutorial were produced this way. These sessions are split into segments displayed in boxes with a number in their top right hand corner (to indicate their position in the complete session).

The interactions in these boxes should be understood as occurring in sequence. For example, variable bindings made in earlier boxes are assumed to persist to later ones. To enter the HOL system one types `hol` or `hol.unquote` to Unix, possibly preceded by path information if the HOL system's `bin` directory is not in one's path. The HOL system then prints a sign-on message and puts one into ML. The ML prompt is `-`, so lines beginning with `-` are typed by the user and other lines are the system's responses.

Here, as elsewhere in the *TUTORIAL*, we will be assuming use of `hol.unquote`.

1
```
$ bin/hol.unquote
Moscow ML version 1.44 (August 1999)
Enter `quit();' to quit.

-----------------------------------------------------------
           HOL98 [Taupo 3]

   For introductory HOL help, type: help "hol";
-----------------------------------------------------------

[closing file "/local/scratch/mn200/Work/hol98/tools/end-init.sml"]
- 1 :: [2,3,4,5];
> val it = [1, 2, 3, 4, 5] : int list
```

The ML expression `1 :: [2,3,4,5]` has the form e_1 *op* e_2 where e_1 is the expression `1` (whose value is the integer 1), e_2 is the expression `[2,3,4,5]` (whose value is a list of four integers) and *op* is the infixed operator '`::`' which is like Lisp's *cons* function. Other list processing functions include `hd` (*car* in Lisp), `tl` (*cdr* in Lisp) and `null` (*null* in Lisp). The semicolon '`;`' terminates a top-level phrase. The system's response is shown on the line starting with the > prompt. It consists of the value of the expression followed, after a colon, by its type. The ML type checker infers the type of expressions using methods invented by Robin Milner [7]. The type `int list` is the type of 'lists of integers'; `list` is a unary type operator. The type system of ML is very similar to the type system of the HOL logic which is explained in Chapter 3.

The value of the last expression evaluated at top-level in ML is always remembered in a variable called `it`.

2
```
- val l = it;
> val l = [1, 2, 3, 4, 5] : int list

- tl l;
> val it = [2, 3, 4, 5] : int list

- hd it;
> val it = 2 : int

- tl(tl(tl(tl(tl l))));
> val it = [] : int list
```

Following standard λ-calculus usage, the application of a function f to an argument x can be written without brackets as $f\ x$ (although the more conventional $f(x)$ is also allowed). The expression $f\ x_1\ x_2\ \cdots\ x_n$ abbreviates the less intelligible expression $(\cdots((f\ x_1)x_2)\cdots)x_n$ (function application is left associative).

Declarations have the form `val` x_1=e_1 `and` $\cdots$ `and` x_n=e_n and result in the value of each expression e_i being bound to the name x_i.

```
- val l1 = [1,2,3] and l2 = ["a","b","c"];                          3
> val l1 = [1, 2, 3] : int list
  val l2 = ["a", "b", "c"] : string list
```

ML expressions like `"a"`, `"b"`, `"foo"` etc. are *strings* and have type `string`. Any sequence of ASCII characters can be written between the quotes.[1] The function `explode` splits a string into a list of single characters, which are written like single character strings, with a # character prepended.

```
- explode "a b c";                                                  4
> val it = [#"a", #" ", #"b", #" ", #"c"] : char list
```

An expression of the form (e_1,e_2) evaluates to a pair of the values of e_1 and e_2. If e_1 has type σ_1 and e_2 has type σ_2 then (e_1,e_2) has type σ_1*σ_2. The first and second components of a pair can be extracted with the ML functions `#1` and `#2` respectively. If a tuple has more than two components, its n-th component can be extracted with a function #n.

The values `(1,2,3)`, `(1,(2,3))` and `((1,2), 3)` are all distinct and have types `int * int * int`, `int * (int * int)` and `(int * int) * int` respectively.

```
- val triple1 = (1,true,"abc");                                     5
> val triple1 = (1, true, "abc") : int * bool * string
- #2 triple1;
> val it = true : bool

- val triple2 = (1, (true, "abc"));
> val triple2 = (1, (true, "abc")) : int * (bool * string)
- #2 triple2;;
> val it = (true, "abc") : bool * string
```

The ML expressions `true` and `false` denote the two truth values of type `bool`.

ML types can contain the *type variables* `'a`, `'b`, `'c`, etc. Such types are called *polymorphic*. A function with a polymorphic type should be thought of as possessing all the types obtainable by replacing type variables by types. This is illustrated below with the function `zip`.

Functions are defined with declarations of the form `fun` $f\ v_1\ \ldots\ v_n$ = e where each v_i is either a variable or a pattern built out of variables.

The function `zip`, below, converts a pair of lists `(`[x_1,...,x_n], [y_1,...,y_n]`)` to a list of pairs `[`(x_1,y_1),...,(x_n,y_n)`]`.

[1]Newlines must be written as \n, and quotes as \".

6

```
- fun zip(l1,l2) =
    if null l1 orelse null l2 then []
    else (hd l1,hd l2) :: zip(tl l1,tl l2);
> val zip = fn : 'a list * 'b list -> ('a * 'b) list

- zip([1,2,3],["a","b","c"]);
> val it = [(1, "a"), (2, "b"), (3, "c")] : (int * string) list
```

Functions may be *curried*, i.e. take their arguments 'one at a time' instead of as a tuple. This is illustrated with the function `curried_zip` below:

7

```
- fun curried_zip l1 l2 = zip(l1,l2);
> val curried_zip = fn : 'a list -> 'b list -> ('a * 'b) list

- fun zip_num l2 = curried_zip [0,1,2] l2;
> val zip_num = fn : 'a list -> (int * 'a) list

- zip_num ["a","b","c"];
> val it = [(0, "a"), (1, "b"), (2, "c")] : (int * string) list
```

The evaluation of an expression either *succeeds* or *fails*. In the former case, the evaluation returns a value; in the latter case the evaluation is aborted and an *exception* is raised. This exception passed to whatever invoked the evaluation. This context can either propagate the failure (this is the default) or it can *trap* it. These two possibilities are illustrated below. An exception trap is an expression of the form e_1 `handle _ =>` e_2. An expression of this form is evaluated by first evaluating e_1. If the evaluation succeeds (i.e. doesn't fail) then the value of the whole expression is the value of e_1. If the evaluation of e_1 raises an exception, then the value of the whole is obtained by evaluating e_2.[2]

8

```
- 3 div 0;
! Uncaught exception:
! Div

- 3 div 0 handle _ => 0;
> val it = 0 : int
```

The sessions above are enough to give a feel for ML. In the next chapter, the logic supported by the HOL system (higher order logic) will be introduced, together with the tools in ML for manipulating it.

[2]This description of exception handling is actually a gross simplification of the way exceptions can be handled in ML; consult a proper text for a better explanation.

Chapter 3

The HOL Logic

The HOL system supports *higher order logic*. This is a version of predicate calculus with three main extensions:

- Variables can range over functions and predicates (hence ‘higher order’).
- The logic is *typed*.
- There is no separate syntactic category of *formulae* (terms of type `bool` fulfill their role).

3.1 Overview of higher order logic

It is assumed the reader is familiar with predicate logic. The syntax and semantics of the particular logical system supported by HOL is described in detail in *DESCRIPTION*. The table below summarizes the notation used.

Terms of the HOL Logic

Kind of term	*HOL notation*	*Standard notation*	*Description*
Truth	`T`	$\top$	*true*
Falsity	`F`	$\bot$	*false*
Negation	`~`t	$\neg t$	*not* t
Disjunction	t_1`\/`t_2	$t_1 \vee t_2$	t_1 *or* t_2
Conjunction	t_1`/\`t_2	$t_1 \wedge t_2$	t_1 *and* t_2
Implication	t_1`==>`t_2	$t_1 \Rightarrow t_2$	t_1 *implies* t_2
Equality	t_1`=`t_2	$t_1 = t_2$	t_1 *equals* t_2
$\forall$-quantification	`!`x`.`t	$\forall x.\ t$	*for all* $x : t$
$\exists$-quantification	`?`x`.`t	$\exists x.\ \ t$	*for some* $x : t$
ε-term	`@`x`.`t	$\varepsilon x.\ t$	*an* x *such that:* t
Conditional	`if` t `then` t_1 `else` t_2	$(t \rightarrow t_1, t_2)$	*if* t *then* t_1 *else* t_2

Note on HOL example sessions All of the examples below assume that the user is running `hol.unquote`, the executable for which is in the `bin/` directory along with that for `hol`. Further, the user needs to execute the following commands before starting the sessions below:

```
- load "arithmeticTheory";                                                  0
> val it = () : unit
- load "pairTheory";
> val it = () : unit
```

These commands load the HOL theories supporting pairs and arithmetic. When HOL starts up, it only knows about the basic boolean operators and quantifiers, so we augment it with these two theories to allow us more interesting examples.

Terms of the HOL logic are represented in ML by an *abstract type* called `term`. They are normally input between double back-quote marks. For example, the expression ``` ``x /\ y ==> z`` ``` evaluates in ML to a term representing x∧y⇒z. Terms can be manipulated by various built-in ML functions. For example, the ML function `dest_imp` with ML type `term -> term * term` splits an implication into a pair of terms consisting of its antecedent and consequent, and the ML function `dest_conj` of type `term -> term * term` splits a conjunction into its two conjuncts.

```
- ``x /\ y ==> z``;                                                         1
> val it = ``x /\ y ==> z`` : term

- dest_imp it;
> val it = (``x /\ y``, ``z``) : term * term

- dest_conj(#1 it);
> val it = (``x``, ``y``) : term * term
```

Terms of the HOL logic are quite similar to ML expressions, and this can at first be confusing. Indeed, terms of the logic have types similar to those of ML expressions. For example, ``` ``(1,2)`` ``` is an ML expression with ML type `term`. The HOL type of this term is `num # num`. By contrast, the ML expression ``` (``1``, ``2``) ``` has type `term * term`.

The types of HOL terms form an ML type called `hol_type`. Expressions having the form ``` ``: ··· `` ``` evaluate to logical types. The built-in function `type_of` has ML type `term->type` and returns the logical type of a term.

```
- ``(1,2)``;                                                          2
> val it = ``(1,2)`` : term

- type_of it;
> val it = ``:num # num`` : hol_type

- (``1``, ``2``);
> val it = (``1``, ``2``) : term * term

- type_of(#1 it);
> val it = ``:num`` : hol_type
```

To try to minimise confusion between the logical types of HOL terms and the ML types of ML expressions, the former will be referred to as *object language types* and the latter as *meta-language types*. For example, ``` ``(1,T)`` ``` is an ML expression that has meta-language type `term` and evaluates to a term with object language type ``` ``:num#bool`` ```.

```
- ``(1,T)``;                                                          3
> val it = ``(1,T)`` : term

- type_of it;
> val it = ``:num # bool`` : hol_type
```

HOL terms can be input using explicit *quotation*, as above, or they can be constructed using ML constructor functions. The function `mk_var` constructs a variable from a string and a type. In the example below, three variables of type `bool` are constructed. These are used later.

```
- val x = mk_var("x", ``:bool``)                                      4
  and y = mk_var("y", ``:bool``)
  and z = mk_var("z", ``:bool``);
> val x = ``x`` : term
  val y = ``y`` : term
  val z = ``z`` : term
```

The constructors `mk_conj` and `mk_imp` construct conjunctions and implications respectively.

```
- val t = mk_imp(mk_conj(x,y),z);                                     5
> val t = ``x /\ y ==> z`` : term
```

3.2 Terms

There are only four different kinds of terms:

1. Variables.

2. Constants.
3. Function applications: ``$t_1$ $t_2$``.
4. λ-abstractions: ``\\$x.t$``.

Both variables and constants have a name and a type; the difference is that constants cannot be bound by quantifiers, and their type is fixed when they are declared (see below). The type checking algorithm uses the types of constants to infer the types of variables in the same quotation. If there is not enough type information type variables will be guessed:

6
```
- ``~x``;
val it = ``~x`` : term

- ``x``;
<<HOL message: inventing new type variable names: 'a.>>
> val it = ``x`` : Term.term
- type_of it;
> val it = ``:'a`` : hol_type
```

In the first case, the HOL type checker used the known type `bool->bool` of ~ to deduce that the variable `x` must have type `bool`. In the second case, it cannot deduce the type of `x`. The default 'scope' of type information for type checking is a single quotation, so a type in one quotation cannot affect the type-checking of another. If there is not enough contextually-determined type information to resolve the types of all variables in a quotation, then the system will guess different type variables for all the unconstrained variables. Alternatively, it is possible to explicitly indicate the required types by using ``*term*:*type*`` as illustrated below.

7
```
- ``(x,y)``;
<<HOL message: inventing new type variable names: 'a, 'b.>>
> val it = ``(x,y)`` : term
- type_of it;
> val it = ``:'a # 'b`` : hol_type

- ``x:num``;
> val it = ``x`` : term
- type_of it;
> val it = ``:num`` : hol_type
```

Functions have types of the form σ_1->σ_2, where σ_1 and σ_2 are the types of the domain and range of the function, respectively.

8
```
- type_of ``$==>``;
> val it = ``:bool -> bool -> bool`` : hol_type

- type_of ``$+``;
> val it = ``:num -> num -> num`` : hol_type
```

Both + and ==> are infixes, so their use in contexts where they are not being used as such requires their prefixing by the $-sign. This is analogous to the way in which op is used in ML. The session below illustrates the use of these constants as infixes; it also illustrates object language versus meta-language types.

9
```
- ``(x + 1, t1 ==> t2)``;
> val it = ``(x + 1,t1 ==> t2)`` : term

- type_of it;
> val it = ``:num # bool`` : hol_type

- (``x=1``, ``t1==>t2``);
> val it = (``x = 1``, ``t1 ==> t2``) : term * term

- (type_of (#1 it), type_of (#2 it));
> val it = (``:bool``, ``:bool``) : hol_type * hol_type
```

The types of constants are declared in *theories*. This is described later.

An application $t_1\ t_2$ is badly typed if t_1 is not a function:

10
```
- ``1 2``;

Type inference failure: unable to infer a type for the application of

(1 :num)

to

(2 :num)

unification failure message: unify failed
! Uncaught exception:
! HOL_ERR <poly>
```

or if it is a function, but t_2 is not in its range:

11
```
- ``~1``;

Type inference failure: unable to infer a type for the application of

$~

to

(1 :num)

unification failure message: unify failed
! Uncaught exception:
! HOL_ERR <poly>
```

As before, the dollar in front of ~ indicates that the constant has a special syntactic status (in this case a non-standard precedence). Putting $ in front of any symbol causes the parser to ignore any special syntactic status (like being an infix) it might have.

```
- ``$==> t1 t2``;                                                    12
> val it = ``t1 ==> t2`` : term
- ``$/\ t1 t2``;
> val it = ``t1 /\ t2`` : term
```

Lambda-terms, or λ-terms, denote functions. The symbol '\' is used as an ASCII approximation to λ. Thus '\\$x.t$' should be read as '$\lambda x.\ t$'. For example, ``` ``\x. x+1`` ``` is a term that denotes the function $n \mapsto n{+}1$.

```
- ``\x. x + 1``;                                                     13
> val it = ``\x. x + 1`` : term

- type_of it;
> val it = ``:num -> num`` : hol_type
```

Other binding symbols in the logic are its two most important quantifiers: `!` and `?`, universal and existential quantifiers. For example, the logical statement that every number is either even or odd might be phrased as `!n. (n MOD 2 = 1) \/ (n MOD 2 = 0)`, while a version of Euclid's result about the infinitude of primes is: `!n. ?p. prime p /\ p > n`

Binding symbols such as these can be used over multiple symbols thus:

```
- ``\x y. (x, y * x)``;                                              14
> val it = ``\x y. (x,y * x)`` : term
- type_of it;
> val it = ``:num -> num -> num # num`` : hol_type

- ``!x y. x <= x + y``;
> val it = ``!x y. x <= x + y`` : term
```

3.3 Exceptions

Almost all of the HOL system's functions raise special `HOL_ERR` exceptions to signal abnormal or erroneous conditions. These exceptions do not print well by default, so the special `Raise` function is provided to make dealing with these exceptions easier:

```
- dest_conj ``p ==> q``;                                             15
! Uncaught exception:
! HOL_ERR <poly>

- dest_conj ``p ==> q`` handle e => Raise e;

Exception raised at Dsyntax.dest_conj:
not a conj
! Uncaught exception:
! HOL_ERR <poly>
```

The Raise function passes on all of the exceptions it sees; it does not affect the semantics of the computation at all. However, when passed a HOL_ERR exception, it prints out some useful information before passing it on to the next level.

Chapter 4

Euclid’s theorem

In this chapter, we prove in hol98 that for every number, there is a prime number that is larger, i.e., that the prime numbers form an infinite sequence. This proof has been excerpted and adapted from a much larger example due to John Harrison, in which he proved the $n = 4$ case of Fermat’s Last Theorem. The proof development will be performed using the facilities of `bossLib`, one of HOL’s many libraries and is intended to serve as an introduction to performing high-level interactive proofs in hol98. Many of the details may be difficult to grasp the novice reader; nonetheless, it is recommended that the example be followed through in order to gain a true taste of using HOL to prove non-trivial theorems.

Some tutorial descriptions of proof systems show the system performing amazing feats of automated theorem proving. In this example, we will *not* take this approach; instead, we try to show how one actually goes about the business of proving theorems in hol98: when more than one way to prove something is possible, we will consider the choices; when a difficulty rears its ugly head, we will attempt to explain how to fight one’s way clear.

One ‘drives’ hol98 by interacting with the ML top-level loop. In this interaction style, ML function calls are made to bring in already-established logical context (usually via `load`), to define new context (via `Hol_datatype` and `Define` from `bossLib`), and to perform proofs using the goalstack interface, and the proof tools from `bossLib` (or if they fail to do the job, from lower-level libraries).

First, we start the system. We will use make use of the quotation pre-processor in the example, so we invoke `.../<holdir>/bin/hol.unquote`. Then we load and open `bossLib`. This library provides high-level support for interactive proof. We also open the theory of arithmetic, since we will use some of its theorems. Finally, the function `by` from `bossLib` needs to be declared as infix before it can be used as such.

16
```
- load "bossLib";
> ...

- open bossLib arithmeticTheory;
> ...

- infix 8 by;
```

We specialize the rewriter provided by `bossLib` to a simplification set that knows about arithmetic. This is not necessary and only serves to make some of the proofs typeset more nicely.

```
- val ARW_TAC = RW_TAC arith_ss;                                          17

> val ARW_TAC =
    fn
    : Thm.thm list -> Term.term list * Term.term ->
      (Term.term list * Term.term) list * (Thm.thm list -> Thm.thm)
```

The ML type of `ARW_TAC` is *thm list* $\longrightarrow$ *tactic*. When `ARW_TAC` is applied to a list of theorems, the theorems will be added to `arith_ss` as rewrite rules. We will see that `ARW_TAC` is fairly knowledgeable about arithmetic.[1]

We now begin the formalization. In order to define the concept of *prime* number, we first need to define the *divisibility* relation:

```
- val divides = Define ‘divides a b = ?x. b = a * x‘;                     18

> Definition stored under "divides_def".
> val divides = |- !a b. divides a b = ?x. b = a * x : Thm.thm
```

The definition is added to the current theory with the name `divides_def`, and also returned from the invocation of `Define`. We take advantage of this and make an ML binding of the name `divides` to the definition. In the usual way of interacting with HOL, such an MLbinding is made for each definition and (useful) proved theorem: the MLenvironment is thus being used as a convenient place to hold definitions and theorems for later reference in the session.

We want to treat `divides` as a (right associative) infix:

```
- set_fixity "divides" (Infixr 450);                                      19
```

Now we can define the property of a number being *prime*: a number p is prime if and only if it is not equal to 1 and it has no divisors other than 1 and itself:

```
- val prime =                                                             20
    Define ‘prime p = ~(p=1) /\ !x. x divides p ==> (x=1) \/ (x=p)‘;

Definition stored under "prime_def".
> val prime =
    |- !p. prime p = ~(p = 1) /\ !x. x divides p ==> (x = 1) \/ (x = p)
    : Thm.thm
```

[1]Linear arithmetic especially: purely universal statements involving the operators `SUC`, $+$, $-$, numeric literals, $<$, $\leq$, $>$, $\geq$, $=$, and multiplication by numeric literals.

That concludes the definitions to be made. Now we “just” have to prove that there are an infinite number of primes. If we were coming to this problem fresh, then we would have to go through a not-well-understood and often tremendously difficult process of finding the right lemmas required to prove our target theorem.[2] Fortunately, we are working from a detailed and accurate source and can devote ourselves to the far simpler problem of explaining how to prove the required theorems.

The development will illustrate that there is often more than one way to tackle a HOL proof, even if one has only a single (informal) proof in mind. We often *find* the proof using `ARW_TAC` to unwind definitions and perform basic simplifications, i.e., to reduce the goal to its essence. Sometimes this proves the goal immediately. Often however, we are left with a goal that requires some study before one realizes what lemmas are needed to conclude the proof. Once these lemmas have been proven (or located in ancestor theories), `PROVE_TAC` can be invoked with them, with the expectation that it will find the right instantiations needed to finish the proof. (These two operations do not suffice to perform all proofs; in particular, our development will also need case analysis and induction.)

This raises the following question: how does one find the right lemmas to use? This is quite a problem, especially when the number of theorems in ancestor theories is large. There are are couple of possibilities: the help system can be used to look up definitions and theorems, as well as proof procedures; for example, an invocation of

```
help "arithmeticTheory"
```

will display all the definitions and theorems that have been stored in the theory of arithmetic. However, the complete name of the item being searched for must be known before the help system is useful. Alternatively, the functions in `DB` are often easier to use. `DB.match` allows the use of first order patterns to look for the relevant items, while `DB.find` will use fragments of names as keys with with to lookup information.

Once a proof of a proposition has been found, it is customary, although not necessary, to embark on a process of *revision*, in which the original sequence of tactics is composed into a single tactic. Sometimes the resulting tactic is much shorter, and more aesthetically pleasing in some sense. Some users spend a fair bit of time polishing these tactics, although there doesn’t seem much real benefit in doing so, since they are *ad hoc* proof recipes, one for each theorem. In the following, we will show how this is done in a few cases.

[2]This is of course a general problem in doing any kind of proof.

4.1 Divisibility

We start by proving a number of theorems about the `divides` relation. Each theorem is proved with a single invocation of `PROVE_TAC`. Both `ARW_TAC` and `PROVE_TAC` are quite powerful reasoners, and the choice of a reasoner in a particular situation is a matter of experience. In the following, the major reason that `PROVE_TAC` has been selected is that `divides` is defined by means of an existential quantifier, and `PROVE_TAC` is quite good at automatically instantiating existentials in the course of proof. For a simple example, consider proving $\forall x.\ x$ `divides` 0. A new proposition to be proved is entered to the proof manager via "g", which starts a fresh goalstack:

```
- g `!x. x divides 0`;                                                    21

> val it =
>    Proof manager status: 1 proof.
>    1. Incomplete:
>         Initial goal:
>         !x. x divides 0
>
>    : GoalstackPure.proofs
```

The proof manager tells us that it has only one proof to manage, and echoes the given goal. Now we expand the definition of `divides`. Notice that α-conversion takes place in order to keep distinct the x of the goal and the x in the definition of `divides`:

```
- e (ARW_TAC [divides]);                                                  22

> OK..
> 1 subgoal:
> val it =
>    ?x'. (x = 0) \/ (x' = 0)
```

It is of course quite easy to instantiate the existential quantifier by hand.

```
- e (EXISTS_TAC ``0``);                                                   23

> OK..
> 1 subgoal:
> val it =
>     (x = 0) \/ (0 = 0)
```

Then a simplification step finishes the proof.

```
- e (ARW_TAC []);                                                      24

> OK..
>
> Goal proved.
> |- (x = 0) \/ (0 = 0)
>
> Goal proved.
> |- ?x'. (x = 0) \/ (x' = 0)
> val it =
>     Initial goal proved.
>     |- !x. x divides 0
```

What just happened here? The application of ARW_TAC to the goal decomposed it to an empty list of subgoals; in other words the goal was proved by ARW_TAC. Once a goal has been proved, it is popped off the goalstack, prettyprinted to the output, and the theorem becomes available for use by the level of the stack. When all the sub-goals required by *that* level are proven, the corresponding goal at that level can be proven too. This 'unwinding' process continues until the stack is empty, or until it hits a goal with more than one remaining unproved subgoal. This process may be hard to visualize,[3] but that doesn't matter, since the goalstack was expressly written to allow the user to ignore such details.

If the three interactions are joined together with THEN to form a single tactic, we can try the proof again from the beginning (using the restart function) and this time it will take just one step:

```
- restart();                                                           25
>   ...

- e (ARW_TAC [divides] THEN EXISTS_TAC ``0`` THEN ARW_TAC[]);

> OK..
> val it =
>     Initial goal proved.
>     |- !x. x divides 0
```

We have seen one way to prove the theorem. However, there is another: one can let PROVE_TAC expand the definition of divides and find the required instantiation for x' from MULT_CLAUSES.

[3]Perhaps since we have used a stack to implement what is notionally a tree!

```
- restart();                                                          26
>   ...

- e (PROVE_TAC [divides, MULT_CLAUSES]);

> OK..
> Meson search level: .....
> val it =
>     Initial goal proved.
>     |- !x. x divides 0
```

In any case, having done our proof inside the goalstack package, we now want to have access to the theorem value that we have proved. We use the `top_thm` function to do this, and then use `drop` to dispose of the stack:

```
- val DIVIDES_0 = top_thm();                                          27

> val DIVIDES_0 = |- !x. x divides 0 : Thm.thm

- drop();
> OK..
> val it = There are currently no proofs. : GoalstackPure.proofs
```

We have used `PROVE_TAC` in this way to prove the following collection of theorems about `divides`. As mentioned previously, the theorems supplied to `PROVE_TAC` in the following proofs did not (usually) come from thin air: in most cases some exploratory work with `ARW_TAC` was done to open up definitions and see what lemmas would be required by `PROVE_TAC`.

(*DIVIDES_0*) `!x. x divides 0`
`PROVE_TAC [divides, MULT_CLAUSES]`

(*DIVIDES_ZERO*) `!x. 0 divides x = (x = 0)`
`PROVE_TAC [divides, MULT_CLAUSES]`

(*DIVIDES_ONE*) `!x. x divides 1 = (x = 1)`
`PROVE_TAC [divides, MULT_CLAUSES, MULT_EQ_1]`

(*DIVIDES_REFL*) `!x. x divides x`
`PROVE_TAC [divides, MULT_CLAUSES]`

(*DIVIDES_TRANS*) `!a b c. a divides b /\ b divides c ==> a divides c`
`PROVE_TAC [divides, MULT_ASSOC]`

(*DIVIDES_ADD*) `!d a b. d divides a /\ d divides b ==> d divides (a+b)`
`PROVE_TAC [divides,LEFT_ADD_DISTRIB]`

(*DIVIDES_SUB*)
```
!d a b. d divides a /\ d divides b ==> d divides (a-b)
PROVE_TAC [divides, LEFT_SUB_DISTRIB]
```

(*DIVIDES_ADDL*)
```
!d a b. d divides a /\ d divides (a+b) ==> d divides b
PROVE_TAC [ADD_SUB, ADD_SYM, DIVIDES_SUB]
```

(*DIVIDES_LMUL*)
```
!d a x. d divides a ==> d divides (x * a)
PROVE_TAC [divides, MULT_ASSOC, MULT_SYM]
```

(*DIVIDES_RMUL*)
```
!d a x. d divides a ==> d divides (a * x)
PROVE_TAC [MULT_SYM, DIVIDES_LMUL]
```

We'll assume that the above proofs have been performed, and that the appropriate ML names have been given to all of the theorems. Now we encounter a lemma about divisibility that doesn't succumb to a single invocation of `PROVE_TAC`:

(*DIVIDES_LE*)
```
!m n. m divides n ==> m <= n \/ (n = 0)
ARW_TAC [divides]
   THEN Cases_on ‘x‘
   THEN ARW_TAC [MULT_CLAUSES]
```

Let's see how this is proved. The easiest way to start is to simplify with the definition of `divides`:

28
```
- g ‘!m n . m divides n ==> m <= n \/ (n = 0)‘;
>   ...

- e (ARW_TAC [divides]);

> 1 subgoal:
> val it =
>     m <= m * x \/ (m * x = 0)
```

Considering the goal, we basically have three choices: (1) find a collection of lemmas that together imply the goal and use `PROVE_TAC`; (2) do a case split on m; or (3) do a case split on x. The first doesn't seem simple, because the goal doesn't really fit in the 'shape' of any pre-proved theorem(s) that the author knows about. Although option (2) will be rejected in the end, let's try it anyway. To perform the case split, we use `Cases_on`, which stands for "find the given term in the goal and do a case split on the possible means of building it out of datatype constructors". Since the occurrence of m in the goal has type num, the cases considered will be whether m is 0 or a successor.

29
```
- e (Cases_on ‘m‘);

> OK..
> 2 subgoals:
> val it =
>     SUC n <= SUC n * x \/ (SUC n * x = 0)
>
>    0 <= 0 * x \/ (0 * x = 0)
```

The first subgoal (the last one printed) is trivial:

```
- e (ARW_TAC []);                                              30

> OK..
>
> Goal proved.
>   ...
>
> Remaining subgoals:
> val it =
>     SUC n <= SUC n * x \/ (SUC n * x = 0)
```

Let's try `ARW_TAC` again:

```
- e (ARW_TAC []);                                              31

> OK..
> 1 subgoal:
> val it =
>     SUC n <= SUC n * x \/ (x = 0)
```

The right disjunct has been simplified; however, the left disjunct has failed to expand the definition of multiplication in the expression SUC $n * x$, which would have been convenient. Why not, when `arith_ss` and hence `ARW_TAC` is supposed to be expert in arithmetic? The answer is that the recursive clauses for addition and multiplication are not in `arith_ss` because uncontrolled application of them by the rewriter seems, in general, to make some proofs *more* complicated, rather than simpler. OK, so let's add in the definition of multiplication. This uncovers a new problem: how to locate this definition. The function

```
   DB.match : string list -> term
                 -> ((string * string) * (thm * DB.class)) list
```

is often helpful for such tasks. It takes a list of theory names, and a pattern, and looks in the list of theories for any theorem, definition, or axiom that has an instance of the pattern as a subterm. If the list of theory names is empty, then all loaded theories are included in the search. Let's look in the theory of arithmetic for the subterm to be rewritten.

```
- DB.match ["arithmetic"] (Term`SUC n * x`);                                32

> val it =
>    [(("arithmetic", "FACT"),
>      (|- (FACT 0 = 1) /\ !n. FACT (SUC n) = SUC n * FACT n, DB.Def)),
>     (("arithmetic", "LESS_MULT_MONO"),
>      (|- !m i n. SUC n * m < SUC n * i = m < i, DB.Thm)),
>     (("arithmetic", "MULT"),
>      (|- (!n. 0 * n = 0) /\ !m n. SUC m * n = m * n + n, DB.Def)),
>     (("arithmetic", "MULT_CLAUSES"),
>      (|- !m n.
>            (0 * m = 0) /\ (m * 0 = 0) /\ (1 * m = m) /\ (m * 1 = m) /\
>            (SUC m * n = m * n + n) /\ (m * SUC n = m + m * n), DB.Thm)),
>     (("arithmetic", "MULT_LESS_EQ_SUC"),
>      (|- !m n p. m <= n = SUC p * m <= SUC p * n, DB.Thm)),
>     (("arithmetic", "MULT_MONO_EQ"),
>      (|- !m i n. (SUC n * m = SUC n * i) = m = i, DB.Thm)),
>     (("arithmetic", "ODD_OR_EVEN"),
>      (|- !n. ?m. (n = SUC (SUC 0) * m) \/ (n = SUC (SUC 0) * m + 1), DB.Thm))]
```

For some, this returns slightly too much information; however, we can focus the search by stipulating that the pattern look like a rewrite rule:

```
- DB.match [] (Term`SUC n * x = M`);                                        33

> val it =
>     [(("arithmetic", "MULT"),
>       (|- (!n. 0 * n = 0) /\ !m n. SUC m * n = m * n + n, DB.Def)),
>      (("arithmetic", "MULT_CLAUSES"),
>       (|- !m n.
>             (0 * m = 0) /\ (m * 0 = 0) /\ (1 * m = m) /\ (m * 1 = m) /\
>             (SUC m * n = m * n + n) /\ (m * SUC n = m + m * n), DB.Thm)),
>      (("arithmetic", "MULT_MONO_EQ"),
>       (|- !m i n. (SUC n * m = SUC n * i) = m = i, DB.Thm))]
```

Either `arithmeticTheory.MULT` or `arithmeticTheory.MULT_CLAUSES` would be acceptable; we choose the latter.

```
- e (ARW_TAC [MULT_CLAUSES]);                                               34

> OK..
> 1 subgoal:
> val it =
>     SUC n <= x + n * x \/ (x = 0)
```

Now we see that, in order to make progress in the proof, we will have to do a case split on x anyway, and that we should have split on it originally. Hence we backup. We will have to backup (undo) four times:

```
- b();                                                                    35
> val it =
    SUC n <= SUC n * x \/ (x = 0)

- b();
> val it =
    SUC n <= SUC n * x \/ (SUC n * x = 0)

- b();
> val it =
    SUC n <= SUC n * x \/ (SUC n * x = 0)


    0 <= 0 * x \/ (0 * x = 0)

- b();
> val it =
    m <= m * x \/ (m * x = 0)
```

Now we can go forward and do case analysis on x. We will also make a compound tactic invocation, since we already know that we'll have to invoke `ARW_TAC` in both branches of the case split. This can be done using `THEN`. When t_1 `THEN` t_2 is applied to a goal g, first t_1 is applied to g, giving a list of new subgoals, then t_2 is applied to each member of the list. All goals resulting from these applications of t_2 are gathered together and returned.

```
- e (Cases_on ‘x‘ THEN ARW_TAC [MULT_CLAUSES]);                          36

> OK..
>
> Goal proved.
> |- m <= m * x \/ (m * x = 0)
> val it =
>     Initial goal proved.
>     |- !m n. m divides n ==> m <= n \/ (n = 0)
```

That was easy! Obviously making a case split on x was the right choice. The process of *finding* the proof has now finished, and all that remains is for the proof to be packaged up into the single tactic we saw above. Rather than use `top_thm` and the goalstack, we can bypass it and use the `store_thm` function. This function takes a string, a term and a tactic and applies the tactic to the term to get a theorem, and then stores the theorem in the current theory under the given name.

```
- val DIVIDES_LE = store_thm (                                              37
     "DIVIDES_LE",
     ``!m n. m divides n ==> m <= n \/ (n = 0)``,
     ARW_TAC [divides]
       THEN Cases_on `x`
       THEN ARW_TAC [MULT]);

> val DIVIDES_LE = |- !m n. m divides n ==> m <= n \/ (n = 0) : Thm.thm
```

Storing theorems in our script record of the session in this style (rather than with the goalstack) results in a more concise script, and also makes it easier to turn our script into a theory file, as we do in section 4.5.

4.1.1 Divisibility and factorial

The next lemma, *DIVIDES_FACT*, says that every number greater than 0 and less-than-or-equal-to n divides the factorial of n. Factorial is found at `arithmeticTheory.FACT` and has been defined by primitive recursion:

(*FACT*)
```
(FACT 0 = 1) /\
(!n. FACT (SUC n) = SUC n * FACT n)
```

A polished proof of *DIVIDES_FACT* is the following:

(*DIVIDES_FACT*)
```
!m n. 0 < m /\ m <= n ==> m divides (FACT n)
----------------------------------------------
ARW_TAC [LESS_EQ_EXISTS]
 THEN Induct_on `p`
 THEN ARW_TAC [FACT,ADD_CLAUSES]
 THENL [Cases_on `m`, ALL_TAC]
 THEN PROVE_TAC [FACT, DECIDE `!x. ~(x < x)`,
             DIVIDES_RMUL, DIVIDES_LMUL, DIVIDES_REFL]
```

We will examine this proof in detail, so we should first attempt to understand why the theorem is true. What's the underlying intuition? Suppose $0 < m \leq n$, and so `FACT` $n = 1 * \cdots * m * \cdots * n$. To show m `divides` (`FACT` n) means exhibiting a q such that $q * m =$ `FACT` n. Thus $q =$ `FACT` $n \div m$. If we were to take this approach to the proof, we would end up having to find and apply lemmas about $\div$. This seems to take us a little out of our way; isn't there a proof that doesn't use division? Well yes, we can prove the theorem by induction on $n - m$: in the base case, we will have to prove n `divides` (`FACT` n), which ought to be easy; in the inductive case, the inductive hypothesis seems like it should give us what we need. This last is a bit vague, because we are trying to mentally picture a slightly complicated formula, but we can rely on the system to accurately calculate the cases of the induction for us. If the inductive case turns out to be not what we expect, we will have to re-think our approach.

```
- g `!m n. 0 < m /\ m <= n ==> m divides (FACT n)`;                    38

> val it =
>    Proof manager status: 1 proof.
>    1. Incomplete:
>         Initial goal:
>         !m n. 0 < m /\ m <= n ==> m divides FACT n
```

Instead of directly inducting on $n - m$, we will induct on a witness variable, obtained by use of the theorem LESS_EQ_EXISTS.

```
- LESS_EQ_EXISTS;                                                      39
> val it = |- !m n. m <= n = (?p. n = m + p) : Thm.thm

- e (ARW_TAC [LESS_EQ_EXISTS]);

> OK..
> 1 subgoal:
> val it =
>    m divides FACT (m + p)
>    ------------------------------------
>       0 < m
```

Now we induct on p:

```
- e (Induct_on `p`);                                                   40

> OK..
> 2 subgoals:
> val it =
>     m divides FACT (m + SUC p)
>     ------------------------------------
>       0.  0 < m
>       1.  m divides FACT (m + p)
>
>     m divides FACT (m + 0)
>    ------------------------------------
>       0 < m
```

The first goal can obviously be simplified:

```
- e (ARW_TAC []);                                                      41

> OK..
> 1 subgoal:
> val it =
>     m divides FACT m
>     ------------------------------------
>       0 < m
```

Now we can do a case analysis on m: if it is 0, we have a trivial goal; if it is a successor, then we can use the definition of FACT and the theorems DIVIDES_RMUL and DIVIDES_REFL.

```
- e (Cases_on ‘m‘);                                                      42

> OK..
> 2 subgoals:
> val it =
>     SUC n divides FACT (SUC n)
>     ------------------------------------
>       0 < SUC n
>
>     0 divides FACT 0
>     ------------------------------------
>       0 < 0
```

Here the first sub-goal goal has an assumption that is false. We can demonstrate this to the system by using the DECIDE function to prove a simple fact about arithmetic (namely, that no number x is less than itself), and then passing the resulting theorem to PROVE_TAC, which can combine this with the contradictory assumption.

```
- e (PROVE_TAC [DECIDE ‘!x. ~(x < x)‘]);                                 43

> OK..
> Meson search level: ..
>
> Goal proved.
>  [.] |- 0 divides FACT 0
>
> Remaining subgoals:
> val it =
>     SUC n divides FACT (SUC n)
>     ------------------------------------
>       0 < SUC n
```

Using the theorems identified above, this, the second sub-goal, can be proved with ARW_TAC.

```
- e (ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);                        44

> OK..
>
> Goal proved.   ...
>
> Remaining subgoals:
> val it =
>     m divides FACT (m + SUC p)
>     ------------------------------------
>       0.  0 < m
>       1.  m divides FACT (m + p)
```

Note that this last step (the invocation of ARW_TAC) could also have been accomplished with PROVE_TAC:

```
- b();                                                                  45

> ...

- e (PROVE_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);
> OK..
>
> Goal proved.  ...
```

Now we have finished the base case of the induction and can move to the step case. An obvious thing to try is simplification with the definitions of addition and factorial:

```
- e (ARW_TAC [FACT, ADD_CLAUSES]);                                      46

> OK..
> 1 subgoal:
> val it =
>     m divides SUC (m + p) * FACT (m + p)
>     ------------------------------------
>       0.  0 < m
>       1.  m divides FACT (m + p)
```

And now, by DIVIDES_LMUL and the inductive hypothesis, we are done:

```
- e (PROVE_TAC [DIVIDES_LMUL]);                                         47

> OK..
> Meson search level: ...
> Goal proved.
>   ...
> val it =
>     Initial goal proved.
>     |- !m n. 0 < m /\ m <= n ==> m divides FACT n
```

We have finished the search for the proof, and now turn to the task of making a single tactic out of the sequence of tactic invocations we have just made. We assume that the sequence of invocations has been kept track of in a file or a text editor buffer. We would thus have something like the following:

```
    e (ARW_TAC [LESS_EQ_EXISTS]);
    e (Induct_on ‘p‘);
    (*1*)
    e (ARW_TAC []);
    e (Cases_on ‘m‘);
    (*1.1*)
    e (PROVE_TAC [DECIDE ‘!x. ~(x < x)‘]);
    (*1.2*)
```

```
e (ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);
(*2*)
e (ARW_TAC [FACT, ADD_CLAUSES]);
e (PROVE_TAC [DIVIDES_LMUL]);
```

We have added a numbering scheme to keep track of the branches in the proof. We can stitch the above directly into the following compound tactic:

```
ARW_TAC [LESS_EQ_EXISTS]
 THEN Induct_on ‘p‘
 THENL [ARW_TAC [] THEN Cases_on ‘m‘
        THENL [PROVE_TAC [DECIDE ‘!x. ~(x < x)‘],
               ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]],
        ARW_TAC [FACT, ADD_CLAUSES] THEN PROVE_TAC [DIVIDES_LMUL]]
```

This can be tested to see that we have made no errors:

```
- restart();                                                                48

> ...

- e (ARW_TAC [LESS_EQ_EXISTS]
       THEN Induct_on ‘p‘
       THENL [ARW_TAC [] THEN Cases_on ‘m‘
              THENL [PROVE_TAC [DECIDE ‘!x. ~(x < x)‘],
                     ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]],
              ARW_TAC [FACT, ADD_CLAUSES] THEN PROVE_TAC [DIVIDES_LMUL]]);
> OK..
> Meson search level: ...
> Meson search level: ..
> val it =
>     Initial goal proved.
>     |- !m n. 0 < m /\ m <= n ==> m divides FACT n
```

For many users, this would be the end of dealing with this proof: the tactic can now be packaged into an invocation of `prove`[4] or `store_thm` and that would be the end of it. However, another class of user would notice that this tactic could be shortened.

To start, both arms of the induction start with an invocation of `ARW_TAC`, and the semantics of `THEN` allow us to merge the occurrences of `ARW_TAC` above the `THENL`. The recast tactic is

```
ARW_TAC [LESS_EQ_EXISTS]
  THEN Induct_on ‘p‘
  THEN ARW_TAC [FACT, ADD_CLAUSES]
```

[4]The `prove` function takes a term and a tactic and attempts to prove the term using the supplied tactic. It returns the proved theorem if the tactic succeeds. It doesn’t save the theorem to the developing theory.

```
      THENL [Cases_on ‘m‘
              THENL [PROVE_TAC [DECIDE ‘!x. ~(x < x)‘],
                     ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]],
             PROVE_TAC [DIVIDES_LMUL]]
```

(Of course, when a tactic has been revised, it should be tested to see if it still proves the goal!) Now recall that the use of ARW_TAC in the base case could be replaced by a call to PROVE_TAC. Thus it seems possible to merge the two sub-cases of the base case into a single invocation of PROVE_TAC:

```
    ARW_TAC [LESS_EQ_EXISTS]
      THEN Induct_on ‘p‘
      THEN ARW_TAC [FACT, ADD_CLAUSES]
      THENL [Cases_on ‘m‘
              THEN PROVE_TAC [DECIDE ‘!x. ~(x < x)‘,
                              FACT, DIVIDES_RMUL, DIVIDES_REFL],
             PROVE_TAC [DIVIDES_LMUL]]
```

Finally, pushing this dubious revisionism nearly to its limit, we'd like there to be only a single invocation of PROVE_TAC to finish the proof off. The semantics of THEN and ALL_TAC come to our rescue: we will split on the construction of m in the base case, as in the current incarnation of the tactic, but we will let the inductive case pass unaltered through the THENL. This is achieved by using ALL_TAC, which is a tactic that acts as an identity function on the goal.

```
    ARW_TAC [LESS_EQ_EXISTS]
      THEN Induct_on ‘p‘
      THEN ARW_TAC [FACT, ADD_CLAUSES]
      THENL [Cases_on ‘m‘, ALL_TAC]
      THEN PROVE_TAC [DECIDE ‘!x. ~(x < x)‘, FACT,
                      DIVIDES_RMUL, DIVIDES_REFL, DIVIDES_LMUL]
```

The result is that there will be three subgoals emerging from the THENL: the two sub-cases in the base case and the unaltered step case. Each is proved with a call to PROVE_TAC. We have now finished our exercise in tactic polishing.

4.1.2 Divisibility and factorial (again!)

In the previous proof, we made an initial simplification step in order to expose a variable upon which to induct. However, the proof is really by induction on $n - m$. Can we express this directly? The answer is a qualified yes: the induction can be naturally stated, but it leads to somewhat less natural goals.

```
- restart();                                                          49

- e (Induct_on ‘n - m‘);

> OK..
2 subgoals:
> val it =
>     !n m. (SUC v = n - m) ==> 0 < m /\ m <= n ==> m divides FACT n
>     ------------------------------------
>       !n m. (v = n - m) ==> 0 < m /\ m <= n ==> m divides FACT n
>
>     !n m. (0 = n - m) ==> 0 < m /\ m <= n ==> m divides FACT n
```

This is slighly hard to read, so we use `STRIP_TAC` and `REPEAT` to move the antecedents of the goals to the assumptions. Use of `THEN` ensures that the tactic gets applied in both branches of the induction.

```
- b();                                                                50
  ...

- e (Induct_on ‘n - m‘ THEN REPEAT STRIP_TAC);

> OK..
> 2 subgoals:
> val it =
>     m divides FACT n
>     ------------------------------------
>       0.  !n m. (v = n - m) ==> 0 < m /\ m <= n ==> m divides FACT n
>       1.  SUC v = n - m
>       2.  0 < m
>       3.  m <= n
>
>     m divides FACT n
>     ------------------------------------
>       0.  0 = n - m
>       1.  0 < m
>       2.  m <= n
```

Looking at the first goal, we reason that if $0 = n - m$ and $m \leq n$, then $m = n$. We can prove this fact, and add it to the hypotheses by use of the infix operator “`by`”:

```
- e (‘m = n‘ by DECIDE_TAC);                                          51
OK..
1 subgoal:
> val it =
    m divides FACT n
    ------------------------------------
      0.  0 = n - m
      1.  0 < m
      2.  m <= n
      3.  m = n
```

We can now use ARW_TAC to propagate the newly derived equality throughout the goal.

```
- e (ARW_TAC []);                                                        52

> OK..
> 1 subgoal:
> val it =
>     m divides FACT m
>     ------------------------------------
>       0.  0 = m - m
>       1.  0 < m
>       2.  m <= m
```

At this point in the previous proof we did a case analysis on m. However, we already have the hypothesis that m is positive. Thus we know that m is the successor of some number k. We might wish to assert this fact with an invocation of "by" as follows:

```
‘?k.  m = SUC k‘ by <tactic>
```

But what is the tactic? If we try DECIDE_TAC, it will fail since it doesn't handle existential statements. An application of ARW_TAC will also prove to be unsatisfactory. What to do?

When such situations occur, it is often best to start a new proof for the required lemma. This can be done simply by invoking "g" again. A new goalstack will be created and stacked upon the current one[5] and an overview of the extant proof attempts will be printed:

```
- g ‘!m. 0 < m ==> ?k. m = SUC k‘;                                       53

> val it =
>     Proof manager status: 2 proofs.
>     2. Incomplete:
>          Initial goal:
>          !m n. 0 < m /\ m <= n ==> m divides FACT n
>
>
>          Current goal:
>          m divides FACT m
>          ------------------------------------
>            0.  0 = m - m
>            1.  0 < m
>            2.  m <= m
>
>     1. Incomplete:
>          Initial goal:
>          !m. 0 < m ==> ?k. m = SUC k
```

[5]This stacking of proof attempts (goalstacks) is different than the stacking of goals and justifications inside a particular goalstack.

Our new goal can be proved quite quickly. Once we have proved it, we can bind it to an ML name and use it in the previous proof, by some sleight of hand with the "before"[6] function.

```
- e (Cases THEN ARW_TAC []);                                                54

> OK..
> val it =
>     Initial goal proved.
>     |- !m. 0 < m ==> ?k. m = SUC k

- val lem = top_thm() before drop();

> OK..
> val lem = |- !m. 0 < m ==> ?k. m = SUC k : Thm.thm
```

Now we can return to the main thread of the proof. The state of the current sub-goal of the proof can be displayed using the function "p".

```
- p ();                                                                     55

> val it =
>     m divides FACT m
>     ------------------------------------
>       0.  0 = m - m
>       1.  0 < m
>       2.  m <= m
```

Now we can use lem in the proof. Somewhat opportunistically, we will tack on the invocation used in the earlier proof at (roughly) the same point, hoping that it will solve the goal:

```
- e (‘?k. m = SUC k‘ by                                                     56
      PROVE_TAC [lem] THEN ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);

> OK..
> Meson search level: ...
>
> Goal proved.   ...
>
> Remaining subgoals:
> val it =
>     m divides FACT n
>     ------------------------------------
>       0.  !n m. (v = n - m) ==> 0 < m /\ m <= n ==> m divides FACT n
>       1.  SUC v = n - m
>       2.  0 < m
>       3.  m <= n
```

[6]An infix version of the K combinator, defined by fun (x before y) = x.

It does! That takes care of the base case. For the induction step, things look a bit more difficult than in the earlier proof. However, we can make progress by realizing that the hypotheses imply that $0 < n$ and so, again by `lem`, we can transform n into a successor, thus enabling the unfolding of `FACT`, as in the previous proof:

```
- e (`0 < n` by DECIDE_TAC THEN `?k. n = SUC k` by PROVE_TAC [lem]);     57

> OK..
> Meson search level: ...
> 1 subgoal:
> val it =
>     m divides FACT n
>     ------------------------------------
>       0.  !n m. (v = n - m) ==> 0 < m /\ m <= n ==> m divides FACT n
>       1.  SUC v = n - m
>       2.  0 < m
>       3.  m <= n
>       4.  0 < n
>       5.  n = SUC k
```

The proof now finishes in much the same manner as the previous one:

```
- e (ARW_TAC [FACT, DIVIDES_LMUL]);                                         58

> OK..
>
> Goal proved.  ...
> val it =
>     Initial goal proved.
>     |- !m n. 0 < m /\ m <= n ==> m divides FACT n
```

We leave the details of stitching the proof together to the interested reader.

4.2 Primality

Now we move on to establish some facts about the primality of the first few numbers: 0 and 1 are not prime, but 2 is. Also, all primes are positive. These are all quite simple to prove.

(*NOT_PRIME_0*)
```
~prime 0
ARW_TAC [prime,DIVIDES_0]
```

(*NOT_PRIME_1*)
```
~prime 1
ARW_TAC [prime]
```

(*PRIME_2*)
```
prime 2
-------------------------------------------------------------
ARW_TAC [prime]
 THEN PROVE_TAC [DIVIDES_LE, DIVIDES_ZERO,
                 DECIDE ‘~(2=1)‘, DECIDE ‘~(2=0)‘,
                 DECIDE ‘x <= 2 = (x=0) \/ (x=1) \/ (x=2)‘]
```

(*PRIME_POS*)
```
!p. prime p ==> 0<p
----------------------------------
Cases THEN ARW_TAC[NOT_PRIME_0]
```

4.3 Existence of prime factors

Now we are in position to prove a more substantial lemma: every number other than 1 has a prime factor. The proof proceeds by a *complete induction* on n. Complete induction is necessary since a prime factor won't be the predecessor. After induction, the proof splits into cases on whether n is prime or not. The first case (n is prime) is trivial. In the second case, there must be an x that divides n, and x is not 1 or n. By *DIVIDES_LE*, $n = 0$ or $x \leq n$. If $n = 0$, then 2 is a prime that divides 0. On the other hand, if $x \leq n$, there are two cases: if $x < n$ then we can use the inductive hypothesis and by transitivity of divides we are done; otherwise, $x = n$ and then we have a contradiction with the fact that x is not 1 or n. The polished tactic is the following:

(*PRIME_FACTOR*)
```
!n. ~(n = 1) ==> ?p. prime p /\ p divides n
---------------------------------------------------------
completeInduct_on ‘n‘
  THEN ARW_TAC []
  THEN Cases ‘prime n‘ THENL
  [PROVE_TAC [DIVIDES_REFL],
   ‘?x. x divides n /\ ~(x=1) /\ ~(x=n)‘
    by PROVE_TAC[prime]
     THEN PROVE_TAC [LESS_OR_EQ, PRIME_2,
                     DIVIDES_LE,DIVIDES_TRANS,DIVIDES_0]]
```

We start by invoking complete induction. This gives us an inductive hypothesis that holds at every number m strictly smaller than n:

```
- g ‘!n. ~(n = 1) ==> ?p. prime p /\ p divides n‘;                        59

- e (completeInduct_on ‘n‘);

> OK..
> 1 subgoal:
> val it =
>     ~(n = 1) ==> ?p. prime p /\ p divides n
>     ------------------------------------
>       !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
```

We can move the antecedent to the hypotheses and make our case split. Notice that the term given to `Cases_on` need not occur in the goal:

```
- e (ARW_TAC [] THEN Cases_on ‘prime n‘);                                60
OK..
2 subgoals:
> val it =
    ?p. prime p /\ p divides n
    ------------------------------------
      0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
      1.  ~(n = 1)
      2.  ~prime n

    ?p. prime p /\ p divides nnn
    ------------------------------------
      0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
      1.  ~(n = 1)
      2.  prime n
```

As mentioned, the first case is proved with the reflexivity of divisibility:

```
- e (PROVE_TAC [DIVIDES_REFL]);                                          61

> OK..
> Meson search level: ...
>
> Goal proved.  ...
```

In the second case, we can get a divisor of n that isn't 1 or n (since n is not prime):

```
- e (‘?x. x divides n /\ ~(x=1) /\ ~(x=n)‘ by PROVE_TAC [prime]);        62

> OK..
> Meson search level: ...........
> 1 subgoal:
> val it =
>     ?p. prime p /\ p divides n
>     ------------------------------------
>       0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
>       1.  ~(n = 1)
>       2.  ~prime n
>       3.  x divides n
>       4.  ~(x = 1)
>       5.  ~(x = n)
```

At this point, the polished tactic simply invokes `PROVE_TAC` with a collection of theorems. We will attempt a more detailed exposition. Given the hypotheses, and by *DIVIDES_LE*, we can assert $x < n \vee n = 0$ and thus split the proof into two cases:

```
- e (`x < n \/ (n=0)` by PROVE_TAC [DIVIDES_LE,LESS_OR_EQ]);          63

> OK..
> Meson search level: ......
> 2 subgoals:
> val it =
>     ?p. prime p /\ p divides n
>     ------------------------------------
>       0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
>       1.  ~(n = 1)
>       2.  ~prime n
>       3.  x divides n
>       4.  ~(x = 1)
>       5.  ~(x = n)
>       6.  n = 0
>
>     ?p. prime p /\ p divides n
>     ------------------------------------
>       0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
>       1.  ~(n = 1)
>       2.  ~prime n
>       3.  x divides n
>       4.  ~(x = 1)
>       5.  ~(x = n)
>       6.  x < n
```

In the first subgoal, we can see that the antecedents of the inductive hypothesis are met and so x has a prime divisor. We can then use the transitivity of divisibility to get the fact that this divisor of x is also a divisor of n, thus finishing this branch of the proof:

```
- e (PROVE_TAC [DIVIDES_TRANS]);                                       64

> OK..
> Meson search level: .........
> Goal proved.
```

The remaining goal can be clarified by simplification:

```
- e (ARW_TAC []);                                                         65

> OK..
> 1 subgoal:
> val it =
>     ?p. prime p /\ p divides 0
>     ------------------------------------
>       0.  !m. m < 0 ==> ~(m = 1) ==> ?p. prime p /\ p divides m
>       1.  ~(0 = 1)
>       2.  ~prime 0
>       3.  x divides 0
>       4.  ~(x = 1)
>       5.  ~(x = 0)

- DIVIDES_0;

> val it = |- !x. x divides 0 : Thm.thm

- e (ARW_TAC [it]);

> OK..
> 1 subgoal:
> val it =
>     ?p. prime p
>     ------------------------------------
>       0.  !m. m < 0 ==> ~(m = 1) ==> ?p. prime p /\ p divides m
>       1.  ~(0 = 1)
>       2.  ~prime 0
>       3.  x divides 0
>       4.  ~(x = 1)
>       5.  ~(x = 0)
```

The two steps of exploratory simplification have led us to a state where all we have to do is exhibit a prime. And we already have one to hand:

```
- e (PROVE_TAC [PRIME_2]);                                                66

> OK..
> Meson search level: ..
>
> Goal proved.   ...
> val it =
>     Initial goal proved.
>     |- !n. ~(n = 1) ==> ?p. prime p /\ p divides n
```

Again, work now needs to be done to compose and perhaps polish a single tactic from the individual proof steps, but we will not describe it. Instead we move forward, because our ultimate goal is in reach.

4.4 Euclid's theorem

Theorem. Every number has a prime greater than it.
Informal proof.
Suppose the opposite; then there's an n such that all p greater than n are not prime. Consider FACT$(n) + 1$: it's not equal to 1 so, by *PRIME_FACTOR*, there's a prime p that divides it. Note that p also divides FACT(n) because $p \leq n$. By *DIVIDES_ADDL*, we have $p = 1$. But then p is not prime, which is a contradiction.
End of proof.
A HOL rendition of the proof may be given as follows:

(*EUCLID*)
```
!n. ?p. n < p /\ prime p
------------------------------------------------------------
SPOSE_NOT_THEN STRIP_ASSUME_TAC
  THEN MP_TAC (SPEC ``FACT n + 1`` PRIME_FACTOR)
  THEN ARW_TAC [FACT_LESS, DECIDE `~(x=0) = 0<x`]
  THEN PROVE_TAC [NOT_PRIME_1, NOT_LESS, PRIME_POS,
                  DIVIDES_FACT, DIVIDES_ADDL, DIVIDES_ONE]
```

Let's prise this apart and look at it in some detail. A proof by contradiction can be started by using the bossLib function SPOSE_NOT_THEN. With it, one assumes the negation of the current goal and then uses that in an attempt to prove falsity (F). The assumed negation $\neg(\forall n.\ \exists p.\ n < p \wedge \texttt{prime}\ p)$ is simplified a bit into $\exists n.\ \forall p.\ n < p \supset \neg\,\texttt{prime}\ p$ and then is passed to the tactic STRIP_ASSUME_TAC. This moves its argument to the assumption list of the goal after eliminating the existential quantification on n.

67
```
- g `!n. ?p. n < p /\ prime p`;

- e (SPOSE_NOT_THEN STRIP_ASSUME_TAC);

> OK..
> 1 subgoal:
> val it =
>     F
>     ------------------------------------
>       !p. n < p ==> ~prime p
```

Thus we have the hypothesis that all p beyond a certain unspecified n are not prime, and our task is to show that this cannot be. At this point we take advantage of Euclid's great inspiration and we build an explicit term from n. In the informal proof we are asked to 'consider' the term FACT $n + 1$.[7] This term will have certain properties (i.e., it has a prime factor) that lead to contradiction. Question: how do we 'consider' this term in the formal HOL proof? Answer: by instantiating a lemma with it and bringing the lemma into the proof. The lemma and its instantiation are:[8]

[7]The HOL parser thinks FACT $n + 1$ is equivalent to (FACT n) $+ 1$.
[8]The function SPEC implements the rule of universal specialization.

```
- PRIME_FACTOR;                                                              68

> val it = |- !n. ~(n = 1) ==> (?p. prime p /\ p divides n) : Thm.thm

- val th = SPEC ``FACT n + 1`` PRIME_FACTOR;

> val th =
>     |- ~(FACT n + 1 = 1) ==> (?p. prime p /\ p divides FACT n + 1)
```

It is evident that the antecedent of `th` can be eliminated. In hol98, one could do this in a so-called *forward* proof style (by proving $\vdash \neg(\texttt{FACT}\ n + 1 = 1)$ and then applying *modus ponens*, the result of which can then be used in the proof), or one could bring `th` into the proof and simplify it *in situ*. We choose the latter approach.

```
- e (MP_TAC (SPEC ``FACT n + 1`` PRIME_FACTOR));                             69

> OK..
> 1 subgoal:
> val it =
>     (~(FACT n + 1 = 1) ==> ?p. prime p /\ p divides FACT n + 1) ==> F
>     ------------------------------------
>       !p. n < p ==> ~prime p
```

The invocation `MP_TAC` ($\vdash M$) applied to a goal (Δ, g) returns the goal $(\Delta, M \supset g)$. Now we simplify:

```
- e (ARW_TAC []);                                                            70

> OK..
> 2 subgoals:
> val it =
>     ~(p divides FACT n + 1)
>     ------------------------------------
>       0.  !p. n < p ==> ~prime p
>       1.  prime p
>
>     ~(FACT n = 0)
>     ------------------------------------
>       !p. n < p ==> ~prime p
```

We recall that zero is less than every factorial, a fact found in `arithmeticTheory` under the name `FACT_LESS`. Thus we can solve the top goal by simplification:

```
- e (ARW_TAC [FACT_LESS, DECIDE `!x. ~(x=0) = 0 < x`]);                     71

> OK..
> Goal proved.  ...
```

Notice the 'on-the-fly' use of `DECIDE` to provide an *ad hoc* rewrite. Looking at the remaining goal, one might think that our aim, to prove falsity, has been lost. But this is not so: a goal $\neg M$ is equivalent to $M \supset$ F. We can quickly proceed to show that p divides (FACT n), and thus that $p = 1$, hence that p is not prime, at which point we are done. This can all be packaged into a single invocation of PROVE_TAC:

```
- e (PROVE_TAC [PRIME_POS, NOT_LESS, DIVIDES_FACT,                         72
                DIVIDES_ADDL, DIVIDES_ONE, NOT_PRIME_1]);
> OK..
> Meson search level: ...........
>
> Goal proved.
>  [..] |- ~(p divides FACT n + 1)
>
> Goal proved.
>  [.]
> |- (~(FACT n + 1 = 1) ==> ?p. prime p /\ p divides FACT n + 1) ==> F
>
> Goal proved.
>  [.] |- F
> val it =
>     Initial goal proved.
>     |- !n. ?p. n < p /\ prime p
```

Euclid's theorem is now proved, and we can rest. However, this presentation of the final proof will be unsatisfactory to some, because the proof is completely hidden in the invocation of the automated reasoner. Well then, let's try another proof, this time employing the so-called 'assertional' style. When used uniformly, this can allow a readable linear presentation that mirrors the informal proof. The following proves Euclid's theorem in the assertional style. We think it is fairly readable, certainly much more so than the standard tactic proof just given.[9]

(*AGAIN*)

```
!n. ?p. n < p /\ prime p
------------------------------------------------------------------------
CCONTR_TAC THEN
`?n. !p. n < p ==> ~prime p`  by PROVE_TAC []              THEN
`~(FACT n + 1 = 1)`           by ARW_TAC [FACT_LESS,
                                      DECIDE`~(x=0)=0<x`]  THEN
`?p. prime p /\
     p divides (FACT n + 1)`  by PROVE_TAC [PRIME_FACTOR]  THEN
`0 < p`                       by PROVE_TAC [PRIME_POS]     THEN
`p <= n`                      by PROVE_TAC [NOT_LESS]      THEN
`p divides FACT n`            by PROVE_TAC [DIVIDES_FACT]  THEN
`p divides 1`                 by PROVE_TAC [DIVIDES_ADDL]  THEN
`p = 1`                       by PROVE_TAC [DIVIDES_ONE]   THEN
`~prime p`                    by PROVE_TAC [NOT_PRIME_1]   THEN
PROVE_TAC []
```

[9]Note that CCONTR_TAC, which is used to start the proof, initiates a proof by contradiction by negating the goal and placing it on the hypotheses, leaving F as the new goal.

4.5 Turning the script into a theory

Having proved our result, we probably want to package it up in a way that makes it available to future sessions, but which doesn't require us to go all through the theorem-proving effort again. Even having a complete script from which it would be possible to cut-and-paste is an error-prone solution.

In order to do this we need to create a file with the name x`Script.sml`, where x is the name of the theory we wish to export. This file then needs to be compiled. In fact, we really do use the Moscow ML compiler, carefully augmented with the appropriate logical context. However, the language accepted by the compiler is not quite the same as that accepted by the interactive system, so we will need to do a little work to massage the original script into the correct form.

We'll give an illustration of converting to a form that can be compiled using the script

```
<holdir>/examples/euclid.sml
```

as our base-line. This file is already close to being in the right form. It has all of the proofs of the theorems in "sewn-up" form so that when run, it does not involve the goal-stack at all. In its given form, it can be run as input to hol.unquote thus:

```
$ cd examples/                                                      1
$ ../bin/hol.unquote < euclid.sml
Moscow ML version 1.44 (August 1999)
Enter ‘quit();’ to quit.
For HOL help, type: help "hol";
  ...

> val EUCLID_AGAIN = |- !n. ?p. n < p /\ prime p : Thm.thm
- Theory: scratch
  ...

Theorems:
    DIVIDES_0 |- !x. x divides 0
    DIVIDES_ZERO |- !x. 0 divides x = x = 0
    DIVIDES_ONE |- !x. x divides 1 = x = 1
  ...
    PRIME_POS |- !p. prime p ==> 0 < p
    PRIME_FACTOR |- !n. ~(n = 1) ==> ?p. prime p /\ p divides n
    EUCLID |- !n. ?p. n < p /\ prime p

Theory "scratch" is inconsistent with disk.
-
```

However, we now want to create a `euclidTheory` that we can load in other interactive sessions, just as we loaded `arithmeticTheory` in the development above. So, our first step is to create a file `euclidScript.sml`, and to copy the body of `euclid.sml` into it. The first (non-comment) line of our new script is

```
load "bossLib";
```

This is necessary in the interactive system, but is redundant when creating compiled code as the compiler detects the dependency on `bossLib` itself (it needs to `open` it in the very next line). This line should be deleted.

The following line opens `arithmeticTheory` and `bossLib`. However, when writing for the compiler, we need to explicitly mention the other HOL modules that we depend on. We must add

```
open HolKernel basicHol90Lib Parse
```

Further, while hol (and hol.unquote) know that `THEN` and `THENL` are infixes, the compiler doesn't, so we must also add

```
infix THEN THENL
```

The next line that poses a difficulty is

```
set_fixity "divides" (Infixr 450);
```

While it is legitimate to type expressions directly into the interactive system, the compiler requires that every top-level phrase be a declaration. We satisfy this requirement by altering this line into a "do nothing" declaration that does not record the result of the expression:

```
val _ = set_fixity "divides" (Infixr 450)
```

The only extra changes are to bracket the rest of the script file text with calls to `new_theory` and `export_theory`. So, before the definition of `divides`, we add:

```
val _ = new_theory "euclid";
```

and at the end of the file:

```
val _ = export_theory();
```

Now, we can compile the script we have created using the Holmake tool. To keep things a little tidier, we first move our script into a new directory.

```
$ mkdir euclid                                                          2
$ mv euclidScript.sml euclid
$ cd euclid
$ ../../bin/Holmake
Analysing euclidScript.sml
Trying to create directory .HOLMK for dependency files
Compiling euclidScript.sml
Linking euclidScript.uo to produce theory building executable
<<HOL message: inventing new type variable names: 'a, 'b.>>
<<HOL message: Created theory "euclid".>>
Definition stored under "divides_def".
Definition stored under "prime_def".
Meson search level: .....
Meson search level: ................
 ...
Theory "euclid" is inconsistent with disk.
Exporting theory ... done.
Analysing euclidTheory.sml
Analysing euclidTheory.sig
Compiling euclidTheory.sig
Compiling euclidTheory.sml
```

Now we have created four new files, various forms of `euclidTheory` with four different suffixes. Only `euclidTheory.sig` is really suitable for human consumption. While still in the `euclid` directory that we created, we can demonstrate:

```
$ ../../bin/hol.unquote                                                 3
[...]

[closing file "/local/scratch/mn200/Work/hol98/tools/end-init.sml"]
- load "euclidTheory";
> val it = () : unit
- open euclidTheory;
> type thm = Thm.thm
  val DIVIDES_TRANS =
    |- !a b c. a divides b / b divides c ==> a divides c
    : Thm.thm
  ...
  val DIVIDES_REFL = |- !x. x divides x : Thm.thm
  val DIVIDES_0 = |- !x. x divides 0 : Thm.thm
```

4.6 Summary

The reader has now seen an interesting theorem proved, in great detail, in hol98. The discussion illustrated the high-level tools provided in `bossLib` and touched on issues including tool selection, undo, 'tactic polishing', exploratory simplification, and the 'forking-off' of new proof attempts. We also attempted to give a flavour of the thought processes a user would employ. Following is a more-or-less random collection of other observations.

- Even though the proof of Euclid's theorem is short and easy to understand when presented informally, a perhaps surprising amount of support development was required to set the stage for Euclid's classic argument.

- The proof support offered by `bossLib` (`RW_TAC`, `PROVE_TAC`, `DECIDE_TAC`, `DECIDE`, `Cases_on`, `Induct_on`, and the "`by`" construct) was nearly complete for this example: it was rarely necessary to resort to lower-level tactics.

- Simplification is a workhorse tactic; even when an automated reasoner like `PROVE_TAC` is used, its application has often been set up by some exploratory simplifications. It therefore pays to become familiar with the strengths and weaknesses of the simplifier.

- A common problem with interactive proof systems is dealing with hypotheses. Often `PROVE_TAC` and the "`by`" construct allow the use of hypotheses without directly resorting to indexing into them (or naming them, which amounts to the same thing). This is desirable, since the hypotheses are notionally a *set*, and moreover, experience has shown that profligate indexing into hypotheses results in hard-to-maintain proof scripts. However, it can be clumsy to work with a large set of hypotheses, in which case the following approaches may be useful.

  One can directly refer to hypotheses by using `UNDISCH_TAC` (makes the designated hypothesis the antecedent to the goal), `ASSUM_LIST` (gives the entire hypothesis list to a tactic), `POP_ASSUM` (gives the top hypothesis to a tactic), and `PAT_ASSUM` (gives the first *matching* hypothesis to a tactic). (See the *REFERENCE* for further details on all of these.) The numbers attached to hypotheses by the proof manager could likely be used to access hypotheses (it would be quite simple to write such a tactic). However, starting a new proof is sometimes the most clarifying thing to do.

  In some cases, it is useful to be able to delete a hypothesis. This can be accomplished by passing the hypothesis to a tactic that ignores it. For example, to discard the top hypothesis, one could invoke `POP_ASSUM (K ALL_TAC)`.

- In the example, we didn't use the more advanced features of `bossLib`, largely because they do not, as yet, provide much more functionality than the simple sequencing of simplification, decision procedures, and automated first order reasoning. The `THEN` tactical has thus served as an adequate replacement. In the future, these entrypoints should become more powerful.

- It is almost always necessary to have an idea of the *informal* proof in order to be successful when doing a formal proof. However, all too often the following strategy is adopted by novices: (1) rewrite the goal with a few relevant definitions, and then (2) rely on the syntax of the resulting goal to guide subsequent tactic selection. Such an approach constitutes a clear case of the tail wagging the dog, and is a poor strategy to adopt. Insight into the high-level structure of the proof is one of the most important factors in successful verification exercises.

The author has noticed that many of the most successful verification experts work using a sheet of paper to keep track of the main steps that need to be made. Perhaps looking away to the paper helps break the mesmerizing effect of the computer screen.

On the other hand, one of the advantages of having a mechanized logic is that the machine can be used as a formal expression calculator, and thus the user can use it to quickly and accurately explore various proof possibilities.

- High powered tools like `PROVE_TAC`, `DECIDE_TAC`, and `RW_TAC` are the principal way of advancing a proof in `bossLib`. In many cases, they do exactly what is desired, or even manage to surprise the user with their power. In the formalization of Euclid's theorem, the tools performed fairly well. However, sometimes they are overly aggressive, or they simply flounder. In such cases, more specialized proof tools need to be used, or even written, and hence the support underlying `bossLib` must eventually be learned.

- Having a good knowledge of the available lemmas, and where they are located, is an essential part of being successful. Often powerful tools can replace lemmas in a restricted domain, but in general, one has to know what has already been proved. We have found that the entrypoints in `DB` help in quickly finding lemmas.

Chapter 5

Introduction to Proof with HOL

Preliminaries This chapter discusses the nature of proof in HOL in more detail. The previous chapter has provided a broad overview of how proof is done in HOL, while here the emphasis is on attaining a more thorough grounding in the material. As before, we are using hol.unquote and we require the following command to be entered as a first step:

```
- app load ["arithmeticTheory", "pairTheory"];                    0
> val it = () : unit
```

For a logician, a formal proof is a sequence, each of whose elements is either an *axiom* or follows from earlier members of the sequence by a *rule of inference*. A theorem is the last element of a proof.

Theorems are represented in HOL by values of an abstract type `thm`. The only way to create theorems is by generating a proof. In HOL (following LCF), this consists in applying ML functions representing *rules of inference* to axioms or previously generated theorems. The sequence of such applications directly corresponds to a logician's proof.

There are five axioms of the HOL logic and eight primitive inference rules. The axioms are bound to ML names. For example, the Law of Excluded Middle is bound to the ML name `BOOL_CASES_AX`:

```
- BOOL_CASES_AX;                                                  1
> val it = |- !t. (t = T) \/ (t = F) : thm
```

Theorems are printed with a preceding turnstile `|-` as illustrated above; the symbol '`!`' is the universal quantifier '$\forall$'. Rules of inference are ML functions that return values of type `thm`. An example of a rule of inference is *specialization* (or $\forall$-elimination). In standard 'natural deduction' notation this is:

$$\frac{\Gamma \vdash \forall x.\, t}{\Gamma \vdash t[t'/x]}$$

- $t[t'/x]$ denotes the result of substituting t' for free occurrences of x in t, with the restriction that no free variables in t' become bound after substitution.

This rule is represented in ML by a function `SPEC`,[1] which takes as arguments a term ``` ``$a$`` ``` and a theorem `|-` $!x.t[x]$ and returns the theorem `|-` $t[a]$, the result of substituting a for x in $t[x]$.

```
- val Th1 = BOOL_CASES_AX;                                              2
> val Th1 = |- !t. (t = T) \/ (t = F) : thm

- val Th2 = SPEC ``1 = 2`` Th1;
> val Th2 = |- ((1 = 2) = T) \/ ((1 = 2) = F) : thm
```

This session consists of a proof of two steps: using an axiom and applying the rule `SPEC`; it interactively performs the following proof:

1. $\vdash\ \forall t.\, t = \top\ \ \vee\ \ t = \bot$ [Axiom `BOOL_CASES_AX`]
2. $\vdash\ (1{=}2) = \top\ \ \vee\ \ (1{=}2) = \bot$ [Specializing line 1 to '1=2']

If the argument to an ML function representing a rule of inference is of the wrong kind, or violates a condition of the rule, then the application fails. For example, `SPEC` $t\ th$ will fail if th is not of the form `|-` $!x.\cdots$ or if it is of this form but the type of t is not the same as the type of x, or if the free variable restriction is not met. When one of the standard `HOL_ERR` exceptions is raised, more information about the failure can often be gained by using the `Raise` function.

```
- SPEC ``1=2`` Th2;                                                     3
! Uncaught exception:
! HOL_ERR <poly>

- SPEC ``1 = 2`` Th2 handle e => Raise e;

Exception raised at Thm.SPEC:

! Uncaught exception:
! HOL_ERR <poly>
```

However, as this session illustrates, the failure token does not always indicate the exact reason for failure. The failure conditions for rules of inference are given in *REFERENCE*.

A proof in the HOL system is constructed by repeatedly applying inference rules to axioms or to previously proved theorems. Since proofs may consist of millions of steps, it is necessary to provide tools to make proof construction easier for the user. The proof generating tools in the HOL system are just those of LCF, and are described later.

The general form of a theorem is $t_1, \ldots, t_n$ `|-` t, where t_1, ... , t_n are boolean terms called the *assumptions* and t is a boolean term called the *conclusion*. Such a theorem

[1]`SPEC` is not a primitive rule of inference in the HOL logic, but is a derived rule. Derived rules are described in Section 5.1.

asserts that if its assumptions are true then so is its conclusion. Its truth conditions are thus the same as those for the single term (t_1/\.../\t_n)==>t. Theorems with no assumptions are printed out in the form |- t.

The five axioms and eight primitive inference rules of the HOL logic are described in detail in the document *DESCRIPTION*. Every value of type thm in the HOL system can be obtained by repeatedly applying primitive inference rules to axioms. When the HOL system is built, the eight primitive rules of inference are defined and the five axioms are bound to their ML names, all other predefined theorems are proved using rules of inference as the system is made.[2] This is one of the reasons why building hol takes so long.

In the rest of this chapter, the process of *forward proof*, which has just been sketched, is described in more detail. In Chapter 6 *goal directed proof* is described, including the important notions of *tactics* and *tacticals*, due to Robin Milner.

5.1 Forward proof

Three of the primitive inference rules of the HOL logic are ASSUME (assumption introduction), DISCH (discharging or assumption elimination) and MP (Modus Ponens). These rules will be used to illustrate forward proof and the writing of derived rules.

The inference rule ASSUME generates theorems of the form t |- t. Note, however, that the ML printer prints each assumption as a dot (but this default can be changed; see below). The function dest_thm decomposes a theorem into a pair consisting of list of assumptions and the conclusion. The ML type goal abbreviates (term)list # term, this is motivated in Section 6.

```
- val Th3 = ASSUME ``t1==>t2``;;                                    4
> val Th3 =  [.] |- t1 ==> t2 : thm

- dest_thm Th3;
> val it = ([``t1 ==> t2``], ``t1 ==> t2``) : term list * term
```

A sort of dual to ASSUME is the primitive inference rule DISCH (discharging, assumption elimination) which infers from a theorem of the form $\cdots t_1 \cdots$ |- t_2 the new theorem $\cdots\ \cdots$ |- t_1==>t_2. DISCH takes as arguments the term to be discharged (i.e. t_1) and the theorem from whose assumptions it is to be discharged and returns the result of the discharging. The following session illustrates this:

```
- val Th4 = DISCH ``t1==>t2`` Th3;                                  5
> val Th4 = |- (t1 ==> t2) ==> t1 ==> t2 : thm
```

[2]This is a slight over-simplification.

Note that the term being discharged need not be in the assumptions; in this case they will be unchanged.

```
- DISCH ``1=2`` Th3;                                                        6
> val it =  [.] |- (1 = 2) ==> t1 ==> t2 : thm

- dest_thm it;
> val it = ([``t1 ==> t2``], ``(1 = 2) ==> t1 ==> t2``) : term list * term
```

In HOL the rule MP of Modus Ponens is specified in conventional notation by:

$$\frac{\Gamma_1 \;\vdash\; t_1 \Rightarrow t_2 \qquad\qquad \Gamma_2 \;\vdash\; t_1}{\Gamma_1 \cup \Gamma_2 \;\vdash\; t_2}$$

The ML function MP takes argument theorems of the form $\cdots$ `|-` t_1 `==>` t_2 and $\cdots$ `|-` t_1 and returns $\cdots$ `|-` t_2. The next session illustrates the use of MP and also a common error, namely not supplying the HOL logic type checker with enough information.

```
- val Th5 = ASSUME ``t1``;                                                  7
<<HOL message: inventing new type variable names: 'a.>>
! Uncaught exception:
! HOL_ERR <poly>
- val Th5 = ASSUME ``t1`` handle e => Raise e;
<<HOL message: inventing new type variable names: 'a.>>

Exception raised at Thm.ASSUME:
not a proposition
! Uncaught exception:
! HOL_ERR <poly>

- val Th5 = ASSUME ``t1:bool``;
> val Th5 =  [.] |- t1 : thm

- val Th6 = MP Th3 Th5;
> val Th6 =  [..] |- t2 : thm
```

The hypotheses of `Th6` can be inspected with the ML function `hyp`, which returns the list of assumptions of a theorem (the conclusion is returned by `concl`).

```
- hyp Th6;                                                                  8
> val it = [``t1 ==> t2``, ``t1``] : term list
```

HOL can be made to print out hypotheses of theorems explicitly by setting the global flag `show_assums` to true.

```
- show_assums := true;                                          9
> val it = () : unit

- Th5;
> val it =  [t1] |- t1 : thm

- Th6;
> val it =  [t1 ==> t2, t1] |- t2 : thm
```

Discharging `Th6` twice establishes the theorem `|- t1 ==> (t1==>t2) ==> t2`.

```
- val Th7 = DISCH ``t1==>t2`` Th6;                              10
> val Th7 = [t1] |- (t1 ==> t2) ==> t2 : thm

- val Th8 = DISCH ``t1:bool`` Th7;
> val Th8 = |- t1 ==> (t1 ==> t2) ==> t2 : thm
```

The sequence of theorems: `Th3`, `Th5`, `Th6`, `Th7`, `Th8` constitutes a proof in HOL of the theorem `|- t1 ==> (t1 ==> t2) ==> t2`. In standard logical notation this proof could be written:

1. $t_1 \Rightarrow t_2 \ \vdash \ t_1 \Rightarrow t_2$ [Assumption introduction]
2. $t_1 \ \vdash \ t_1$ [Assumption introduction]
3. $t_1 \Rightarrow t_2,\ t_1 \ \vdash \ t_2$ [Modus Ponens applied to lines 1 and 2]
4. $t_1 \ \vdash \ (t_1 \Rightarrow t_2) \Rightarrow t_2$ [Discharging the first assumption of line 3]
5. $\vdash \ t_1 \Rightarrow (t_1 \Rightarrow t_2) \Rightarrow t_2$ [Discharging the only assumption of line 4]

5.1.1 Derived rules

A *proof from hypothesis* $th_1, \ldots, th_n$ is a sequence each of whose elements is either an axiom, or one of the hypotheses th_i, or follows from earlier elements by a rule of inference.

For example, a proof of $\Gamma,\ t' \ \vdash \ t$ from the hypothesis $\Gamma \ \vdash \ t$ is:

1. $t' \ \vdash \ t'$ [Assumption introduction]
2. $\Gamma \ \vdash \ t$ [Hypothesis]
3. $\Gamma \ \vdash \ t' \Rightarrow t$ [Discharge t' from line 2]
4. $\Gamma,\ t' \ \vdash \ t$ [Modus Ponens applied to lines 3 and 1]

This proof works for any hypothesis of the form $\Gamma \vdash t$ and any boolean term t' and shows that the result of adding an arbitrary hypothesis to a theorem is another theorem (because the four lines above can be added to any proof of $\Gamma \vdash t$ to get a proof of $\Gamma,\ t' \vdash t$).[3] For example, the next session uses this proof to add the hypothesis ``t3`` to Th6.

```
- val Th9 = ASSUME ``t3:bool``;                                    11
> val Th9 = [t3] |- t3 : thm

- val Th10 = DISCH ``t3:bool`` Th6;
> val Th10 = [t1 ==> t2, t1] |- t3 ==> t2 : thm

- val Th11 = MP Th10 Th9;
> val Th11 = [t1 ==> t2, t1, t3] |- t2 : thm
```

A *derived rule* is an ML procedure that generates a proof from given hypotheses each time it is invoked. The hypotheses are the arguments of the rule. To illustrate this, a rule, called ADD_ASSUM, will now be defined as an ML procedure that carries out the proof above. In standard notation this would be described by:

$$\frac{\Gamma \vdash t}{\Gamma,\ t' \vdash t}$$

The ML definition is:

```
- fun ADD_ASSUM t th = let                                         12
    val th9 = ASSUME t
    val th10 = DISCH t th
  in
    MP th10 th9
  end;
> val ADD_ASSUM = fn : term -> thm -> thm

- ADD_ASSUM ``t3:bool`` Th6;
> val it =  [t1, t1 ==> t2, t3] |- t2 : thm
```

The body of ADD_ASSUM has been coded to mirror the proof done in session 10 above, so as to show how an interactive proof can be generalized into a procedure. But ADD_ASSUM can be written much more concisely as:

```
- fun ADD_ASSUM t th = MP (DISCH t th) (ASSUME t);                 13
> val ADD_ASSUM = fn : term -> thm -> thm

- ADD_ASSUM t3 Th6;
val it = [t1 ==> t2, t1, t3] |- t2 : thm
```

[3]This property of the logic is called *monotonicity*.

Another example of a derived inference rule is UNDISCH; this moves the antecedent of an implication to the assumptions.

$$\frac{\Gamma \ \vdash \ t_1 \Rightarrow t_2}{\Gamma,\ t_1 \ \vdash \ t_2}$$

An ML derived rule that implements this is:

```
- fun UNDISCH th = MP th (ASSUME(#1(dest_imp(concl th))));            14
> val UNDISCH = fn : thm -> thm

- Th10;
> val it =  [t1 ==> t2, t1] |- t3 ==> t2 : thm

- UNDISCH Th10;
> val it =  [t1, t1 ==> t2, t3] |- t2 : thm
```

Each time UNDISCH $\Gamma \ \vdash \ t_1 \Rightarrow t_2$ is executed, the following proof is performed:

1. $t_1 \ \vdash \ t_1$ [Assumption introduction]
2. $\Gamma \ \vdash \ t_1 \Rightarrow t_2$ [Hypothesis]
3. $\Gamma,\ t_1 \ \vdash \ t_2$ [Modus Ponens applied to lines 2 and 1]

The rules ADD_ASSUM and UNDISCH are the first derived rules defined when the HOL system is built. For a description of the main rules see the section on derived rules in *DESCRIPTION*.

5.2 Rewriting

An important derived rule is REWRITE_RULE. This takes a list of conjunctions of equations, i.e. a list of theorems of the form:

$$\Gamma \ \vdash \ (u_1 = v_1) \ \wedge \ (u_2 = v_2) \ \wedge \ \ldots \ \wedge \ (u_n = v_n)$$

and a theorem $\Delta \ \vdash \ t$ and repeatedly replaces instances of u_i in t by the corresponding instance of v_i until no further change occurs. The result is a theorem $\Gamma \cup \Delta \ \vdash \ t'$ where t' is the result of rewriting t in this way. The session below illustrates the use of REWRITE_RULE. In it the list of equations is a list rewrite_list containing the pre-proved theorems ADD_CLAUSES and MULT_CLAUSES. These theorems *autoload* from the theory arithmetic, so we must use a fully qualified name with the name of the theory as the first component to refer to them. (Alternatively, we could, as in the Euclid example of chapter 4, use open to bring declare all of the values in the theory at the top level.)

```
- val rewrite_list = [arithmeticTheory.ADD_CLAUSES,                    15
                      arithmeticTheory.MULT_CLAUSES];
> val rewrite_list =
    [ []
     |- (0 + m = m) /\ (m + 0 = m) /\ (SUC m + n = SUC (m + n)) /\
        (m + SUC n = SUC (m + n)),
      []
     |- !m n.
          (0 * m = 0) /\ (m * 0 = 0) /\ (1 * m = m) /\ (m * 1 = m) /\
          (SUC m * n = m * n + n) /\ (m * SUC n = m + m * n)]
    : Thm.thm list
```

```
- REWRITE_RULE rewrite_list (ASSUME ``(m+0)<(1*n)+(SUC 0)``);          16
> val it =  [m + 0 < 1 * n + SUC 0] |- m < SUC n : thm
```

This can then be rewritten using another pre-proved theorem LESS_THM, this one from the theory prim_rec:

```
- REWRITE_RULE [prim_recTheory.LESS_THM] it;                          17
> val it =  [m + 0 < 1 * n + SUC 0] |- (m = n) \/ m < n : thm
```

REWRITE_RULE is not a primitive in HOL, but is a derived rule. It is inherited from Cambridge LCF and was implemented by Larry Paulson (see his paper [9] for details). In addition to the supplied equations, REWRITE_RULE has some built in standard simplifications:

```
- REWRITE_RULE [] (ASSUME ``(T /\ x) \/ F ==> F``);                    18
> val it = [T /\ x \/ F ==> F] |- ~x : thm
```

There are elaborate facilities in HOL for producing customized rewriting tools which scan through terms in user programmed orders; REWRITE_RULE is the tip of an iceberg, see *DESCRIPTION* for more details.

Chapter 6

Goal Oriented Proof: Tactics and Tacticals

The style of forward proof described in the previous chapter is unnatural and too ‘low level’ for many applications. An important advance in proof generating methodology was made by Robin Milner in the early 1970s when he invented the notion of *tactics*. A tactic is a function that does two things.

(i) Splits a ‘goal’ into ‘subgoals’.

(ii) Keeps track of the reason why solving the subgoals will solve the goal.

Consider, for example, the rule of $\wedge$-introduction[1] shown below:

$$\frac{\Gamma_1 \ \vdash \ t_1 \qquad\qquad \Gamma_2 \ \vdash \ t_2}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t_1 \ \wedge \ t_2}$$

In HOL, $\wedge$-introduction is represented by the ML function `CONJ`:

$$\texttt{CONJ}\ (\Gamma_1 \ \vdash \ t_1)\ (\Gamma_2 \ \vdash \ t_2) \ \ \rightarrow \ \ (\Gamma_1 \cup \Gamma_2 \ \vdash \ t_1 \ \wedge \ t_2)$$

This is illustrated in the following new session (note that the session number has been reset to *1*, but we’ll assume that the same setup (from the previous chapter’s session 0, has been invoked):

```
- show_assums := true;                                              1
val it = () : unit

- val Th1 = ASSUME ``A:bool`` and Th2 = ASSUME ``B:bool``;
> val Th1 =  [A] |- A : thm
  val Th2 =  [B] |- B : thm

- val Th3 = CONJ Th1 Th2;
> val Th3 =  [A, B] |- A /\ B : thm
```

Suppose the goal is to prove $A \ \wedge \ B$, then this rule says that it is sufficient to prove the two subgoals A and B, because from $\vdash \ A$ and $\vdash \ B$ the theorem $\vdash \ A \ \wedge \ B$ can be deduced. Thus:

[1]In higher order logic this is a derived rule; in first order logic it is usually primitive. In HOL the rule is called `CONJ` and its derivation is given in *DESCRIPTION*.

(i) To prove $\vdash A \wedge B$ it is sufficient to prove $\vdash A$ and $\vdash B$.

(ii) The justification for the reduction of the goal $\vdash A \wedge B$ to the two subgoals $\vdash A$ and $\vdash B$ is the rule of $\wedge$-introduction.

A *goal* in HOL is a pair `(`[t_1;...;t_n]`,`t`)` of ML type `term list * term`. An *achievement* of such a goal is a theorem t_1,...,t_n `|-` t. A tactic is an ML function that when applied to a goal generates subgoals together with a *justification function* or *validation*, which will be an ML derived inference rule, that can be used to infer an achievement of the original goal from achievements of the subgoals.

If T is a tactic (i.e. an ML function of type `goal -> (goal list * (thm list -> thm))`) and g is a goal, then applying T to g (i.e. evaluating the ML expression $T\ g$) will result in an object which is a pair whose first component is a list of goals and whose second component is a justification function, i.e. a value with ML type `thm list -> thm`.

An example tactic is `CONJ_TAC` which implements (i) and (ii) above. For example, consider the utterly trivial goal of showing `T /\ T`, where `T` is a constant that stands for $true$:

```
- val goal1 =([]:term list, ``T /\ T``);                            2
> val goal1 = ([], ``T /\ T``) : term list * term

- CONJ_TAC goal1;
> val it =
    ([([], ``T``), ([], ``T``)], fn)
    : (term list * term) list * (thm list -> thm)

- val (goal_list,just_fn) = it;
> val goal_list =
    [([], ``T``), ([], ``T``)]
    : (term list * term) list
  val just_fn = fn : thm list -> thm
```

`CONJ_TAC` has produced a goal list consisting of two identical subgoals of just showing `([],"T")`. Now, there is a preproved theorem in HOL, called `TRUTH`, that achieves this goal:

```
- TRUTH;                                                            3
> val it = [] |- T : thm
```

Applying the justification function `just_fn` to a list of theorems achieving the goals in `goal_list` results in a theorem achieving the original goal:

```
- just_fn [TRUTH,TRUTH];                                            4
> val it =  [] |- T /\ T : thm
```

Although this example is trivial, it does illustrate the essential idea of tactics. Note that tactics are not special theorem-proving primitives; they are just ML functions. For example, the definition of `CONJ_TAC` is simply:

```
fun CONJ_TAC (asl,w) = let
  val (l,r) = dest_conj w
in
  ([(asl,l), (asl,r)], fn [th1,th2] => CONJ th1 th2)
end
```

The ML function `dest_conj` splits a conjunction into its two conjuncts: If `(asl,``$t_1$`/\`$t_2$``)` is a goal, then `CONJ_TAC` splits it into the list of two subgoals `(asl,`$t_1$`)` and `(asl,`$t_2$`)`. The justification function, `fn [th1,th2] => CONJ th1 th2` takes a list `[`$th_1$`,`$th_2$`]` of theorems and applies the rule `CONJ` to th_1 and th_2.

To summarize: if T is a tactic and g is a goal, then applying T to g will result in a pair whose first component is a list of goals and whose second component is a justification function, with ML type `thm list -> thm`.

Suppose T g `= ([`g_1`,...,`g_n`],`p`)`. The idea is that g_1 , ... , g_n are subgoals and p is a 'justification' of the reduction of goal g to subgoals g_1 , ... , g_n. Suppose further that the subgoals g_1 , ... , g_n have been solved. This would mean that theorems th_1 , ... , th_n had been proved such that each th_i ($1 \leq i \leq n$) 'achieves' the goal g_i. The justification p (produced by applying T to g) is an ML function which when applied to the list `[`th_1`,...,`th_n`]` returns a theorem, th, which 'achieves' the original goal g. Thus p is a function for converting a solution of the subgoals to a solution of the original goal. If p does this successfully, then the tactic T is called *valid*. Invalid tactics cannot result in the proof of invalid theorems; the worst they can do is result in insolvable goals or unintended theorems being proved. If T were invalid and were used to reduce goal g to subgoals g_1 , ... , g_n, then effort might be spent proving theorems th_1 , ... , th_n to achieve the subgoals g_1 , ... , g_n, only to find out after the work is done that this is a blind alley because p`[`th_1`,...,`th_n`]` doesn't achieve g (i.e. it fails, or else it achieves some other goal).

A theorem *achieves* a goal if the assumptions of the theorem are included in the assumptions of the goal *and* if the conclusion of the theorems is equal (up to the renaming of bound variables) to the conclusion of the goal. More precisely, a theorem

$$t_1, \ldots, t_m \ \texttt{|-}\ t$$

achieves a goal

$$\texttt{([}u_1\texttt{,}\ldots\texttt{,}u_n\texttt{],}u\texttt{)}$$

if and only if $\{t_1, \ldots, t_m\}$ is a subset of $\{u_1, \ldots, u_n\}$ and t is equal to u (up to renaming of bound variables). For example, the goal `([``x=y``, ``y=z``, ``z=w``], ``x=z``)` is achieved by the theorem `[x=y, y=z] |- x=z` (the assumption ``` ``z=w`` ``` is not needed).

A tactic *solves* a goal if it reduces the goal to the empty list of subgoals. Thus T solves g if $T\ g$ `= ([],`p`)`. If this is the case and if T is valid, then p`[]` will evaluate to a theorem achieving g. Thus if T solves g then the ML expression `snd(`$T\ g$`) []` evaluates to a theorem achieving g.

Tactics are specified using the following notation:

$$\frac{goal}{\overline{goal_1 \quad goal_2 \quad \cdots \quad goal_n}}$$

For example, a tactic called `CONJ_TAC` is described by

$$\frac{t_1 \texttt{/\textbackslash} t_2}{\overline{t_1 \qquad t_2}}$$

Thus `CONJ_TAC` reduces a goal of the form `(`Γ`,``t_1`/\`t_2```)` to subgoals `(`$\Gamma$`,``t_1```)` and `(`$\Gamma$`,``t_2```)`. The fact that the assumptions of the top-level goal are propagated unchanged to the two subgoals is indicated by the absence of assumptions in the notation.

Another example is `numLib.INDUCT_TAC`, the tactic for doing mathematical induction on the natural numbers:

$$\frac{\texttt{!}n\texttt{.}t[n]}{\overline{t[0] \qquad \{t[n]\}\ t[\texttt{SUC}\ n]}}$$

`INDUCT_TAC` reduces a goal `(`Γ`,``!`$n$`.`$t[n]$```)` to a basis subgoal `(`$\Gamma$`,``$t[0]$```)` and an induction step subgoal `(`$\Gamma \cup \{$```$t[n]$```$\}$`,``$t[$`SUC` $n]$```)`. The extra induction assumption ```$t[n]$``` is indicated in the tactic notation with set brackets.

```
- numLib.INDUCT_TAC([], ``!m n. m+n = n+m``);                                    5
> val it =
    ([([], ``!n. 0 + n = n + 0``),
      ([``!n. m + n = n + m``], ``!n. SUC m + n = n + SUC m``)], fn)
    : (term list * term) list * (thm list -> thm)
```

The first subgoal is the basis case and the second subgoal is the step case.

Tactics generally fail (in the ML sense, i.e. raise an exception) if they are applied to inappropriate goals. For example, `CONJ_TAC` will fail if it is applied to a goal whose conclusion is not a conjunction. Some tactics never fail, for example `ALL_TAC`

$$\frac{t}{\overline{t}}$$

is the 'identity tactic'; it reduces a goal `(`Γ`,`t`)` to the single subgoal `(`Γ`,`t`)`—i.e. it has no effect. `ALL_TAC` is useful for writing complex tactics using tacticals.

6.1 Using tactics to prove theorems

Suppose goal g is to be solved. If g is simple it might be possible to immediately think up a tactic, T say, which reduces it to the empty list of subgoals. If this is the case then executing:

`val (`gl`,`p`) = `T g

will bind p to a function which when applied to the empty list of theorems yields a theorem th achieving g. (The declaration above will also bind gl to the empty list of goals.) Thus a theorem achieving g can be computed by executing:

`val `th` = `p`[]`

This will be illustrated using `REWRITE_TAC` which takes a list of equations (empty in the example that follows) and tries to prove a goal by rewriting with these equations together with `basic_rewrites`:

```
- val goal2 = ([]:term list, ``T /\ x ==> x \/ (y /\ F)``);                    6
> val goal2 = ([], ``T /\ x ==> x \/ y /\ F``) : (term list # term)

- REWRITE_TAC [] goal2;
> val it = ([], -) : (term list * term) list * (thm list -> thm)

- #2 it [];
> val it =  [] |- T /\ x ==> x \/ y /\ F : thm
```

Proved theorems are usually stored in the current theory so that they can be used in subsequent sessions.

The built-in function `store_thm` of ML type `(string * term * tactic) -> thm` facilitates the use of tactics: `store_thm("foo",`t`,`T`)` proves the goal `([],`t`)` (i.e. the goal with no assumptions and conclusion t) using tactic T and saves the resulting theorem with name `foo` on the current theory.

If the theorem is not to be saved, the function `prove` of type `(term * tactic) -> thm` can be used. Evaluating `prove(`t`,`T`)` proves the goal `([],`t`)` using T and returns the result without saving it. In both cases the evaluation fails if T does not solve the goal `([],`t`)`.

When conducting a proof that involves many subgoals and tactics, it is necessary to keep track of all the justification functions and compose them in the correct order. While this is feasible even in large proofs, it is tedious. HOL provides a package for building and traversing the tree of subgoals, stacking the justification functions and applying them properly; this package was originally implemented for LCF by Larry Paulson.

The subgoal package implements a simple framework for interactive proof. A proof tree is created and traversed top-down. The current goal can be expanded into subgoals using a tactic; the subgoals are pushed onto a goal stack and the justification function onto a proof stack. Subgoals can be considered in any order. If the tactic solves a subgoal (i.e. returns an empty subgoal list), then the package proceeds to the next subgoal in the tree.

The function `set_goal` of type `goal -> void` initializes the subgoal package with a new goal. Usually top-level goals have no assumptions; the function `g` is useful in this case.

To illustrate the subgoal package the trivial theorem $\vdash \forall m.\ m + 0 = m$ will be proved from the definition of addition (we first `open` the theory of arithmetic and `numLib`, to bring the theorems and `INDUCT_TAC` to the top level):

```
- open arithmeticTheory numLib;                                          7
> ...
- ADD;
> val it = |- (!n. 0 + n = n) /\ (!m n. (SUC m) + n = SUC(m + n)) : thm
```

Notice that `ADD` specifies $0 + m = m$ but not $m + 0 = m$. Of course, $\forall m\ n.\ m + n = n + m$ is true, but the first step of the proof is to show $\forall m.\ m + 0 = m$ from the definition of addition. Notice that the function `g` does not take a term as an argument, but rather a *quotation*, with only one set of back-quotes.

```
- g ‘!m. m+0=m‘;                                                         1
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !m. m + 0 = m
```

This sets up the goal. Next the goal is split into a basis and step case with `INDUCT_TAC`. To do this the function `e` (or, equivalently, `expand`) is used. This applies a tactic to the top goal on the stack, then pushes the resulting subgoals onto the goal stack, then prints the resulting subgoals. If there are no subgoals, the justification function is applied to the theorems solving the subgoals that have been proved and the resulting theorems are printed.

```
- e INDUCT_TAC;;                                                         2
OK..
2 subgoals:
> val it =
    SUC m + 0 = SUC m
    ------------------------------------
      m + 0 = m

    0 + 0 = 0
```

The top of the goal stack is printed last. The basis case is an instance of the definition of addition, so is solved by rewriting with `ADD`.

```
- e(REWRITE_TAC[ADD]);                                                   3
OK..

Goal proved.
 [] |- 0 + 0 = 0

Remaining subgoals:
> val it =
    SUC m + 0 = SUC m
    ------------------------------------
      m + 0 = m
```

The basis is solved and the goal stack popped so that its top is now the step case, namely showing that `(SUC m) + 0 = SUC m` under the assumption `m + 0 = m`. This goal can be solved by rewriting first with the definition of addition:

```
- e(REWRITE_TAC[ADD]);                                                   4
OK..
1 subgoal:
> val it =
    SUC (m + 0) = SUC m
    ------------------------------------
      m + 0 = m
```

and then with the assumption `m+0=m`. The tactic `ASM_REWRITE_TAC` is used to rewrite with the assumptions of a goal. It is just like `REWRITE_TAC` except that it adds the assumptions to the list of equations used for rewriting. For the example here no equations besides the assumptions are needed, so `ASM_REWRITE_TAC` is given the empty list of equations.

```
- e(ASM_REWRITE_TAC[]);                                                  5
OK..

Goal proved.
 [m + 0 = m] |- SUC (m + 0) = SUC m

Goal proved.
 [m + 0 = m] |- SUC m + 0 = SUC m
> val it =
    Initial goal proved.
     [] |- !m. m + 0 = m
    : GoalstackPure.goalstack
```

The top goal is solved, hence the preceding goal (the step case) is solved too, and since the basis is already solved, the main goal is solved.

The theorem achieving the goal can be extracted from the subgoal package with `top_thm`:

```
- top_thm();                                                    6
val it = [] |- !m. m + 0 = m : thm
```

The proof just done can be 'optimized'. For example, instead of first rewriting with ADD (box 4) and then with the assumptions (box 5), a single rewriting with ADD and the assumptions would suffice. To illustrate, the last two steps of the proof will be 'undone' using the function backup (also, b) which restores the previous state of the goal and theorem stacks.

```
- b();                                                          7
> val it =
    SUC (m + 0) = SUC m
    ------------------------------------
      m + 0 = m

- b();
> val it =
    SUC m + 0 = SUC m
    ------------------------------------
      m + 0 = m
```

The proof can now be completed in one step instead of two:

```
- e(ASM_REWRITE_TAC[ADD]);                                      8
OK..

Goal proved.
 [m + 0 = m] |- SUC m + 0 = SUC m
> val it =
    Initial goal proved.
     [] |- !m. m + 0 = m
    : GoalstackPure.goalstack
```

The order in which goals are attacked can be adjusted using rotate n (alternatively, r) which rotates the goal stack by n. For example:

```
- b(); b();                                                          9
> ...

> val it =
    SUC m + 0 = SUC m
    ------------------------------------
      m + 0 = m

    0 + 0 = 0

- r 1;
> val it =
    0 + 0 = 0


    SUC m + 0 = SUC m
    ------------------------------------
      m + 0 = m
```

The top goal is now the step case not the basis case, so expanding with a tactic will apply the tactic to the step case.

```
- e(ASM_REWRITE_TAC[ADD]);                                           10
OK..

Goal proved.
 [m + 0 = m] |- SUC m + 0 = SUC m

Remaining subgoals:
> val it =
    0 + 0 = 0
```

It is possible to do the whole proof in one step, but this requires a compound tactic built using the *tactical*[2] `THENL`. Tacticals are higher order operations for combining tactics.

6.2 Tacticals

A *tactical* is an ML function that returns a tactic (or tactics) as result. Tacticals may take various parameters; this is reflected in the various ML types that the built-in tacticals have. Some important tacticals in the HOL system are listed below.

[2]This word was invented by Robin Milner: ‘tactical’ is to ‘tactic‘ as ‘functional’ is to ‘function’.

6.2.1 THENL : tactic -> tactic list -> tactic

If tactic T produces n subgoals and $T_1, \ldots, T_n$ are tactics then T THENL [T_1;...;T_n] is a tactic which first applies T and then applies T_i to the ith subgoal produced by T. The tactical THENL is useful if one wants to do different things to different subgoals.

THENL can be illustrated by doing the proof of $\vdash \forall m.\ m + 0 = m$ in one step.

```
- g `!m. m + 0 = m`;                                                    1
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !m. m + 0 = m

- e(INDUCT_TAC THENL [REWRITE_TAC[ADD], ASM_REWRITE_TAC[ADD]]);
OK..
> val it =
    Initial goal proved.
     [] |- !m. m + 0 = m
```

The compound tactic INDUCT_TAC THENL [REWRITE_TAC[ADD];ASM_REWRITE_TAC[ADD]] first applies INDUCT_TAC and then applies REWRITE_TAC[ADD] to the first subgoal (the basis) and ASM_REWRITE_TAC[ADD] to the second subgoal (the step).

The tactical THENL is useful for doing different things to different subgoals. The tactical THEN can be used to apply the same tactic to all subgoals.

6.2.2 THEN : tactic -> tactic -> tactic

The tactical THEN is an ML infix. If T_1 and T_2 are tactics, then the ML expression T_1 THEN T_2 evaluates to a tactic which first applies T_1 and then applies T_2 to all the subgoals produced by T_1.

In fact, ASM_REWRITE_TAC[ADD] will solve the basis as well as the step case of the induction for $\forall m.\ m + 0 = m$, so there is an even simpler one-step proof than the one above:

```
- g `!m. m+0 = m`;                                                      1
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !m. m + 0 = m

- e(INDUCT_TAC THEN ASM_REWRITE_TAC[ADD]);
OK..
> val it =
    Initial goal proved.
     [] |- !m. m + 0 = m
```

This is typical: it is common to use a single tactic for several goals. Here, for example, are the first four consequences of the definition `ADD` of addition that are pre-proved when the built-in theory `arithmetic` HOL is made.

```
val ADD_0 = prove (
  ``!m. m + 0 = m``,
  INDUCT_TAC THEN ASM_REWRITE_TAC[ADD]);

val ADD_SUC = prove (
  ``!m n. SUC(m + n) = m + SUC n``,
  INDUCT_TAC THEN ASM_REWRITE_TAC[ADD]);

val ADD_CLAUSES = prove (
  ``(0 + m = m)              /\
    (m + 0 = m)              /\
    (SUC m + n = SUC(m + n)) /\
    (m + SUC n = SUC(m + n))``,
  REWRITE_TAC[ADD, ADD_0, ADD_SUC]);

val ADD_COMM = prove (
  ``!m n. m + n = n + m``,
  INDUCT_TAC THEN ASM_REWRITE_TAC[ADD_0, ADD, ADD_SUC]);
```

These proofs are performed when the HOL system is made and the theorems are saved in the theory `arithmetic`. The complete list of proofs for this built-in theory can be found in the file `src/num/arithmeticScript.sml`.

6.2.3 `ORELSE : tactic -> tactic -> tactic`

The tactical `ORELSE` is an ML infix. If T_1 and T_2 are tactics, then T_1 `ORELSE` T_2 evaluates to a tactic which applies T_1 unless that fails; if it fails, it applies T_2. `ORELSE` is defined in ML as a curried infix by[3]

(T_1 `ORELSE` T_2) g = T_1 g `handle _ =>` T_2 g

6.2.4 `REPEAT : tactic -> tactic`

If T is a tactic then `REPEAT` T is a tactic which repeatedly applies T until it fails. This can be illustrated in conjunction with `GEN_TAC`, which is specified by:

$$\frac{\texttt{!}x\,.\,t[x]}{t[x']}$$

[3]This is a minor simplification.

- Where x' is a variant of x not free in the goal or the assumptions.

`GEN_TAC` strips off one quantifier; `REPEAT GEN_TAC` strips off all quantifiers:

```
- g `!x y z. x+(y+z) = (x+y)+z`;                                          2
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !x y z. x + (y + z) = x + y + z

- e GEN_TAC;
OK..
1 subgoal:
> val it =
    !y z. x + (y + z) = x + y + z

- e(REPEAT GEN_TAC);
OK..
1 subgoal:
> val it =
    x + (y + z) = x + y + z
```

6.3 Some tactics built into HOL

This section contains a summary of some of the tactics built into the HOL system (including those already discussed). The tactics given here are those that are used in the parity checking example.

Before beginning, allow the ML type `thm_tactic` to abbreviate `thm->tactic`, and the type `conv`[4] to abbreviate `term->thm`.

6.3.1 `REWRITE_TAC : thm list -> tactic`

- **Summary:** `REWRITE_TAC[`th_1`,...,`th_n`]` simplifies the goal by rewriting it with the explicitly given theorems th_1, ... , th_n, and various built-in rewriting rules.

$$\frac{\{t_1,\ldots,t_m\}t}{\{t_1,\ldots,t_m\}t'}$$

where t' is obtained from t by rewriting with

1. th_1, ... , th_n and

[4]The type `conv` comes from Larry Paulson's theory of conversions [9].

2. the standard rewrites held in the ML variable `basic_rewrites`.

- **Uses:** Simplifying goals using previously proved theorems.

- **Other rewriting tactics**:

1. `ASM_REWRITE_TAC` adds the assumptions of the goal to the list of theorems used for rewriting.
2. `PURE_REWRITE_TAC` uses neither the assumptions nor the built-in rewrites.
3. `bossLib.RW_TAC` of type `simpLib.simpset -> thm list -> tactic`. A `simpset` is a special collection of rewriting theorems and other theorem-proving functionality. Values defined by HOL include `bossLib.base_ss`, which has basic knowledge of the boolean connectives, `bossLib.arith_ss` which "knows" all about arithmetic, and `HOLSimps.hol_ss`, which includes theorems appropriate for lists, pairs, and arithmetic. Additional theorems for rewriting can be added using the second argument of `RW_TAC`.

6.3.2 `CONJ_TAC : tactic`

- **Summary:** Splits a goal ``` ``$t_1$/\$t_2$`` ``` into two subgoals ``` ``$t_1$`` ``` and ``` ``$t_2$`` ```.

$$\frac{t_1 \;/\backslash\; t_2}{t_1 \qquad t_2}$$

- **Uses:** Solving conjunctive goals. `CONJ_TAC` is invoked by `STRIP_TAC` (see below).

6.3.3 `EQ_TAC : tactic`

- **Summary:** `EQ_TAC` splits an equational goal into two implications (the 'if-case' and the 'only-if' case):

$$\frac{u = v}{u \texttt{ ==> } v \qquad v \texttt{ ==> } u}$$

- **Use:** Proving logical equivalences, i.e. goals of the form "u=v" where u and v are boolean terms.

6.3.4 `DISCH_TAC : tactic`

- **Summary:** Moves the antecedent of an implicative goal into the assumptions.

$$\frac{u \texttt{ ==> } v}{\{u\}v}$$

- **Uses:** Solving goals of the form ``$u$ `==>` $v$`` by assuming ``$u$`` and then solving ``$v$``. `STRIP_TAC` (see below) will invoke `DISCH_TAC` on implicative goals.

6.3.5 `GEN_TAC : tactic`

- **Summary:** Strips off one universal quantifier.

$$\frac{\texttt{!}x\texttt{.}t[x]}{t[x']}$$

  Where x' is a variant of x not free in the goal or the assumptions.

- **Uses:** Solving universally quantified goals. `REPEAT GEN_TAC` strips off all universal quantifiers and is often the first thing one does in a proof. `STRIP_TAC` (see below) applies `GEN_TAC` to universally quantified goals.

6.3.6 `bossLib.PROVE_TAC : thm list -> tactic`

- **Summary:** Used to do first order reasoning, solving the goal completely if successful, failing otherwise. Using the provided theorems and the assumptions of the goal, `PROVE_TAC` does a search for possible proofs of the goal. Eventually fails if the search fails to find a proof shorter than a reasonable depth.

- **Uses:** To finish a goal off when it is clear that it is a consequence of the assumptions and the provided theorems.

6.3.7 `STRIP_TAC : tactic`

- **Summary:** Breaks a goal apart. `STRIP_TAC` removes one outer connective from the goal, using `CONJ_TAC`, `DISCH_TAC`, `GEN_TAC`, etc. If the goal is t_1`/\`$\cdots$`/\`t_n `==>` t then `DISCH_TAC` makes each t_i into a separate assumption.

- **Uses:** Useful for splitting a goal up into manageable pieces. Often the best thing to do first is `REPEAT STRIP_TAC`.

6.3.8 `SUBST_TAC : thm list -> thm`

- **Summary:** `SUBST_TAC[|-`u_1`=`v_1`,...,|-`u_n`=`v_n`]` converts a goal $t[u_1, \ldots, u_n]$ to the subgoal form $t[v_1, \ldots, v_n]$.
- **Uses:** To make replacements for terms in situations in which `REWRITE_TAC` is too general or would loop.

6.3.9 `ACCEPT_TAC : thm -> tactic`

- **Summary:** `ACCEPT_TAC` th is a tactic that solves any goal that is achieved by th.
- **Use:** Incorporating forward proofs, or theorems already proved, into goal directed proofs. For example, one might reduce a goal g to subgoals g_1, ..., g_n using a tactic T and then prove theorems th_1 , ..., th_n respectively achieving these goals by forward proof. The tactic

  ```
  T THENL[ACCEPT_TAC th₁,...,ACCEPT_TAC thₙ]
  ```

  would then solve g, where `THENL` is the tactical that applies the respective elements of the tactic list to the subgoals produced by `T`.

6.3.10 `ALL_TAC : tactic`

- **Summary:** Identity tactic for the tactical `THEN` (see *DESCRIPTION*).
- **Uses:**
  1. Writing tacticals (see description of `REPEAT` in *DESCRIPTION*).
  2. With `THENL`; for example, if tactic T produces two subgoals and we want to apply T_1 to the first one but to do nothing to the second, then the tactic to use is T `THENL[`T_1`;ALL_TAC]`.

6.3.11 `NO_TAC : tactic`

- **Summary:** Tactic that always fails.
- **Uses:** Writing tacticals.

Chapter 7

Example: a simple parity checker

This chapter consists of a worked example: the specification and verification of a simple sequential parity checker. The intention is to accomplish two things:

(i) To present a complete piece of work with HOL.

(ii) To give a flavour of what it is like to use the HOL system for a tricky proof.

Concerning (ii), note that although the theorems proved are, in fact, rather simple, the way they are proved illustrates the kind of intricate 'proof engineering' that is typical. The proofs could be done more elegantly, but presenting them that way would defeat the purpose of illustrating various features of HOL. It is hoped that the small example here will give the reader a feel for what it is like to do a big one.

Readers who are not interested in hardware verification should be able to learn something about the HOL system even if they do not wish to penetrate the details of the parity-checking example used here. The specification and verification of a slightly more complex parity checker is set as an exercise (a solution is provided).

7.1 Introduction

This case study is supported by three files in the HOL distribution directory. These files are:

```
examples/parity/PARITY.sml
examples/parity/RESET_REG.sml
examples/parity/RESET_PARITY.sml
```

The file `PARITY.sml` contains the HOL sessions in this chapter; the files `RESET_REG.sml` and `RESET_PARITY.sml` contain the solutions to the exercises described in Section 7.5.

The goal of the case study is to illustrate detailed 'proof hacking' on a small and fairly simple example.

The sessions of this example comprise the specification and verification of a device that computes the parity of a sequence of bits. More specifically, a detailed verification is given of a device with an input `in`, an output `out` and the specification that the nth output on out is T if and only if there have been an even number of T's input on in. A theory named PARITY is constructed; this contains the specification and verification of the device. All the ML input in the boxes below can be found in the file parity/PARITY.sml. It is suggested that the reader interactively input this to get a 'hands on' feel for the example.

7.2 Specification

The first step is to start up the HOL system. We will again use hol.unquote and start by loading and opening bossLib. The ML prompt is -, so lines beginning with - are typed by the user and other lines are the system's response.

```
% hol.unquote                                                           1
Moscow ML version 1.44 (August 1999)
  ...
[closing file "/local/scratch/mn200/Work/hol98/tools/unquote-init.sml"]
- load "bossLib";
> val it = () : unit
- open bossLib;
```

To specify the device, a primitive recursive function PARITY is defined so that for $n > 0$, PARITY nf is true if the number of T's in the sequence $f(1), \ldots, f(n)$ is even.

```
- val PARITY_def = Define`                                              2
    (PARITY 0 f = T) /\
    (PARITY(SUC n)f = if f(SUC n) then ~(PARITY n f) else PARITY n f)`;
Definition stored under "PARITY_def".
> val PARITY_DEF =
    |- (!f. PARITY 0 f = T) /\
       !n f. PARITY (SUC n) f =
             (if f (SUC n) then ~PARITY n f else PARITY n f)
    : thm
```

The effect of our call to Define is to store the definition of PARITY on the current theory with name PARITY_def and to bind the defining theorem to the ML variable with the same name. Notice that there are two name spaces being written into: the names of constants in theories and the names of variables in ML. The user is generally free to manage these names however he or she wishes (subject to the various lexical requirements), but a common convention is (as here) to give the definition of a constant CON the name CON_DEF in the theory and also in ML. Another commonly-used convention is to use just CON for the theory and ML name of the definition of a constant CON. Unfortunately, the HOL system does not use a uniform convention, but users are recommended to adopt one. In this case Define has made one of the choices for us, but there are other scenarios where have to choose the name used in the theory file.

The specification of the parity checking device can now be given as:

```
!t. out t = PARITY t inp
```

It is *intuitively* clear that this specification will be satisfied if the signal[1] functions `inp` and `out` satisfy[2]:

```
out(0) = T
```

and

```
!t. out(t+1)  =  (if inp(t+1) then ~(out t) else out t)
```

This can be verified formally in HOL by proving the following lemma:

```
!in out.
 (out 0 = T) /\ (!t. out(SUC t) = (if inp(SUC t) then ~(out t) else out t))
 ==>
 (!t. out t = PARITY t inp)
```

The proof of this is done by Mathematical Induction and, although trivial, is a good illustration of how such proofs are done. The lemma is proved interactively using HOL's subgoal package. The proof is started by putting the goal to be proved on a goal stack using the function `g` which takes a goal as argument.

```
- g `!inp out.                                                              3
        (out 0 = T) /\
        (!t. out(SUC t) = (if inp(SUC t) then ~(out t) else out t)) ==>
        (!t. out t = PARITY t inp)`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !inp out.
           (out 0 = T) /\
           (!t. out (SUC t) = (if inp (SUC t) then ~out t else out t)) ==>
           !t. out t = PARITY t inp
```

The subgoal package prints out the goal on the top of the goal stack. The top goal is expanded by stripping off the universal quantifier (with `GEN_TAC`) and then making the two conjuncts of the antecedent of the implication into assumptions of the goal (with `STRIP_TAC`). The ML function `expand` takes a tactic and applies it to the top goal; the resulting subgoals are pushed on to the goal stack. The message '`OK..`' is printed out just before the tactic is applied. The resulting subgoal is then printed.

[1]Signals are modelled as functions from numbers, representing times, to booleans.

[2]We'd like to use `in` as one of our variable names, but this is a reserved word for `let`-expressions.

```
- expand(REPEAT GEN_TAC THEN STRIP_TAC);                                   4
OK..
1 subgoal:
> val it =
    !t. out t = PARITY t inp
    ____________________________________
      0.  out 0 = T
      1.  !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

Next induction on `t` is done using `Induct`, which does induction on the outermost universally quantified variable.

```
- expand Induct;                                                           5
OK..
2 subgoals:
> val it =
    out (SUC t) = PARITY (SUC t) inp
    ____________________________________
      0.  out 0 = T
      1.  !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
      2.  out t = PARITY t inp

    out 0 = PARITY 0 inp
    ____________________________________
      0.  out 0 = T
      1.  !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

The assumptions of the two subgoals are shown in numbered underneath the horizontal lines of hyphens. The last goal printed is the one on the top of the stack, which is the basis case. This is solved by rewriting with its assumptions and the definition of `PARITY`.

```
- expand(ASM_REWRITE_TAC[PARITY_def]);                                     6
OK..

Goal proved.
 [.] |- out 0 = PARITY 0 inp

Remaining subgoals:
> val it =
    out (SUC t) = PARITY (SUC t) inp
    ____________________________________
      0.  out 0 = T
      1.  !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
      2.  out t = PARITY t inp
```

The top goal is proved, so the system pops it from the goal stack (and puts the proved theorem on a stack of theorems). The new top goal is the step case of the induction. This goal is also solved by rewriting.

```
- expand(ASM_REWRITE_TAC[PARITY_DEF]);                                    7
OK..

Goal proved.
 [..] |- out (SUC t) = PARITY (SUC t) inp

Goal proved.
 [..] |- !t. out t = PARITY t inp
> val it =
    Initial goal proved.
    |- !inp out.
         (out 0 = T) /\
         (!t. out (SUC t) = (if inp (SUC t) then ~out t else out t)) ==>
         !t. out t = PARITY t inp
```

The goal is proved, i.e. the empty list of subgoals is produced. The system now applies the justification functions produced by the tactics to the lists of theorems achieving the subgoals (starting with the empty list). These theorems are printed out in the order in which they are generated (note that assumptions of theorems are printed as dots).

The ML function

```
top_thm : unit -> thm
```

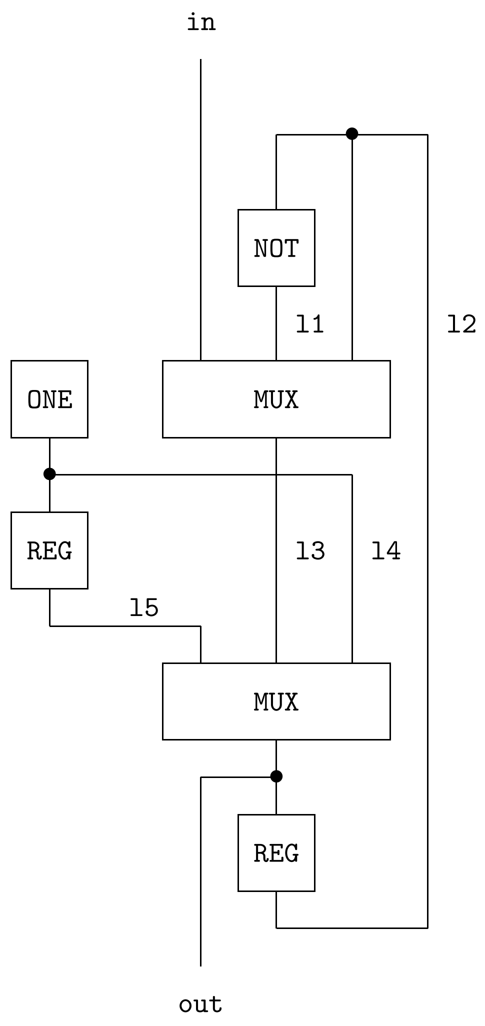
returns the theorem just proved (i.e. on the top of the theorem stack) in the current theory, and we bind this to the ML name `UNIQUENESS_LEMMA`.

```
- val UNIQUENESS_LEMMA = top_thm();                                       8
> val UNIQUENESS_LEMMA =
    |- !inp out.
         (out 0 = T) /\
         (!t. out (SUC t) = (if inp (SUC t) then ~out t else out t)) ==>
         !t. out t = PARITY t inp
    : thm
```

7.3 Implementation

The lemma just proved suggests that the parity checker can be implemented by holding the parity value in a register and then complementing the contents of the register whenever `T` is input. To make the implementation more interesting, it will be assumed that registers 'power up' storing `F`. Thus the output at time `0` cannot be taken directly from a register, because the output of the parity checker at time `0` is specified to be `T`. Another tricky thing to notice is that if `t>0`, then the output of the parity checker at time `t` is a function of the input at time `t`. Thus there must be a combinational path from the input to the output.

The schematic diagram below shows the design of a device that is intended to implement this specification. (The leftmost input to `MUX` is the selector.) This works by storing the parity of the sequence input so far in the lower of the two registers. Each time `T` is input at `in`, this stored value is complemented. Registers are assumed to 'power up' in a state in which they are storing `F`. The second register (connected to `ONE`) initially outputs `F` and then outputs `T` forever. Its role is just to ensure that the device works during the first cycle by connecting the output `out` to the device `ONE` via the lower multiplexer. For all subsequent cycles `out` is connected to `l3` and so either carries the stored parity value (if the current input is `F`) or the complement of this value (if the current input is `T`).



The devices making up this schematic will be modelled with predicates [4]. For example, the predicate `ONE` is true of a signal `out` if for all times `t` the value of `out` is `T`.

```
- val ONE_def = Define `ONE(out:num->bool) = !t. out t = T`;          9
Definition stored under "ONE_def".
> val ONE_def = |- !out. ONE out = !t. out t = T : thm
```

Note that, as discussed above, ‘`ONE_def`’ is used both as an ML variable and as the name of the definition in the theory. Note also how ‘`:num->bool`’ has been added to resolve type ambiguities; without this (or some other type information) the typechecker would not be able to infer that `t` is to have type `num`.

The binary predicate `NOT` is true of a pair of signals `(inp,out)` if the value of `out` is always the negation of the value of `inp`. Inverters are thus modelled as having no delay. This is appropriate for a register-transfer level model, but not at a lower level.

```
- val NOT_def = Define`NOT(inp, out:num->bool) = !t. out t = ~(inp t)`;   10
Definition stored under "NOT_def".
> val NOT_def = |- !inp out. NOT (inp,out) = !t. out t = ~inp t : Thm.thm
```

The final combinational device needed is a multiplexer. This is a ‘hardware conditional’; the input `sw` selects which of the other two inputs are to be connected to the output `out`.

```
- val MUX_def = Define`                                                  11
    MUX(sw,in1,in2,out:num->bool) =
      !t. out t = if sw t then in1 t else in2 t`;
Definition stored under "MUX_def".
> val MUX_def =
    |- !sw in1 in2 out.
         MUX (sw,in1,in2,out) = !t. out t = (if sw t then in1 t else in2 t)
    : thm
```

The remaining devices in the schematic are registers. These are unit-delay elements; the values output at time `t+1` are the values input at the preceding time `t`, except at time `0` when the register outputs `F`.[3]

```
- val REG_def =                                                          12
    Define `REG(inp,out:num->bool) =
              !t. out t = if (t=0) then F else inp(t-1)`;
Definition stored under "REG_def".
> val REG_def =
    |- !inp out. REG (inp,out) = !t. out t =
                 (if t = 0 then F else inp (t - 1))
    : thm
```

The schematic diagram above can be represented as a predicate by conjoining the relations holding between the various signals and then existentially quantifying the internal lines. This technique is explained elsewhere (e.g. see [3, 4]).

[3]Time 0 represents when the device is switched on.

```
- val PARITY_IMP_def = Define                                        13
   ‘PARITY_IMP(inp,out) =
      ?l1 l2 l3 l4 l5.
        NOT(l2,l1) /\ MUX(inp,l1,l2,l3) /\ REG(out,l2) /\
        ONE l4     /\ REG(l4,l5)        /\ MUX(l5,l3,l4,out)‘;
Definition stored under "PARITY_IMP_def".
> val PARITY_IMP_def =
    |- !inp out.
         PARITY_IMP (inp,out) =
         ?l1 l2 l3 l4 l5.
           NOT (l2,l1) /\ MUX (inp,l1,l2,l3) /\ REG (out,l2) /\ ONE l4 /\
           REG (l4,l5) /\ MUX (l5,l3,l4,out)
    : thm
```

7.4 Verification

The following theorem will eventually be proved:

```
|- !inp out. PARITY_IMP(inp,out) ==> (!t. out t = PARITY t inp)
```

This states that *if* `inp` and `out` are related as in the schematic diagram (i.e. as in the definition of `PARITY_IMP`), *then* the pair of signals `(inp,out)` satisfies the specification.

First, the following lemma is proved; the correctness of the parity checker follows from this and `UNIQUENESS_LEMMA` by the transitivity of `==>`.

```
- g ‘!inp out.                                                       14
        PARITY_IMP(inp,out) ==>
        (out 0 = T) /\
        !t. out(SUC t) = if inp(SUC t) then ~(out t) else out t‘;
> val it =
    Proof manager status: 2 proofs.
    2. Completed: ...
    1. Incomplete:
         Initial goal:
         !inp out.
           PARITY_IMP (inp,out) ==>
           (out 0 = T) /\
           !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

The first step in proving this goal is to rewrite with definitions followed by a decomposition of the resulting goal using `STRIP_TAC`. The rewriting tactic `PURE_REWRITE_TAC` is used; this does no built-in simplifications, only the ones explicitly given in the list of theorems supplied as an argument. One of the built-in simplifications used by `REWRITE_TAC` is `|- (x = T) = x`. `PURE_REWRITE_TAC` is used to prevent rewriting with this being done.

```
- expand(PURE_REWRITE_TAC                                              15
           [PARITY_IMP_def, ONE_def, NOT_def, MUX_def, REG_def] THEN
         REPEAT STRIP_TAC);
OK..
2 subgoals:
> val it =
    out (SUC t) = (if inp (SUC t) then ~out t else out t)
    ____________________________________
      0.  !t. l1 t = ~l2 t
      1.  !t. l3 t = (if inp t then l1 t else l2 t)
      2.  !t. l2 t = (if t = 0 then F else out (t - 1))
      3.  !t. l4 t = T
      4.  !t. l5 t = (if t = 0 then F else l4 (t - 1))
      5.  !t. out t = (if l5 t then l3 t else l4 t)

    out 0 = T
    ____________________________________
      0.  !t. l1 t = ~l2 t
      1.  !t. l3 t = (if inp t then l1 t else l2 t)
      2.  !t. l2 t = (if t = 0 then F else out (t - 1))
      3.  !t. l4 t = T
      4.  !t. l5 t = (if t = 0 then F else l4 (t - 1))
      5.  !t. out t = (if l5 t then l3 t else l4 t)
```

The top goal is the one printed last; its conclusion is `out 0 = T` and its assumptions are equations relating the values on the lines in the circuit. The natural next step would be to expand the top goal by rewriting with the assumptions. However, if this were done the system would go into an infinite loop because the equations for `out`, `l2` and `l3` are mutually recursive. Instead we use the first-order reasoner `PROVE_TAC` to do the work:

```
- expand(PROVE_TAC []);                                                16
OK..
Meson search level: .....

Goal proved.
 [......] |- out 0 = T

Remaining subgoals:
> val it =
    out (SUC t) = (if inp (SUC t) then ~out t else out t)
    ____________________________________
      0.  !t. l1 t = ~l2 t
      1.  !t. l3 t = (if inp t then l1 t else l2 t)
      2.  !t. l2 t = (if t = 0 then F else out (t - 1))
      3.  !t. l4 t = T
      4.  !t. l5 t = (if t = 0 then F else l4 (t - 1))
      5.  !t. out t = (if l5 t then l3 t else l4 t)
```

The first of the two subgoals is proved. Inspecting the remaining goal it can be seen that it will be solved if its left hand side, `out(SUC t)`, is expanded using the assumption:

```
!t. out t = if l5 t then l3 t else l4 t
```

However, if this assumption is used for rewriting, then all the subterms of the form `out t` will also be expanded. To prevent this, we really want to rewrite with a formula that is specifically about `out (SUC t)`. We want to somehow pull the assumption that we do have out of the list and rewrite with a specialised version of it. We can do just this using `PAT_ASSUM`. This tactic is of type `term -> thm -> tactic`. It selects an assumption that is of the form given by its term argument, and passes it to the second argument, a function which expects a theorem and returns a tactic. Here it is in action:

```
- e (PAT_ASSUM ``!t. out t = X t``                                        17
       (fn th => REWRITE_TAC [SPEC ``SUC t`` th]))
<<HOL message: inventing new type variable names: 'a, 'b.>>
OK..
1 subgoal:
> val it =
    (if l5 (SUC t) then l3 (SUC t) else l4 (SUC t)) =
    (if inp (SUC t) then ~out t else out t)
    ------------------------------------
      0.  !t. l1 t = ~l2 t
      1.  !t. l3 t = (if inp t then l1 t else l2 t)
      2.  !t. l2 t = (if t = 0 then F else out (t - 1))
      3.  !t. l4 t = T
      4.  !t. l5 t = (if t = 0 then F else l4 (t - 1))
```

The pattern used here exploited something called *higher order matching*. The actual assumption that was taken off the assumption stack did not have a RHS that looked like the application of a function (`X` in the pattern) to the `t` parameter, but the RHS could nonetheless be seen as equal to the application of *some* function to the `t` parameter. In fact, the value that matched `X` was ``` ``\x. if l5 x then l3 x else l4 x`` ```.

Inspecting the goal above, it can be seen that the next step is to unwind the equations for the remaining lines of the circuit. We do this using the `arith_ss` simpset that comes with `bossLib` to help with the arithmetic embodied by the subtractions and `SUC` terms.

```
- e (RW_TAC arith_ss []);                                        18
OK..

Goal proved.
 [.....]
|- (if l5 (SUC t) then l3 (SUC t) else l4 (SUC t)) =
   (if inp (SUC t) then ~out t else out t)

Goal proved.
 [......] |- out (SUC t) = (if inp (SUC t) then ~out t else out t)
> val it =
    Initial goal proved.
    |- !inp out.
         PARITY_IMP (inp,out) ==>
         (out 0 = T) /\
         !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

The theorem just proved is named `PARITY_LEMMA` and saved in the current theory.

```
- val PARITY_LEMMA = top_thm ();                                 19
> val PARITY_LEMMA =
    |- !inp out.
         PARITY_IMP (inp,out) ==>
         (out 0 = T) /\
         !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

`PARITY_LEMMA` could have been proved in one step with a single compound tactic. Our initial goal can be expanded with a single tactic corresponding to the sequence of tactics that were used interactively:

```
- restart()                                                      20
> ...
- e (PURE_REWRITE_TAC [PARITY_IMP_def, ONE_def, NOT_def,
                       MUX_def, REG_def] THEN
     REPEAT STRIP_TAC THENL [
       PROVE_TAC [],
       PAT_ASSUM ``!t. out t = X t``
                 (fn th => REWRITE_TAC [SPEC ``SUC t`` th]) THEN
       RW_TAC arith_ss []
     ]);
<<HOL message: inventing new type variable names: 'a, 'b.>>
OK..
Meson search level: .....
> val it =
    Initial goal proved.
    |- !inp out.
         PARITY_IMP (inp,out) ==>
         (out 0 = T) /\
         !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

Armed with PARITY_LEMMA, the final theorem is easily proved. This will be done in one step using the ML function prove.

```
- val PARITY_CORRECT = prove(                                                  21
    ``!inp out. PARITY_IMP(inp,out) ==> (!t. out t = PARITY t inp)``,
    REPEAT STRIP_TAC THEN MATCH_MP_TAC UNIQUENESS_LEMMA THEN
    MATCH_MP_TAC PARITY_LEMMA THEN ASM_REWRITE_TAC []);
> val PARITY_CORRECT =
    |- !inp out. PARITY_IMP (inp,out) ==> !t. out t = PARITY t inp
```

This completes the proof of the parity checking device.

7.5 Exercises

Two exercises are given in this section: Exercise 1 is straightforward, but Exercise 2 is quite tricky and might take a beginner several days to solve. The solutions to these exercises should be in the files:

```
hol/examples/parity/RESET_REG.sml
hol/examples/parity/RESET_PARITY.sml
```

7.5.1 Exercise 1

Using *only* the devices ONE, NOT, MUX and REG defined in Section 7.3, design and verify a register RESET_REG with an input in, reset line reset, output out and behaviour specified as follows.

- If reset is T at time t, then the value at out at time t is also T.
- If reset is T at time t or t+1, then the value output at out at time t+1 is T, otherwise it is equal to the value input at time t on inp.

This is formalized in HOL by the definition:

```
RESET_REG(reset,inp,out) =
 (!t. reset t ==> (out t = T)) /\
 (!t. out(t+1) = ((reset t  \/ reset(t+1)) => T | inp t))
```

Note that this specification is only partial; it doesn't specify the output at time 0 in the case that there is no reset.

The solution to the exercise should be a definition of a predicate RESET_REG_IMP as an existential quantification of a conjunction of applications of ONE, NOT, MUX and REG to suitable line names,[4] together with a proof of:

```
RESET_REG_IMP(reset,inp,out) ==> RESET_REG(reset,inp,out)
```

[4]i.e. a definition of the same form as that of PARITY_IMP on page 84.

7.5.2 Exercise 2

1. Formally specify a resetable parity checker that has two boolean inputs `reset` and `inp`, and one boolean output `out` with the following behaviour:

   > The value at `out` is `T` if and only if there have been an even number of `T`s input at `inp` since the last time that `T` was input at `reset`.

2. Design an implementation of this specification built using *only* the devices `ONE`, `NOT`, `MUX` and `REG` defined in Section 7.3.

3. Verify the correctness of your implementation in HOL.

Chapter 8

More examples

In addition to the examples already covered in this text, the hol98 distribution comes with a variety of instructive examples in the `examples` directory. There the following examples (among others) are to be found:

`autopilot.sml` This example is a hol98 rendition (by Mark Staples) of a PVS example due to Ricky Butler of NASA. The example shows the use of the record-definition package, as well as illustrating some aspects of the automation available in hol98.

`bmark` In this directory, there is a standard HOL benchmark: the proof of correctness of a multiplier circuit, due to Mike Gordon.

`euclid.sml` This example is the same as that covered in chapter 4: a proof of Euclid's theorem on the infinitude of the prime numbers, extracted and modified from a much larger development due to John Harrison. It illustrates the automation of HOL on a classic proof.

`ind_def` This directory contains some examples of an inductive definition package in action. Featured are an operational semantics for a small imperative programming language, a small process algebra, and combinatory logic with its type system. The files were originally developed by Tom Melham and Juanito Camilleri and are extensively commented.

Most of the proofs in these theories can now be done much more easily by using some of the recently developed proof tools, namely the simplifier and the first order prover.

`fol.sml` This file illustrates John Harrison's implementation of a model-elimination style first order prover.

`lambda` This directory develops theories of a "de Bruijn" style lambda calculus, and also a name-carrying version. (Both are untyped.) The development is a revision of the proofs underlying the paper *"5 Axioms of Alpha Conversion", Andy Gordon and Tom Melham, Proceedings of TPHOLs'96, Springer LNCS 1125*.

`parity` This sub-directory contains the files used in the parity example of chapter 7.

`MLsyntax` This sub-directory contains an extended example of a facility for defining mutually recursive types, due to Elsa Gunter of Bell Labs. In the example, the type of abstract syntax for a small but not totally unrealistic subset of ML is defined, along with a simple mutually recursive function over the syntax.

`Thery.sml` A very short example due to Laurent Thery, demonstrating a cute inductive proof.

`RSA` This directory develops some of the mathematics underlying the RSA cryptography scheme. The theories have been produced by Laurent Thery of INRIA Sophia-Antipolis.

References

[1] S.F. Allen, R.L. Constable, D.J. Howe and W.E. Aitken, 'The Semantics of Reflected Proof', *Proceedings of the 5th IEEE Symposium on Logic in Computer Science*, pp. 95–105, 1990.

[2] R.S. Boyer and J S. Moore, 'Metafunctions: Proving Them Correct and Using Them Efficiently as New Proof Procedures', in: *The Correctness Problem in Computer Science*, edited by R.S. Boyer and J S. Moore, Academic Press, New York, 1981.

[3] A.J. Camilleri, T.F. Melham and M.J.C. Gordon, 'Hardware Verification using Higher-Order Logic', in: *From HDL Descriptions to Guaranteed Correct Circuit Designs: Proceedings of the IFIP WG 10.2 Working Conference, Grenoble, September 1986*, edited by D. Borrione (North-Holland, 1987), pp. 43–67.

[4] M. Gordon, 'Why higher-order Logic is a good formalism for specifying and verifying hardware', in: *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, edited by G. Milne and P.A. Subrahmanyam (North-Holland, 1986), pp. 153–177.

[5] Donald. E. Knuth. *The Art of Computer Programming*. Volume 1/Fundamental Algorithms. Addison-Wesley, second edition, 1973.

[6] Saunders Mac Lane and Garrett Birkhoff. *Algebra*. Collier-MacMillan Limited, London, 1967.

[7] R. Milner, 'A Theory of Type Polymorphism in Programming', *Journal of Computer and System Sciences*, Vol. 17 (1978), pp. 348–375.

[8] George D. Mostow, Joseph H. Sampson, and Jean-Pierre Meyer. *Fundamental Structures of Algebra*. McGraw-Hill Book Company, 1963.

[9] L. Paulson, 'A Higher-Order Implementation of Rewriting', *Science of Computer Programming*, Vol. 3, (1983), pp. 119–149.

[10] L. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2 (Cambridge University Press, 1987).

[11] R.E. Weyhrauch, ‘Prolegomena to a theory of mechanized formal reasoning’, *Artificial Intelligence* **3(1)**, 1980, pp. 133–170.

[12] A.N. Whitehead and B. Russell, *Principia Mathematica*, 3 volumes (Cambridge University Press, 1910–3).