

HOL98 Draft User's Manual
Athabasca Release
Version 2

Konrad Slind
kxs@cl.cam.ac.uk
Cambridge University Computer Laboratory

January 17, 1999

Introduction

Hol98 is the latest in a line of verification systems originating from the original HOL system [4], which was itself derived from Edinburgh LCF [3]. However, Hol98 is a definite break with the past: it has been re-designed to address the requirements of industrial-scale formal proof in the late 1990s.

Hol98 provides an extensible set of facilities both for doing verification and for writing verification tools. The system contains the work of many people, accumulated over decades. (To those who know the authorship of the tools, there is a slight flavour of being in a museum.) As a result, Hol98 offers many ways to achieve a given task; this is understandable—but tremendously frustrating for beginners. The point of this document therefore, is to give a brief survey of the important facilities available in Hol98; we will give only the ‘big picture’, along with some hints for deeper exploration. The reader who completes the document should have an idea of how to formalize and prove simple things in Hol98, as well as an idea of how to go about extending the system to his or her own purposes.

We proceed as follows. In Chapter One, we give a quick lesson on how to start Hol98. In Chapter Two, the syntax of HOL (higher-order logic, the logic that Hol98 implements) is described. The basic methods of proof that the system provides are surveyed in Chapter Three. In Chapter Four, brief summaries of the available theories and libraries are given. In Chapter Five, we describe the means by which users can build and maintain their own logical objects. In Chapter Six we discuss a package for high-level interactive proof. Some medium-length examples are the subject of Chapter Seven (at present we have only a detailed proof of Euclid’s theorem to offer). Chapter Eight (currently absent) contains listings of the most important parts of the programmer’s interface.

SML’97 [5] plays the role of a *programming metalanguage* in which the HOL logic is defined. Currently, the implementation is in MoscowML,¹ an implementation of a subset of SML’97. Thus Hol98 comprises a number of ML² modules. There are two main ways to utilize these modules: they can be used during an interactive proof effort; or they can be used as libraries in the writing of custom proof tools. Both usages require some knowledge of ML, the former much less than the latter. We shall assume that the reader knows some Standard ML. Several good texts on SML already exist; the ones by Paulson [6] and Ullman [7] have been updated for SML’97.

Finally, you will find that this user’s manual is filled with gaps. There are many positive ways to deal with these, among them being (1) asking a guru, either locally or on the `info-hol` mailing list, (2) attempting to extrapolate the relevant information from [4] (which should in any case be used as a background reference to the often facile treatments given here), or (3) diving into the sources, which is the path to guruhood.

¹<http://www.dina.kvl.dk/~sestoft/mosml.html>

²We will use ML and SML as synonyms for SML’97.

Acknowledgements

As mentioned, Hol98 can be regarded as a conglomeration and synthesis of the work of many people over a long period of time. We shall refrain from an exhaustive listing of authorship (in this draft) since we would not like to omit anyone! Therefore, we limit ourselves to the much smaller set of people who directly contributed to this release. Mike Gordon had the initial idea for tagged inference and pointed out that tags should be abstract. Ken Larsen got us started on using MoscowML, and made the initial port of many of the core libraries. He also implemented `Holmake`. Mark Staples developed the PVS `autopilot` example. Mike Norrish wrote and documented the `RecordType` package, and served some hard time consulting on the syntactical perversities of various Unix shells. Finally, Peter Sestoft, the main developer of MoscowML, has been very responsive to our requests for changes to his compiler.

Acknowledgements for Version 2

Joe Hurd ported John Harrison's development of the real numbers and analysis, including a decision procedure. Graham Collins donated his theory of finite maps. Ken Larsen and Mike Gordon ported Jorn Lind's BuDDy package (<http://britta.it.dtu.dk/~jl/buddy>) to MoscowML and integrated it with Hol98. Norbert Voelker and Louise Dennis spotted some installation problems, one of which Mike Norrish fixed.

Contents

1	Getting Started	5
1.1	Help	5
1.2	Input and Output	6
1.2.1	Quotation preprocessing	7
1.3	First proof	7
2	Syntax	10
2.1	Types	10
2.2	Terms	11
2.2.1	Constants	12
2.2.2	Type constraints	13
2.2.3	Expanded term grammar	14
3	Proof	16
3.1	Rules of inference	16
3.2	Oracles	16
3.3	Tactics	18
3.4	Conversions	18
3.5	Theorem Continuations	19
3.6	Simplifiers and Automatic Reasoners	19
3.6.1	Automatic methods	20
3.7	Definition principles	21
3.7.1	Record types	21
3.8	A simple proof manager	24
3.8.1	Starting a goalstack proof	24
3.8.2	Applying a tactic to a goal	25
3.8.3	Undo	25
3.8.4	Viewing the state of the proof manager	25
3.8.5	Switch focus to a different subgoal or proof attempt	26
4	Existing Context	27
4.1	Theories	27
4.2	Libraries	28

5	Building Logical Developments	31
5.1	Theories	31
5.1.1	Building a theory	31
5.1.2	Information functions	33
5.1.3	Theories and the file system	33
5.2	Holmake	34
5.2.1	System Rebuild	34
5.2.2	Theory construction	34
5.2.3	Making the script separately compilable	35
5.2.4	Summary	37
5.2.5	What Holmake doesn't do	37
5.2.6	Options to Holmake	37
6	High-level interactive proof support	38
6.1	Datatype definition	38
6.2	Function definition	40
6.3	Automated reasoners	42
7	Examples	44
7.1	Euclid's Theorem	44
7.1.1	Divisibility	47
7.1.2	Primality	60
7.1.3	Existence of prime factors	61
7.1.4	Euclid's theorem	64
7.1.5	Summary	67
8	The Programmer's Interface	70
	References	71

Getting Started

Starting up Hol98 is straightforward:

```
<hol-dir>/bin/hol a0 ... an
```

where an argument a_i to the invocation can be either (1) a path (prefixed by `-I`) where the system can find code or, (2) a file to execute before turning control over to the user. For example, invoking

```
<hol-dir>/bin/hol -I /local/foo /home/me/my-hol-init.sml
```

will do the following:

- The path `/local/foo` is added to the pre-existing path. If the user later asks for a module `X` to be loaded, the system will first look in each directory on the pre-existing path before looking in `/local/foo` for `X`. Usually, no paths need be added to the system path.
- The ML file `/home/me/my-hol-init.sml` will be executed. The order of execution of start-up files is left-to-right. There is a system start-up file that gets executed before any user-given ones: it can be found in the file `std.prelude` in the top level of the Hol98 distribution directory. The code in `std.prelude` will load and open some standard basic support (tactics, conversions, simple definition principles) and set up the paths to all the system libraries.

As usual, at least in Unix systems, Hol98 can be exited by entering `^D`. Invoking `quit()`; will serve the same purpose.

1.1 Help

There are several kinds of help available in Hol98, all accessible through the same incantation:

```
help <string>;
```

The kinds of help available are:

MoscowML help. This is uniformly excellent. Information for library routines is available, whether the library is loaded or not via `help "Lib"`.

HOL overview. This is a short summary of important information about Hol98.

HOL help. This is the on-line help from Hol88 and Hol90, and is intended to document all HOL-specific functions available to the user. It is very detailed and often accurate; however, it can be out-of-date, refer to HOL90 or HOL88, or even be missing!

HOL structure information. For most structures in the Hol98 source, one can get a listing of the entrypoints found in the accompanying signature. This is helpful for locating functions and is automatically derived from the system sources, so it is always up-to-date.

Theory facts. These are automatically derived from theory files, so they are always up-to-date. The signature of each theory is available (since theories are represented by structures in Hol98). Also, each axiom, definition, and theorem in the theory can be accessed by name in the help system; the theorem itself is given.

Therefore the following example queries can be made:

<code>help "installPP"</code>	Moscow ML help
<code>help "hol"</code>	Hol98 overview
<code>help "aconv"</code>	on-line HOL help
<code>help "Tactic"</code>	HOL source structure information
<code>help "boolTheory"</code>	theory structure signature
<code>help "list_Axiom"</code>	theory structure signature and theorem statement

1.2 Input and Output

A person usually works with Hol98 by interacting with the ML top level loop¹ in order to build formalizations and perform proofs. In this setting, the user often needs to enter expressions of the HOL logic to ML, and interpret the resulting responses. Since the ML representations of the types, terms, and theorems of the HOL logic are quite unreadable in their ‘raw’ form, so-called *prettyprinters* for HOL logic expressions are automatically invoked by the ML top level when printing output.

Similarly, types and terms often have to be constructed by the user, e.g., in order to make definitions, state goals to prove, provide existential witnesses, etc. Since it would be unbearable to make a type or term of any size ‘by hand’, the system comes equipped with parsers for type and term expressions. The parser for types is called `Type`, and the

¹So far, that is; one of the intended applications of Hol98 is for building batch theorem proving tools.

parser for terms is called `Term`. These parsers take *quotations*. A quotation ‘...’ is much like an SML string, except that it can span several lines without requiring awkward backslashes, as an ML string would.²

1.2.1 Quotation preprocessing

For convenience, the Athabasca release supplies a version of Hol98 that features a *combined parser* that accepts both types and terms. Enclosing some object language concrete syntax between occurrences of ‘‘ will result in the correct parser being invoked. For example

```
‘‘x /\ y /\ z ==> ?p. p’’
```

will parse as a term while

```
‘‘:’a -> (’b -> ’h) -> bool’’
```

parses as an HOL type. Note that the concrete syntax given in the quotation needs to provide a hint: the type parser will only be called if the first character after the leading ‘‘ is a colon (:).

Knowledgeable ML users will notice that the idiom ‘...’ is not ML-typable; it is implemented as a pre-processor to ML, thanks to work by Richard Boulton. Preprocessing is not the default in this release. Users who wish to use the pre-processor should use

```
<hol-dir>/bin/hol.enquote
<hol-dir>/bin/Holmake.enquote.
```

Input containing instances of ‘...’ will be accepted by these versions of Hol98.

1.3 First proof

In this section we show a simple goal being stated and solved in Hol98. Full explanations of what is going on can be found in the rest of this document, but we needn’t wait to exercise the system.

First, we start up the system. The start-up phase executes a standard prelude. (System responses occur on lines starting with >.)

```
bash$ /home/kxs/hol98/bin/hol
> Enter ‘quit();’ to quit.
> For HOL help, type: help "hol";
>
```

²Quotations were a feature in the original LCF system. See the MoscowML User’s Manual for more information.

```

>
>      HHH                LL
>      HHH                LL
>      HHH                LL
>      HHH                LL
>      HHH      0000      LL
>      HHHHHHH  00 00    LL
>      HHHHHHH  00 00    LLL
>      HHH      0000      LLLL
>      HHH                LL LL
>      HHH                LL  LL
>      HHH                LL  LL
>      HHH                LL   LL98 [Athabasca 2]
>
> [closing file "/home/kxs/hol98/std.prelude"]

```

Now we load an automatic first order reasoner, written by John Harrison (and ported by Michael Norrish from Harrison's HOL-Light system).³

```

- load"mesonLib";
> val it = () : unit

```

Now we will set a goal for the reasoner to prove. In mathematical notation, it is

$$\forall R. (\forall x. \exists y. R x y) = \exists f. \forall x. R x (f x).$$

This theorem is the justification of Skolemization. It says that every x is related to a y by R if and only if there is a function f mapping each x to its corresponding y . In Hol98 notation, we have:

```

- set_goal([], Term '(!R. (!x. ?y. R x y) = ?f. !x. R x (f x)');
> <<HOL message: inventing new type variable names: 'a, 'b.>>
> val it =
>   Proof manager status: 1 proof.
>   1. Incomplete:
>     Initial goal:
>     ‘‘!R. (!x. ?y. R x y) = (?f. !x. R x (f x))‘‘

```

Now we apply the reasoner to the goal with the `e` command. It says “OK..” and sets to work. As it searches for a proof, it prints out a row of dots. The proof is found in about a third of a second, and consists of 615 inference steps in the HOL logic.

³One might think that this sort of tool should already be “part of” Hol98; however, all of the reasoners of Hol98 are in external libraries. This is a consequence of a fundamental design tenet for LCF-style systems: keep a very simple kernel to the implementation, and add libraries on top.

```
- e (mesonLib.MESON_TAC[]);  
> OK..  
> Meson search level: ....  
> val it =  
>   Initial goal proved.  
>   |- !R. (!x. ?y. R x y) = (?f. !x. R x (f x))
```

Now we can extract the proved theorem, and bind it to a name in ML.

```
- val Skolem = top_thm();  
> val Skolem = |- !R. (!x. ?y. R x y) = (?f. !x. R x (f x)) : Thm.thm
```

Syntax

The HOL logic is a classical higher-order predicate calculus. Its syntax enjoys two main differences from the syntax of standard first order logic.¹ First, there is no distinction in HOL between terms and formulas: HOL has only terms. Second, each term has a type: types are used in order to build well-formed terms. There are two ways to construct types and terms in HOL: by use of a parser, or by use of the programmer's interface. In this chapter, we will focus on the concrete syntax accepted by the parsers, leaving the programmer's interface for Chapter 8.

2.1 Types

A HOL type can be a variable, a constant, or a compound type, which is a constant of arity n applied to a list of n types.

<i>hol_type</i>	::=	' <i>ident</i>	(type variable)
		bool	(type of truth values)
		ind	(type of individuals)
		<i>hol_type</i> -> <i>hol_type</i>	(function arrow)
		<i>ident</i>	(nullary type constant)
		<i>hol_type ident</i>	(unary compound type)
		(<i>hol_type</i> ₁ , ..., <i>hol_type</i> _{<i>n</i>}) <i>ident</i>	(compound type)

Type constants are also known as type operators. They must be alphanumeric. Type variables are alphanumerics written with a leading prime ('). In Hol98, the type constants **bool**, **fun**, and **ind** are primitive. The introduction of new type constants is described in Chapter 5. **bool** is the two element type of truth values. The binary operator **fun** is used to denote function types; it can be written with an infix arrow. The nullary type constant **ind** denotes an infinite set of individuals; it is used for a few highly technical developments in the system and can be ignored by beginners. Thus

```
'a -> 'b
(bool -> 'a) -> ind
```

are both well-formed types. The function arrow is "right associative", which means that ambiguous uses of the arrow in types are resolved by adding parentheses in a right-to-left sweep: thus the type expression

¹We assume the reader is familiar with first order logic.

`ind -> ind -> ind -> ind`

is identical to

`ind -> (ind -> (ind -> ind)).`

The product (#) and sum (+) are other infix type operators, also right associative; however, they are not loaded by default in Hol98. How to load in useful logical context is dealt with in Chapter 4.

2.2 Terms

Ultimately, a HOL term can only be a variable, a constant, an application, or a lambda term.

```

term ::= ident      (variable or constant)
      | term term   (combination)
      | \ident. term (lambda abstraction)

```

In the system, the usual logical operators have already been defined, including truth (T), falsity (F), negation (~), equality (=), conjunction (/&), disjunction (\/), implication (==>), universal (!) and existential (?) quantification, and an indefinite description operator (@). As well, the basis includes conditional, lambda, and 'let' expressions. Thus the set of terms available is, in general, an extension of the following grammar:

```

term ::= term : hol_type      (type constraint)
      | term term             (application)
      | ~ term                 (negation)
      | term = term           (equality)
      | term ==> term         (implication)
      | term \/ term          (disjunction)
      | term /\ term          (conjunction)
      | term => term | term    (conditional)
      | \ident1 ... identn. term (lambda abstraction)
      | !ident1 ... identn. term (forall)
      | ?ident1 ... identn. term (exists)
      | @ident1 ... identn. term (choose)
      | ?!ident1 ... identn. term (exists-unique)
      | let ident = term
      | [and ident = term]* in term (let expression)
      | T                       (truth)
      | F                       (falsity)
      | ident                   (constant or variable)
      | (term)                  (parenthesized term)

```

Some examples may be found in Table 2.1. Term application can be iterated. Application is left associative so that *term term term... term* is equivalent in the eyes of the parser to $(\dots((\textit{term term}) \textit{term}) \dots) \textit{term}$.

The lexical structure for term identifiers is much like that for ML: identifiers can be alphanumeric or symbolic. Variables must be alphanumeric. A symbolic identifier is any concatenation of the characters in the following list:

`#?+*/\=\<>%@!,:;_ | ~ -`

with the exception of the keywords `\`, `;`, `=>`, `|`, and `:` (colon). Any alphanumeric can be a constant except the keywords `let`, `in`, `and`, and `of`.

<code>x = T</code>	<i>x is equal to true.</i>
<code>!x. Person x ==> Mortal x</code>	<i>All persons are mortal.</i>
<code>!x y z. (x ==> y) /\ (y ==> z) ==> x ==> z</code>	<i>Implication is transitive.</i>
<code>!x. P x ==> Q x</code>	<i>P is a subset of Q</i>
<code>S = \f g x. f x (g x)</code>	<i>Definition of a famous combinator.</i>

Table 2.1: Concrete Syntax Examples

2.2.1 Constants

The HOL grammar gets extended when a new constant is introduced. The introduction of new constants will be discussed in section 5. In order to provide some notational flexibility, constants come in various flavours: besides being an ordinary constant, a constant could also be a *binder* or an *infix*.

2.2.1.1 Binders

A binder is a construct that binds a variable; for example, the universal quantifier. In HOL, this is represented using a trick that goes back to Alonzo Church: a binder is a constant that takes a lambda abstraction as its argument. The lambda binding is used to implement the binding of the construct. This is an elegant and uniform solution. Thus the concrete syntax `!v. M` is represented by the application of the constant `!` to the abstraction `(\v. M)`.

The most common binders are `!`, `?`, `?!`, and `@`. Sometimes one wants to iterate applications of the same binder, e.g.,

`!x. !y. ?p. ?q. ?r. term.`

This can instead be rendered

`!x y. ?p q r. term.`

2.2.1.2 Infixes

All infix constants associate to the right. The precedence ordering for the initial set of infixes is \wedge , \vee , \implies , $=$, $,$ (comma²). Thus

$$X \wedge Y \implies C \vee D, P = E, Q$$

is equal to

$$((X \wedge Y) \implies (C \vee D)), ((P = E), Q).$$

An expression *term* <infix> *term* is internally represented as $((\langle \text{infix} \rangle \text{ term}) \text{ term})$.

2.2.2 Type constraints

A term can be constrained to be of a certain type. For example, `X:bool` constrains the variable `X` to have type `bool`. Similarly, `T:bool` performs a (vacuous) constraint of the constant `T` to `bool`. An attempt to constrain a term inappropriately will raise an exception: for example,

$$T \Rightarrow (X:\text{ind}) \mid (Y:\text{bool})$$

will fail because both branches of a conditional must be of the same type. Type constraints can be seen as an infix that binds more weakly than any term constant. Thus *term* ... *term* : *hol_type* is equal to $(\text{term} \dots \text{term}) : \text{hol_type}$. For example, the prospective term `(x, y : bool)` will not be accepted, since we are attempting to constrain a pair to be a boolean.

The inclusion of `:` in the symbolic identifiers means that some constraints may need to be separated by white space. For example,

$$\$=: \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$$

will be broken up by the HOL lexer as

$$\$=: \text{ bool } \rightarrow \text{ bool } \rightarrow \text{ bool}$$

and parsed as an application of the symbolic identifier `$=:` to the argument list of terms `[bool, ->, bool, ->, bool]`. A well-placed space will avoid this problem:

$$\$= : \text{bool} \rightarrow \text{bool} \rightarrow \text{bool}$$

is parsed as the symbolic identifier `"="` constrained by a type.

²When `pairTheory` has been loaded

2.2.2.1 Type inference

Consider the term $x = T$. Each term (and all of its subterms), has a type in the HOL logic. Now, T has type `bool`. This means that the constant `=` has type `xty -> bool -> bool`, for some type `xty`. Since the type scheme for `=` is `'a -> 'a -> bool`, we know that `xty` must in fact be `bool` in order for the type instance to be well-formed. Knowing this, we can deduce that the type of `'x'` must be `bool`.

Ignoring the jargon ("scheme" and "instance") in the previous paragraph, we have conducted a type assignment to the term structure, ending up with a well-typed term. It would be very tedious for users to conduct such argumentation by hand for each term entered to Hol98. Thus, Hol98 uses an adaptation of Milner's type inference algorithm for ML when constructing terms via parsing. At the end of type inference, unconstrained type variables get assigned by the system. Usually, this assignment does the right thing. However, at times, the most general type is not what is desired and the user must add type constraints to the relevant subterms. For tricky situations, the global variable `show_types` can be assigned. When this flag is set, the prettyprinters for terms and theorems will show how types have been assigned to subterms. If you do not want the system to assign type variables for you, the global variable `guessing_tyvars` can be set to `false`, in which case the existence of unassigned type variables at the end of type inference will raise an exception.

2.2.3 Expanded term grammar

There is some further syntax that is specially treated by the parser. The theory of pairs introduces the infix pairing operator `(,)` as well as the corresponding infix product `(#)` type operator. The theory of sets introduces notation for the empty set `{}` (or `EMPTY`), membership (the infix `IN`) insertion (the infix `INSERT`), set comprehension, enumerated sets, and many other defined constants. The theory of lists introduces the constants `NIL` (the surface syntax `[]` can be used) and `CONS`, as well as notation for enumerated lists. The theories of (Peano) numbers and strings introduce the constructors `0`, `SUC`, `"`, and `STRING`, as well as literals for numbers and strings. If the theory of restricted quantifiers is present, syntax is provided for constraining bound variables by predicates.

Thus, if the theories of pairs, sets, numbers, strings, lists, and restricted quantifiers are loaded, the HOL grammar is an extension of that in Table 2.2.

In the table, the varstruct (`vstr`) construct is used. A varstruct is (apparently) an arbitrarily nested tuple of variables, where each variable only occurs once. The translation of varstructs into the internal abstract syntax trees is complex, so we avoid the explanation (for this draft).

$$\begin{array}{l}
 \text{vstr} ::= \text{ident} : \text{hol_type} \\
 \quad | \text{ident} \\
 \quad | \text{vstr}, \text{vstr} \\
 \quad | (\text{vstr})
 \end{array}$$

Also, in the term grammar a `charseq` is just a finite sequence of characters.

$term ::=$	$term : hol_type$	(type constraint)
	$term\ term$	(application)
	CONS $term\ term$	(list builder)
	INSERT $term\ term$	(set builder)
	SUC $term$	(successor)
	$\sim term$	(negation)
	$term = term$	(equality)
	$term ==> term$	(implication)
	$term \ \backslash / \ term$	(disjunction)
	$term \ \wedge \ term$	(conjunction)
	$term < term$	(less-than)
	$term + term$	(addition)
	$term * term$	(multiplication)
	$term - term$	(subtraction)
	$term \Rightarrow term \mid term$	(conditional)
	$\backslash vstr_1 \dots vstr_n [:: term]. term$	(lambda abstraction)
	$! vstr_1 \dots vstr_n [:: term]. term$	(forall)
	$? vstr_1 \dots vstr_n [:: term]. term$	(exists)
	$@ vstr_1 \dots vstr_n [:: term]. term$	(choose)
	$?! vstr_1 \dots vstr_n [:: term]. term$	(exists-unique)
	let $vstr = term$	
	[and $vstr = term$]* in $term$	(let expression)
	T	(truth)
	F	(falsity)
	0	(zero)
	[]	(empty list)
	{}	(empty set)
	$(term, term)$	(pair)
	$ident$	(constant or variable)
	$numeral$	(numeric literal)
	$"charseq"$	(string literal)
	$(term)$	(parenthesized term)
	$[term; \dots ; term]$	(enumerated list)
	$\{term; \dots ; term\}$	(enumerated set)
	$\{term \mid term\}$	(set comprehension)

Table 2.2: Expanded Term Grammar

Proof

Hol98 provides various mechanisms for doing proof. The user can invoke proof steps at very low levels of abstraction (something like doing assembly programming) or use sophisticated proof procedures that may perform tens or hundreds of thousands of inference steps in a single invocation. We give an overview of the different means by which proof can be performed in Hol98. First, the various kinds of proof procedures provided will be covered. Then we discuss the available definition principles of the system, and finally, we go on to describe the standard Hol98 proof manager.

3.1 Rules of inference

Hol98 follows the LCF tradition of implementing the primitive inference rules of the HOL logic as constructors for an ML abstract type `thm`. *Derived* rules are then built by arbitrary ML programming. In such a design, the only way that a theorem can result is when an ML function having range type `thm` is fully applied to its arguments. In an LCF-style system, therefore, there is no way for the system to produce a theorem other than by eventually invoking a primitive rule of inference. Put another way: if one is able to get an ML entity of type `thm`, then it has been proved via an unbroken chain of inference.

In the following subsections, we examine some packages that have been built upon the `thm` type. In each of these, it is important to remember that they are “merely” ways of organizing proofs, i.e., applications of primitive rules. However, first we treat something a little more radical.

3.2 Oracles

Hol98 extends the LCF tradition by allowing the use of an *oracle* mechanism to allow arbitrary formulas to become elements of the `thm` type. Thus Hol98 can utilize arbitrary proof procedures.¹ In spite of such liberalness, the system can still make strong assertions about the security of ML objects of type `thm`.

To avoid unsoundness, the system ensures that a tag is attached to any theorem coming from an oracle. The system propagates this tag through every inference that the theorem participates in.² If it happens that falsity becomes derived, the offending oracle can be

¹Some care should be taken with proof procedures for theories lacking a formalization in the HOL logic. The product of such procedures can be used in HOL without damage; however, in the end it won't be clear — at least in HOL — what has been proven.

²The idea is due to Mike Gordon.

found by examining the tags component of the theorem. (The Hol98 authors would be quite interested to hear of cases where falsity is derivable without the use of oracles.) A theorem proved without use of any oracle will have an empty tag, and can be considered to have been proved in the HOL logic.

Tagged theorems can be created via

```
val mk_oracle_thm : tag -> term list * term -> thm
```

which directly creates the requested theorem and attaches the given tag to it. Tags may be created with

```
Tag.read : string -> tag.
```

As well as providing principled access to external reasoners, tags are used to implement some useful ‘system’ operations on theorems. For example, Hol98 allows one to directly create a theorem via `mk_thm`. The tag `MK_THM` gets attached to each theorem created with this call. This allows users to directly create useful theorems, e.g., to use as test data for derived rules of inference. Another tag is used to implement validity checking in tactics. Other common pre-existing tags are for “definition schemas” like `num_CONV` (which encapsulates the semantics of numeric literals) and `string_CONV` (which performs the analogous function for strings).

The tags in a theorem can be viewed by setting `Globals.show_tags` to true. For example, we have³

```
- mk_thm([], Term 'F');

val it = [oracles: MK_THM] [axioms: ] [] |- F : thm
```

There are three elements to the left of the turnstile in the printed representation of a theorem: the first two comprise the tags component and the third is the standard assumption list. The tag component of a theorem can be extracted by

```
Thm.tag : thm -> tag
```

and prettyprinted by

```
Tag.pp : ppstream -> tag -> unit.
```

Remark

No serious attempt is made to prevent spoofing: a person may slap tag `X` on an assertion coming from tool `Y` if desired. However, the tags for `mk_thm` and validity checking cannot be spoofed.

³Hol98 currently also uses tags for tracking the use of axioms in proofs.

3.3 Tactics

Tactics are a well-known method for backward proof. The original conception of Robin Milner, which is still that of tactics in HOL, is that a tactic can be represented by the type

$$goal \longrightarrow goal\ list * justification,$$

i.e., a tactic decomposes a goal⁴ into subgoals plus a justification function. The justification function takes the theorems resulting from the solved subgoals and performs inference with them to return a new theorem that *achieves* the original goal. Thus the justification has type

$$thm\ list \longrightarrow thm.$$

A theorem $\Gamma \vdash M$ achieves a goal (Δ, N) when $M =_{\alpha} N$ and also each element of Γ is equal, modulo α convertibility, to an element of Δ . A tactic t *solves* a goal g when $t\ g$ creates an empty list of subgoals and a justification function f such that f applied to the empty list achieves g .

For example, a simple tactic is `CONJ_TAC`. It takes a goal and, if it is a conjunction, splits it into two subgoals. The justification function is `CONJ`, which takes two theorems and returns a new theorem, which has as assumptions the union of the assumptions of the two theorems, and has as the conclusion the conjunction of the conclusions of the two theorems. In ML code this is expressed as:

```
fun CONJ_TAC (as1,c) =
  let val {conj1,conj2} = dest_conj c
  in
    ([ (as1,conj1), (as1,conj2) ], fn [th1,th2] => CONJ th1 th2)
  end
  handle HOL_ERR _ => raise TACTIC_ERR "CONJ_TAC" "";
```

Tactics are composed via tacticals. The basic tacticals are `THEN`, `THENL`, `ORELSE`, and `REPEAT`. The workhorse tacticals can be found in Table 3.1.

3.4 Conversions

Another heavily used method of arranging proofs in Hol98 is *conversions*, which were created by Larry Paulson. In the area of equational reasoning, they provide a high-level language similar to the language of tactics and tacticals. The rewriters and simplifiers of Hol98 are all implemented using conversions. Historically, conversions have also been heavily used in interactive proof; however, the passage of time has seen the emergence of advanced forms of rewriting—such as conditional and contextual rewriting, as well as rewriting using (restricted) higher order matching—which often provides a more convenient alternative for interactive use.

⁴A goal (A, c) has ML type *term list * term*.

<i>tactic</i> ::=	<i>primitactic</i>	(basic tactic)
	NO_TAC	(fail)
	ALL_TAC	(No-op)
	<i>tactic</i> THEN <i>tactic</i>	(composition)
	<i>tactic</i> ORELSE <i>tactic</i>	(alternative)
	<i>tactic</i> THENL [<i>tactic</i> , ..., <i>tactic</i>]	(indexed composition)
	REPEAT <i>tactic</i>	(iteration)

Table 3.1: Tactics and Tacticals

3.5 Theorem Continuations

Another invention of Paulson were *theorem continuations*. These provide support for interactively building tactics that perform very specific manipulation of theorems in the course of inference. Experienced HOL users often swear by theorem continuations, but we will not document them here; theorem continuations are something that one should pick up after learning higher-level proof methods.

3.6 Simplifiers and Automatic Reasoners

Hol98 comes with several simplifiers. Largely that is a tribute to the ease with which simplifiers can be written with conversions. Each of the following items is the name of the ML structure containing the simplifier(s).

Rewrite Performs unconditional rewriting, with various strategies. It uses only first order matching (up to alpha-convertibility). This has been a workhorse proof tool over the years.

Ho_rewrite Performs unconditional rewriting, with various strategies. It employs restricted higher order matching, which is a significant increase in power, since it performs such things as quantifier movement, which was formerly done in a tedious fashion by conversions.

RW Performs conditional and contextual rewriting, with various strategies. It employs first order matching up to alpha conversion. This is again a significant increase in power over unconditional rewriting, since it can automatically apply implicational theorems by instantiating and solving the antecedents while traversing the goal. This package was originally designed to implement termination condition extraction when using `tfllib` to define recursive functions.

Cond_rewrite Performs conditional rewriting with first order matching, and a fixed strategy. This is used to provide support for the `res_quan` library, but is also generally useful.

simpLib Performs conditional and contextual rewriting, with a fixed top-down strategy. It employs higher order matching and also performs ordered rewriting for associative-commutative operators, as in the Boyer-Moore system. This is an ‘Isabelle-style’ simplifier, written by Don Syme; he has significantly extended it past the original Isabelle design, by allowing free application of decision procedures throughout the rewriting process.

The simplifier library provides a range of pre-assembled databases with which to work. We list the structures that must be loaded for each of them.

<code>empty_ss</code>	The empty simplification set
<code>boolSimps</code>	Standard logic simplifications
<code>combinSimps</code>	Combinator rewrites
<code>pairSimps</code>	Rewrite rules for pairs
<code>sumSimps</code>	Rewrite rules for sums
<code>listSimps</code>	Basic list theory rewrites
<code>ListSimps</code>	Extended list theory simplifications
<code>arithSimps</code>	Arithmetic simplification, using linear arithmetic proc.
<code>SatisfySimps</code>	Conversions for witness instantiation
<code>UnwindSimps</code>	Unwinding existentials
<code>HOLSimps</code>	All of the above

By default, only `empty_ss` is available when `simpLib` is loaded.

Although it might seem like there ought to be only one simplification tool supported in Hol98, each of the above tools has found a niche in the system, and the differences among them have been significant enough to deter consideration of their unification. It is therefore up to the user to decide which is most suitable for the purpose at hand.

3.6.1 Automatic methods

Hol98 also provides automatic reasoners for several domains.

- For general first order reasoning, there is `mesonLib`, which has been introduced already.
- For arithmetic, tautologies, pairs, datatypes, and ground equational reasoning, there is `decisionLib`, which implements the Nelson-Oppen method for combining decision procedures.
- For in-the-logic calculations involving numbers or booleans, there is `reduceLib`. This has been incorporated into `decisionLib`, but the reasoners in `reduceLib` can be helpful when writing custom proof tools.
- For the theory of lists, there are some useful conversions in `listLib`.

3.7 Definition principles

One of the main thrusts in the development of the HOL system has been a stubborn insistence on building formalizations by principles of definition, as opposed to the assertion of axioms. Users of definition principles have the knowledge that they have introduced no inconsistency into the system. Such peace of mind comes at a price, however, since the principles of definition for the HOL logic are extremely simple. To remedy this, high-level definition mechanisms have been built as derived rules of inference. The definition principles listed in Table 3.2 are currently offered by Hol98.

Type definition	Type_def	primitive
Recursive types	Define_type	
Mutually recursive types	mutrecLib	
Nested recursive types	nested_recLib	
Quotient types	EquivType	
Record types	RecordType	
Constant specification	Const_spec	primitive primitive
Constant definition	Const_def	
Primitive recursive functions	Prim_rec	
Mutually recursive functions	mutrecLib	
Inductively defined relations	ind_defLib, IndDefLib	
Wellfounded recursive functions	tflLib	

Table 3.2: Definition Principles

Unfortunately, each one of these facilities has separate entrypoint(s), and moreover, using some of them can be quite ungainly. For example, to build and use recursive type typically involves several steps: first the type has to be defined, then separate function calls need to be made to build induction theorems, standard rewrite rule sets, ‘case’ theorems, etc. Moreover, during proof, the names of all these entities must be remembered, and managed. A prototype library aimed at remedying this problem can be found in Chapter 6.

3.7.1 Record types

Record types ⁵ are convenient ways of bundling together a number of component types, and giving those components names so as to facilitate access to them. Record types are semantically equivalent to big pair (cross-product) types, but the ability to label the fields with names of one’s own choosing is a great convenience. Record types as implemented in HOL98 are similar to C’s `struct` types and to Pascal’s records. However, the current HOL implementation doesn’t allow the equivalent of variant records, nor for records to be recursive.

⁵This documentation has been supplied by Michael Norrish

Done correctly, record types provide useful maintainability features. If one can always access the `fieldn` field of a record type by simply writing `fieldn record`, then changes to the type that result in the addition or deletion of other fields, will not invalidate this reference. One failing in SML's record types is that they do not allow the same maintainability as far as (functional) updates of records are concerned. The HOL implementation allows one to write `fieldn_update new_value rec`, which replaces the old value of `fieldn` in the record `rec` with `new_value`. This expression will not need to be changed if another field is added, modified or deleted from the record's original definition.

3.7.1.1 Defining a record type in Hol98

The record type package is defined in the structure `RecordType`. Defining a record type is achieved with the function `create_record`, which is in that structure. This takes two parameters, a string which is the name of the new type, and a list of string-type pairs, which are the names and types of the record type's fields. For example, to create a record type called `person` with boolean, string and number fields called `employed`, `name` and `age`, one would enter:

```
val person_result =
  create_record "person" [("employed", ``:bool``),
                          ("age",      ``:num``),
                          ("name",    ``:string``)];
```

The order in which the fields are entered is not significant. As well as defining the type (called `person`), the `create_record` function also defines three other sets of constants. These are the field access functions, update functions, and functional update functions. The access functions are given the same name as the fields chosen,⁶ so that one would use the expression: `(employed bob)` in order to return the value of bob's `employed` field.

The update functions are given the names `field_update` for each field in the type. They take a value of the type of the field in question and a record value to be modified. They return a new record value that is otherwise the same as the old value but with the specified field having the new value. Having the record value as the second parameter means that chains of updates are easy to write, thus:

```
employed_update T
  (age_update 10
   (name_update "Child_labourer" bob))
```

The functional update functions have the names `field_fupd`. Rather than specifying a new value for the record, these functions take a function as their first parameter, which will be an endomorphism on the field type, so that the resulting record is the same as the original, except that the specified field has had the given function applied to it to generate the new value for that field. The functional update functions allow more concision when writing updates on a record that depend on the field's old value.

⁶Note that this means that a field name can not be re-used from one record type to another, as there can only ever be one constant of a given name.

3.7.1.2 Specifying record literals

In the absence of any dedicated parsing support for record values (which may change in future releases), there are two ways of specifying record values directly. The representing type for records is constructed using the standard datatype package to define a type with one constructor that takes arguments corresponding to the fields. This means that one can specify literal values by remembering the order of types as given in the original definition, and using the constructor, which has the same name as the type. Thus, one might write:

```
(person T 10 "Child_labourer")
```

This does not win many prizes for maintainability. Marginally better is an ML function provided in the `RecordType` structure, `create_term_fn`. This takes a string specifying the name of the type, the accessors theorem for the type (see below), and a list of string-value pairs. Thus, one might write:

```
val t = create_term_fn "person" (#accessor_fns person_result)
      [("age", ``10``), ("employed", ``T``),
       ("name", ``Child_labourer``)];
```

If a field is omitted in this specification, then the value given to that field in the final record value is `ARB`.

3.7.1.3 Using the theorems produced by `create_record`

As well as defining the type and the functions described above, record type definition, also proves a suite of useful theorems. Most of these are returned in a big record; all are stored using `save_thm` so that they can be recovered.

The record returned has the following fields:

`type_axiom` The type axiom for the record type, as returned by the standard datatype definition package.

`accessor_fns` The definitions of the accessor functions. This theorem should be included in rewrites used for this type.

`update_fns` The definitions of the update functions. This theorem should be included in rewrites used for this type.

`cases_thm` The usual cases theorem for a type, stating that for all record values, there exist component values making it up.

`fn_upd_thm` The definitions of the functional update functions. This theorem should be included in rewrites used for this type.

`acc_upd_thm` A theorem stating simpler forms for expressions of the form $field_i (field_j_update\ v\ r)$. If $i = j$, then the RHS is v , if not, it is $(field_i\ r)$. This theorem should be included in rewrites used for this type.

`upd_acc_thm` A theorem stating that `fieldi_update (fieldi r) r = r` for all of the fields defined in the type. This theorem should be included in rewrites used for this type.

`upd_upd_thm` A theorem stating that `fieldi_update v1 (fieldi_update v2 r) = fieldi_update v1 r`. This theorem should be included in rewrites used for this type.

`upd_canon_thm` A theorem that states commutativity results for all possible pairs of field updates. They are constructed in such a way that if used as rewrites, they will canonicalise sequences of updates. This theorem should be included in rewrites used for this type.

`cons_11_thm` The standard result stating the type constructor is injective. This theorem should be included in rewrites used for this type.

`create_term` This last component of the record returned is not a theorem, but rather an ML function. It is identical to the `create_term_fn` already defined in `RecordType`, but is pre-applied to the relevant arguments, so that it is of the type `string-value list to term`.

3.7.1.4 To do

- Parsing and pretty-printing support would be nice.
- Need to have the package automatically prove that equality of records is equivalent to equality of all the fields.

3.8 A simple proof manager

The *goal stack* provides a simple interface to tactic-based proof. When one uses tactics to decompose a proof, many intermediate states arise; the goalstack takes care of the necessary bookkeeping. The implementation of goalstacks reported here is a re-design of Larry Paulson's original conception.

The abstract types *goalstack* and *proofs* are the focus of backwards proof operations. The type `proofs` can be regarded as a list of independent goalstacks. Most operations act on the head of the list of goalstacks; there are operations so that the focus can be changed.

3.8.1 Starting a goalstack proof

```
g          : term quotation -> proofs
set_goal  : goal -> proofs
```

Recall that the type `goal` is an abbreviation for `term list * term`. To start on a new goal, one gives `set_goal` a goal. This creates a new goalstack and makes it the focus of further operations.

A shorthand for `set_goal` is the function `g`: it invokes the parser automatically, and it doesn't allow the the goal to have any assumptions.

Calling `set_goal`, or `g`, adds a new proof attempt to the existing ones, *i.e.*, rather than overwriting the current proof attempt, the new attempt is stacked on top.

3.8.2 Applying a tactic to a goal

```
expandf : tactic -> goalstack
expand  : tactic -> goalstack
e       : tactic -> goalstack
```

How does one actually do a goalstack proof then? In most cases, the application of tactics to the current goal is done with the function `expand`. In the rare case that one wants to apply an *invalid* tactic, then `expandf` is used. (For an explanation of invalid tactics, see Chapter 24 of Gordon & Melham.) The abbreviation `e` may also be used to expand a tactic.

3.8.3 Undo

```
b          : unit -> goalstack
drop       : unit -> proofs
dropn     : int  -> proofs
backup    : unit -> goalstack
restart   : unit -> goalstack
set_backup : int  -> unit
```

Often (we are tempted to say *usually!*) one takes a wrong path in doing a proof, or makes a mistake when setting a goal. To undo a step in the goalstack, the function `backup` and its abbreviation `b` are used. This will restore the goalstack to its previous state.

To directly back up all the way to the original goal, the function `restart` may be used. Obviously, it is also important to get rid of proof attempts that are wrong; for that there is `drop`, which gets rid of the current proof attempt, and `dropn`, which eliminates the top n proof attempts.

Each proof attempt has its own *undo-list* of previous states. The undo-list for each attempt is of fixed size (initially 12). If you wish to set this value for the current proof attempt, the function `set_backup` can be used. If the size of the backup list is set to be smaller than it currently is, the undo list will be immediately truncated. You can not undo a "proofs-level" operation, such as `set_goal` or `drop`.

3.8.4 Viewing the state of the proof manager

```
p          : unit -> goalstack
status     : unit -> proofs
top_goal   : unit -> goal
top_goals  : unit -> goal list
```

```
initial_goal : unit -> goal
top_thm      : unit -> thm
```

To view the state of the proof manager at any time, the functions `p` and `status` can be used. The former only shows the top subgoals in the current goalstack, while the second gives a summary of every proof attempt.

To get the top goal or goals of a proof attempt, use `top_goal` and `top_goals`. To get the original goal of a proof attempt, use `initial_goal`.

Once a theorem has been proved, the goalstack that was used to derive it still exists (including its undo-list): its main job now is to hold the theorem. This theorem can be retrieved with `top_thm`.

3.8.5 Switch focus to a different subgoal or proof attempt

```
r           : int -> goalstack
R           : int -> proofs
rotate      : int -> goalstack
rotate_proofs : int -> proofs
```

Often we want to switch our attention to a different goal in the current proof, or a different proof. The functions that do this are `rotate` and `rotate_proofs`, respectively. The abbreviations `r` and `R` are simpler to type in.

Existing Context

4.1 Theories

In Hol98, theories are represented by separately compiled ML structures. The theories listed in Table 4.1 come pre-built in the system (we have omitted a few of the less commonly used ones).

minTheory	the origin theory
boolTheory	definitions of logical operators and basic axioms
pairTheory	basic theory of pairs
numTheory	Peano's axioms derived from the axiom of infinity
prim_recTheory,	the primitive recursion theorem
arithmeticTheory	Peano arithmetic development
integerTheory	integers, by John Harrison
TCTheory	transitive closure of a relation
primWFTTheory	wellfounded relations, plus WF induction and recursion
WFTTheory	instances of wellfoundedness at various types
setTheory	sets as a separate type (includes finite sets)
pred_setTheory	sets as predicates (includes finite sets)
listTheory	lists
ListTheory	extended theory of lists
optionTheory	the "option" type
finite_mapTheory	finite maps from α to β
ltreeTheory	polymorphic finitely branching trees
combinTheory	combinators
sumTheory	disjoint sums
restr_binderTheory	definitions of binder restrictions
res_quanTheory	restricted quantifier support
asciiTheory	ascii
stringTheory	strings
wordTheory	(<i>plus others</i>) theory of bitstrings
realTheory	(<i>plus others</i>) real numbers and analysis
HOLTheory	equivalent to HOL theory from hol88/90

Table 4.1: Native Theories

The only theory that is initially loaded by an invocation of Hol98 is `boolTheory`. To gain access to any other theory when working interactively, simply invoke

```
load "xTheory";
```

where `x` is the name of the theory. Once the theory has been loaded by the system, access to its contents is through the “dot” notation of ML, e.g., `xTheory.FOO_DEF`, or if you prefer, by “opening” the structure and then directly accessing its contents.

We will consider the construction of theories in Chapter 5.

4.2 Libraries

Hol98 currently offers the libraries found in Table 4.2.

As for theories, a library is represented by a separately compiled ML structure and can thus be brought into an interactive session by invoking “load”, e.g.,

```
load "xLib";
```

One difference between libraries and theories is that a library can in general consist of a collection of theories and support ML structures. Thus sometimes, but not always, the functionality of a library is distributed through a collection of ML structures, all of which have been brought into the interactive session by the call to “load”. It (unfortunately) falls to the user to know about the functionality of a library. Some libraries provide “help” and manuals for their use; others do not.

decisionLib	cooperating decision procedures
mesonLib	model- elimination first order reasoner
simpLib	Isabelle-style simplifier
ind_defLib	inductive defn. package
IndDefLib	generalized inductive defn. package
tflLib	wellfounded recursive definitions
mutrecLib,	mutually recursive datatype definitions
nestrecLib	nested recursive datatype definitions
mutualLib	improved mutual/nested datatypes
goalstackLib	simple manager for building tactic proofs
basicHol90Lib	derived rules, tactics, conversions, rewriting, etc.
optionLib	option type
pairLib	extended support for pairs
setLib	sets as a separate type
pred_setLib	sets as predicates
listLib	extensive development of lists
stringLib	characters and strings
wordLib	theories and proof support for bitstrings
unwindLib	unwinding existential quantifiers
res_quanLib	bounded quantification
hol88Lib	support for hol88 compatibility
liteLib	support for portability with HOL-Lite
ho_matchLib	higher-order versions of various proof tools
refuteLib	support for refutation procedures
reduceLib	basic reasoners for nums and bools
tautLib	tautology prover
arithLib	linear arith. decision procedures
BoyerMooreLib	automatic proof procedure based on the one in Nqthm
bossLib	collection of automatic tools
realLib	theories for the real numbers and analysis.
robddLib	Reduced Ordered Binary Decision Diagrams

Table 4.2: Native Libraries

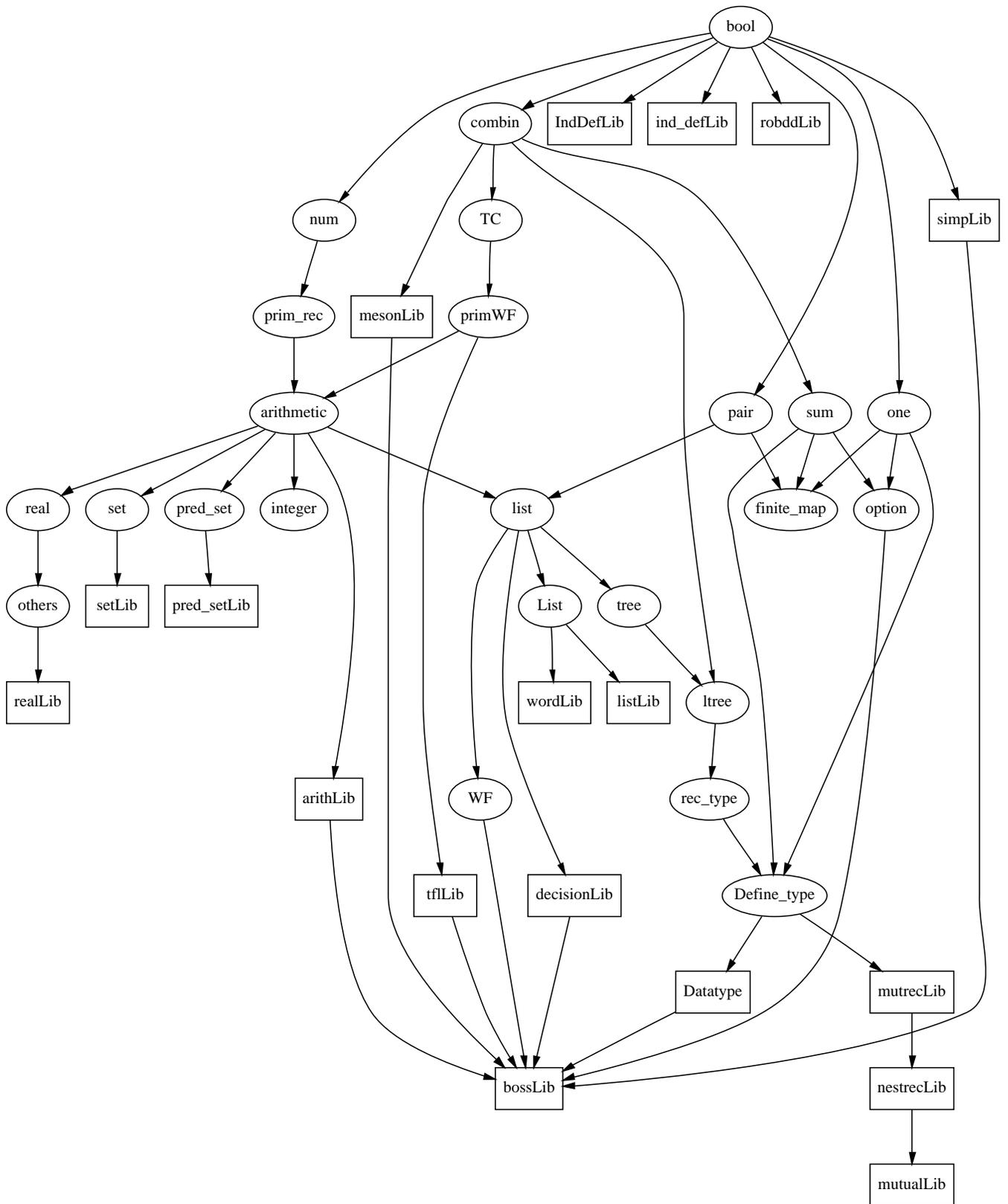


Figure 4.1: System dependencies

Building Logical Developments

This chapter deals with theories and `Holmake`. Theories are a simple structuring mechanism, with which formalizations can be broken into chunks. `Holmake` is a tool for handling all the ML code and theories associated with a large formalization.

5.1 Theories

Building theories can be considered to be the main point of work in `Hol98`. A *theory* is a related collection of types, constants, axioms, definitions, and theorems, plus ‘pointers’ to ancestor theories. In `Hol98`, theories are built in an interactive manner: axioms, definitions and theorems can be entered into, or deleted from, the theory under construction at any time. The system maintains the required dependencies so that inconsistency cannot result. Once the user has built the theory to his or her satisfaction, it can be exported to disk, to be reloaded at a later date, without having to replay the proofs that created the theory in the first place. This is known as persistence.

In `Hol98`, there is always a single current theory. When `Hol98` starts up, the current theory is called `scratch`. Its only parent is the theory `bool`. Every theory except for the initial theory (named `min` in `Hol98`) has one or more parent theories (found by calling `parents`). The transitive closure of the parent relation provides the ancestry of a theory and is computed by the `ancestry` function.

5.1.1 Building a theory

One makes a theory by a call to `new_theory`. This allocates a new ‘scratch area’ where subsequent theory operations take effect. Elements stored in this area may be overwritten by subsequent additions, or deleted outright. Any theory elements that were built on those elements are now held to be *out-of-date*, and will not be included in the theory when it is exported to disk. Moreover, out-of-date constants and types are printed surrounded by odd-looking syntax to alert the user.

```
new_theory    : string -> unit

new_type      : {Name : string, Arity : int} -> unit
new_constant  : {Name : string, Ty : hol_type} -> unit
new_infix     : {Name : string, Ty : hol_type, Prec : int} -> unit
new_binder    : {Name : string, Ty : hol_type} -> unit
```

```

set_fixity    : string -> fixity -> unit

new_axiom     : string * term -> thm
save_thm      : string * thm -> thm
store_thm     : string * term * tactic -> thm

```

The `new_` functions listed above add types and constants to the current HOL signature. These entrypoints are rarely invoked directly by the user; instead, definition principles are typically used to update the logic signature. The function `set_fixity` can be used to change the parsing status of a constant after it has been declared. Notice that there is no entrypoint for adding a new parent; that is because a theory can be added as a parent to the current theory simply by loading it. The entire ancestry of the new parent is then recursively and silently loaded.

Usually, when `new_theory` is called, the contents of the current theory are written to disk before the new theory is constructed. That is, unless the theory is already consistent with disk (this is kept track of internally). If however, `new_theory` is invoked with the same name as the current theory, it is assumed that the user wishes to clear the scratchpad and start over, so internally that is what is done. This allows the current theory to be repeatedly re-loaded without having to restart the ML session.

The functions `new_axioms` and `save_thm` add axioms and theorems into the current theory, under the given name. This name should be an acceptable ML identifier. The function `store_thm` takes a term and a tactic and treats the term as a goal to apply the tactic to. The proved theorem is stored in the current theory under the given name.

```

delete_type   : string -> unit
delete_const  : string -> unit
delete_axiom  : string -> unit
delete_theorem : string -> unit
uptodate_type : hol_type -> bool
uptodate_term : term -> bool
uptodate_thm  : thm -> bool
scrub         : unit -> unit

```

There are also some functions that allow any element of a theory to be deleted. Any other elements that depend on that element become out-of-date. There is also a complementary suite of functions that tell whether an item is current with respect to the current theory.

Finally, the following situation can often occur: an item held in the current theory all of a sudden becomes out-of-date because something it was built from got deleted. For reasons of efficiency, it is undesirable for the system to re-check all dependencies after every modification to the theory; thus the function `scrub` is made available so that the user can decide when to clean up a theory in disarray.

5.1.2 Information functions

The following functions can be used to find out information about theory items. Most of them are fairly self-explanatory and so we won't cover them.

```

arity      : string -> int      (* of a type constant *)
fixity     : string -> fixity   (* of a term constant *)
precedence : string -> int

is_type    : string -> bool
is_constant : string -> bool
is_binder  : string -> bool
is_infix   : string -> bool

parents    : string -> string list (* of a theory *)
ancestry   : string -> string list

(* items from given theory *)
types      : string -> {Name : string, Arity : int} list
constants  : string -> term list
infixes    : string -> term list
binders    : string -> term list

(* named theorem from current theory *)
axiom      : string -> thm
definition : string -> thm
theorem    : string -> thm

(* all items in a certain class from current theory only *)
axioms     : unit -> (string * thm) list
theorems   : unit -> (string * thm) list
definitions : unit -> (string * thm) list

print_theory : unit -> unit      (* The whole picture *)
current_theory : unit -> string

```

The function `print_theory` can be used to give a listing of the types, constants, axioms, definitions, and theorems of the current theory.

5.1.3 Theories and the file system

Hol98 provides persistent theories: once a user has finished working on a theory, it can be written out to a file, to be used in future formalization efforts. Before the current theory is written out, all out-of-date entities are scrubbed out. Also the parenthood of the theory is computed: it is the fringe of the theory graph at the time `export_theory` is called.

```
export_theory : unit -> unit
```

There is also a more general export operation—`prim_export_theory` — which invokes user-supplied prettyprinters just before printing the end of the theory signature and structure. This can be used to, for example, store theory-specific proof tools with the theory.

When `export_theory` is invoked from an interactive session, the theory is exported, but not compiled. This makes it difficult to load the theory in a later session. Currently, `export_theory` is only useful when invoked under the control of `Holmake`.

5.2 Holmake

The purpose of `Holmake`¹ is to maintain dependencies in a Hol98 source directory. A single invocation of `Holmake` will compute dependencies between files, (re-)compile plain ML code, (re-)compile and execute theory scripts, and (re-)compile the resulting theory modules. `Holmake` does not require the user to provide any dependency information, e.g., a Makefile. `Holmake` can be very convenient to use, but there are some conventions and restrictions on it that must be followed, which we will describe in the sequel.

`Holmake` can be accessed through

```
<hol-dir>/bin/Holmake.
```

The model of user development that `Holmake` is designed to support is that there are two modes of work: theory construction and system revision. In ‘theory construction’ mode, the user builds up a theory by interacting with HOL, perhaps over many sessions. In ‘system rebuild’ mode, a module that others depend on has been altered, so all intervening modules have to be brought up to date. System re-build mode is simpler so we deal with it first.

5.2.1 System Rebuild

A system rebuild happens when an existing theory has been improved in some way (augmented with a new theorem, a change to a definition, etc.), or perhaps some support ML code has been added. The user needs to find and recompile just those modules affected by the change. This is what an invocation of `Holmake` does, by identifying the out-of-date modules and re-compiling and re-executing them.

5.2.2 Theory construction

To start a theory construction, some context (semantic, and also proof support) is established, typically by loading parent theories and useful libraries. In the course of building the theory, the user keeps track of the ML — which, for example, establishes context, makes definitions, builds and invokes tactics, and saves theorems — in a text file. This file is used to achieve inter-session persistence of the theory being constructed, i.e., the text

¹`Holmake` has been written by Ken Larsen.

file resulting from session n is “use”d to start session $n + 1$; after that, theory construction resumes.

Once the user finishes the perhaps long and arduous task of constructing a theory, the user should

1. make the script separately compilable;
2. invoke `Holmake`. This will (a) compile and execute the script file; and (b) compile the resulting theory file. After this, the theory file is available for use.

5.2.3 Making the script separately compilable

First, the invocation

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

should be added at the end of the file. When the script is finally executed, this call writes the theory to disk.

Second, we address a crucial environmental issue: if a theory script has been constructed using `<holdir>/bin/hol`, then it has been developed in an environment where some commonly used structures, e.g., `Tactic`, have already been loaded and opened for the user’s convenience. When we wish to apply `Holmake` to a script developed in this way, we have to take some extra steps to ensure that the compilation environment also provides these structures. In the common case, this is simple; one must only add, at the head of the theory script, the following “boilerplate”:

```
open HolKernel Parse basicHol90Lib;
infix THEN THENL THENC ORELSE ORELSEC THEN_TCL ORELSE_TCL ## |->;
infixr -->;
```

This will duplicate the starting environment that one obtains with `<holdir>/bin/hol` and `<holdir>/bin/hol.enquote`.

Now the script should be separately compilable. Invoke `Holmake` to check; `MoscowML` will flag any unaccounted-for identifiers it finds. The user has to resolve these, either by using the ‘dot’ notation to locate the identifier for the compiler, or by opening the relevant module. This “compile/resolve-identifier” loop should continue until `Holmake` succeeds in compiling the module.

The following notes may be of some help.

1. The filenames of theory scripts must follow the following convention: a HOL theory script for theory “x” should be named

```
xScript.sml.
```

If there is a corresponding signature (and there needn’t be), it should – following `Moscow ML` convention – be named `xScript.sig`. When `export_theory` is called during an invocation of `Holmake`, the files

```
xTheory.{sig,sml}
```

will be generated and then compiled.

2. In the MoscowML batch compiler, modules are not allowed to have unbound top-level expressions. Hence, something like the following is not allowed:

```
new_theory"ted";
```

To make Moscow ML happy, one must instead write something like

```
val _ = new_theory"ted";
```

3. In the interactive system, one has to explicitly `load` modules; on the other hand, the batch compiler will load modules automatically. For example, in order to execute `open Foo` (or refer to values in `Foo`) in the interactive system, one must first have executed `load "Foo"`. Contrarily, the batch compiler will reject files having occurrences of `load`, since `load` is only defined for the interactive system.
4. Take care not to have the string "Theory" embedded in the name of any of your files. Hol98 generates files containing this string, and when it cleans up after itself, it removes such files using a regular expression. This will also remove other files with names containing "Theory". For example, if, in your development directory, you had a file of ML code named

```
MyTheory.sml
```

and you also were managing a Hol98 development there with `Holmake`, then `MyTheory.sml` would get deleted if `Holmake clean` was invoked.

5. The dependency analysis of `Holmake` will generate dependencies for *all* files ending in ".sig" and ".sml" in the directory where `Holmake` is invoked. If you do not want a particular file to be compiled by `Holmake`, for example, a script for a theory under construction, simply omit the suffix, or invent a different one.
6. We can see that some users may not wish to use (some of) the support provided by `basicHol90Lib`, since it is becoming dated. In that case, the same general principle set out above will apply: the user must ensure that the compilation environment for a theory script is the same as the interactive environment it was developed in.
7. Currently, dependencies can get computed several times. This is not harmful, but it should eventually get fixed.

5.2.4 Summary

A complete theory construction is performed by the following steps:

- Construct theory script, perhaps over many sessions;
- Transform script into separately compilable form;
- Invoke `Holmake` to generate the theory and compile it.

After that, the theory is usable as a module in MoscowML.

5.2.5 What Holmake doesn't do

`Holmake` only works properly on the current directory. If there is an out-of-date dependence leading up out of the current directory, `Holmake` will follow it and attempt to build whatever has to be built. This can fail, and it is not the right way to use `Holmake`. If one is developing a system over more than one directory, one should write a master Makefile (or shell script) that invokes `Holmake` in the subsidiary directories, in the correct order, i.e., such that there never is an out-of-date dependence leading outside of the current directory. This should always be achievable, simply by ordering the directories in the order that one would have to “use” files in them. See the Makefile in the top level of the `Hol98` distribution for a large example.

5.2.6 Options to Holmake

There are several other options to `Holmake`:

clean Removes all compiled files.

cleanAll Removes all compiled files as well as all of the hidden dependency information.

Finally, the user can directly affect the workings of `Holmake` by writing a file named `INCLUDE.mk` to the directory where `Holmake` is to be invoked. This file is typically used to extend the load path used by the MoscowML batch compiler. The following example comprises the entire contents of the file `INCLUDE.mk`, and allows already compiled code/theories to be found by the batch system at the given paths:

```
INCLUDE = -I /some/full/path \  
         -I /some/other/path
```

High-level interactive proof support

The library `bossLib` marshalls some of the most widely used theorem proving tools in HOL and provides them with a convenient interface for interaction. The library currently focuses on two things: definition of datatypes and functions; and composition of automated reasoners. Loading `bossLib` commits one to working in a context that already supplies the theories of booleans, pairs, the option type, arithmetic, and lists.

6.1 Datatype definition

There are several useful consequences of an object logic datatype definition: structural induction, rewrite rules for constructors, etc. However, these have not traditionally been automatically derived at the invocation of the definition package: the user would have to build the required theorems by explicitly invoking various proof procedures. To remedy this, `bossLib` offers the `Hol_datatype` function.¹ The syntax of declarations that `Hol_datatype` accepts is found in Table 6.1.

$\text{Hol_datatype } \langle \textit{ident} = [\textit{clause }]^* \textit{clause} \rangle$ $\textit{clause} ::= \textit{ident}$ $\quad \quad \textit{ident} \text{ of } [\textit{hol_type} \Rightarrow]^* \textit{hol_type}$

Table 6.1: Datatype Declaration

There is an underlying database of datatype facts that supports the activities of `bossLib`. This database already contains the relevant entries for the types `bool`, `prod`, `num`, `option`, and `list`. When a datatype is defined by `Hol_datatype`, the following information is derived and stored in the database.

- initiality theorem for the type
- injectivity of the constructors
- distinctness of the constructors
- structural induction theorem

¹`bossLib` does not yet support mutually recursive or nested datatypes.

- case analysis theorem
- definition of the ‘case’ constant for the type
- congruence theorem for the case constant
- definition of the ‘size’ of the type

The following functions use information in the database to ease the application of Hol98’s underlying functionality:

```

type_rws      : string -> thm list
Induct        : tactic
Cases         : tactic
Cases_on      : term quotation -> tactic
Induct_on     : term quotation -> tactic

```

- The function `type_rws` will search for the given type by name in the underlying database and return useful rewrite rules for that type. The rewrite rules of the datatype are built from the injectivity and distinctness theorems, along with the case constant definition. The pre-existing rewrite rules in the database are already integrated into the simplification sets provided by `bossLib`; however rewrite rules arising from an invocation of `Hol_datatype`, or which come from a user-defined theory, will have to be manually added into the simpsets used by the simplifier.
- The `Induct` tactic makes it convenient to invoke induction. When it is applied to a goal, the leading universal quantifier is examined; if its type is that of a known datatype, the appropriate structural induction tactic is extracted and applied.
- The `Cases` tactic makes it convenient to invoke case analysis. The leading universal quantifier in the goal is examined; if its type is that of a known datatype, the appropriate structural case analysis theorem is extracted and applied.
- The `Cases_on` tactic takes a quotation, which is parsed into a term M , and then M is searched for in the goal. If M is a variable, then a variable with the same name is searched for. Once the term to split over is known, its type and the associated facts are obtained from the underlying database and used to perform the case split. If some free variables of M are bound in the goal, an attempt is made to remove (universal) quantifiers so that the case split has force. Finally, M need not appear in the goal, although it should at least contain some free variables already appearing in the goal. Note that the `Cases_on` tactic is more general than `Cases`, but it does require an explicit term to be given.
- The `Induct_on` tactic takes a quotation, which is parsed into a term M , and then M is searched for in the goal. If M is a variable, then a variable with the same name is searched for. Once the term to induct on is known, its type and the associated facts are obtained from the underlying database and used to perform the induction.

If M is not a variable, a new variable v not already occurring in the goal is created, and used to build a term $v = M$ which the goal is made conditional on before the induction is performed. First however, all terms containing free variables from M are moved from the assumptions to the conclusion of the goal, and all free variables of M are universally quantified. `Induct_on` is more general than `Induct`, but it does require an explicit term to be given.

Two supplementary entrypoints have been provided for more exotic inductions:

`completeInduct_on` performs complete induction on the term denoted by the given quotation. Complete induction allows one to assume a (seemingly) stronger induction hypothesis than ordinary mathematical induction: to wit, when inducting on n , one is allowed to assume the property holds for *all* m smaller than n . Formally: $\forall P. (\forall x. (\forall y. y < x \supset P y) \supset P x) \supset \forall x. P x$. This allows the inductive hypothesis to be used more than once, and allows instantiating the inductive hypothesis to other than the predecessor. Complete induction and ordinary mathematical induction are each derivable from the other, hence the use of ‘seemingly’.

`measureInduct_on` takes a quotation, and breaks it apart to find a term and a measure function with which to induct.

6.2 Function definition

`Define` : term quotation -> thm

The `Define` function is a general-purpose function definition mechanism. It will define non-recursive and primitive recursive functions and attempt to define recursive-but-not-primitive-recursive functions. For these more difficult recursions, it attempts to find a measure under which recursive calls become smaller (and to prove that they do indeed become smaller). Currently, it examines the domain type of the function being defined and synthesizes a “size” measure. Then it does some basic simplifications and then attempts to automatically prove the termination constraints. If this termination proof fails, then the termination constraints remain on the hypotheses. An induction theorem for the function is also automatically derived: the definition and the induction principle are conjoined.

Example. Invoking

```
Define
  '(gcd 0 y = y) /\
  (gcd (SUC x) 0 = SUC x) /\
  (gcd (SUC x) (SUC y) =
    ((y <= x) => gcd (x-y) (SUC y)
     | gcd (SUC x) (y-x)))';
```

proves all termination conditions and returns

```

|- ((gcd 0 y = y) /\
    (gcd (SUC x) 0 = SUC x) /\
    (gcd (SUC x) (SUC y) =
      (y <= x ==> (gcd (x - y) (SUC y))
        | (gcd (SUC x) (y - x))))))
/\
!P. (!y. P 0 y) /\
    (!x. P (SUC x) 0) /\
    (!x y. (~(y <= x) ==> P (SUC x) (y - x)) /\
      (y <= x ==> P (x - y) (SUC y))
      ==> P (SUC x) (SUC y))
==>
!v v1. P v v1.

```

- Nested recursive functions are currently rejected by `Define`. Use `tfllib.Rfunction` for such cases.
- `Define` assumes that the function being defined is to be parsed as a prefix. To define an infix or binder, first make a prefix definition with `Define` and then use the function `set_fixity`.
- `Define` assumes that the function being defined is to be given a theory-level binding built from the name of the defined constant. However, this sometimes will create a badly formed ML identifier, which will lead to failure of theory compilation. As a result, `Define` will warn the user when a malformed name has been generated. The user should then use `set_MLname` to override the system-generated name.

Example. The system responds to

```
Define '## f g (x,y) = (f x, g y)'
```

with (omitting some messages)

```
The name "##_def" should be changed to an alphanumeric.
```

```
Use "set_MLname".
```

```
> val it = |- !f g x y. ## f g (x,y) = (f x,g y) : Thm.thm
```

If we then invoke (for example)

```
set_fixity "##" (Infix 450);
set_MLname "##_def" "fpair_def";
```

then `##` will henceforth be an infix operator, and the theory-level binding for its definition will be accepted by ML.

6.3 Automated reasoners

`bossLib` brings together the most powerful reasoners in `Hol98` and tries to make it easy to compose them in a simple way. We take our basic reasoners from `mesonLib`, `simplib`, and `decisionLib`, but the point of `bossLib` is to provide a layer of abstraction so the user has to know only a few entrypoints.²

```

PROVE      : thm list -> term quotation -> thm
PROVE_TAC  : thm list -> tactic

DECIDE     : term quotation -> thm
DECIDE_TAC : tactic

```

The inference rule `PROVE` (and the corresponding tactic `PROVE_TAC`) takes a list of theorems and a quotation, and attempts to prove the term using a first order reasoner. The inference rule `DECIDE` (and the corresponding tactic `DECIDE_TAC`) applies a decision procedure that (at least) handles statements of linear arithmetic.

```

RW_TAC     : simpset -> thm list -> tactic
&&        : simpset * thm list -> simpset (* infix *)
bool_ss   : simpset
arith_ss  : simpset
list_ss   : simpset

```

The rewriting tactic `RW_TAC` works by first adding the given theorems into the given `simpset`; then it simplifies the goal as much as possible; then it performs case splits on any conditional expressions in the goal; then it repeatedly (1) eliminates all hypotheses of the form $v = M$ or $M = v$ where v is a variable not occurring in M , (2) breaks down any equations between constructor terms occurring anywhere in the goal. The infix combinator `&&` is used to build a new `simpset` from a given `simpset` and a list of theorems.

Simplification sets for its native datatypes are provided by `bossLib`. In general, these are extended versions of those found in `simplib`. The `simpset` for pure logic and pairs and the `option` type is named `bool_ss`. The `simpset` for arithmetic is named `arith_ss`, and the `simpset` for lists is named `list_ss`. The `simpsets` provided by `bossLib` strictly increase in strength: `bool_ss` is contained in `arith_ss`, and `arith_ss` is contained in `list_ss`.

```

STP_TAC   : simpset -> tactic -> tactic
ZAP_TAC   : simpset -> thm list -> tactic

```

The compound reasoners of `bossLib` take a basic approach: they simplify the goal as much as possible with `RW_TAC` and then a ‘finishing’ tactic is applied. The primitive entrypoint for this is `STP_TAC`. Currently, the most powerful reasoner is `ZAP_TAC`, which features

²In the mid 1980’s Graham Birtwistle advocated such an approach, calling it ‘Ten Tactic HOL’.

a finishing tactic that first tries a tautology checking tactic; if that fails, `DECIDE_TAC` is called; if that fails, `PROVE_TAC` is called with the second argument. Although this general approach (simplify as much as possible, then apply automated reasoners in sequence) is crude, we have found that it allows one to make good progress in a high percentage of proof situations.

```
by : term quotation * tactic -> tactic (* infix 8 *)
SPOSE_NOT_THEN : (thm -> tactic) -> tactic
```

The function `by` is an infix operator that takes a quotation and a tactic *tac*. The quotation is parsed into a term *M*. When the invocation “*M* by *tac*” is applied to a goal (A, g) , a new subgoal (A, M) is created and *tac* is applied to it. If the goal is proved, the resulting theorem is broken down and added to the assumptions of the original goal; thus the proof proceeds with the goal $((M :: A), g)$. (Note however, that case-splitting will happen if the breaking-down of $\vdash M$ exposes disjunctions.) Thus `by` allows a useful style of ‘assertional’ or ‘Mizar-like’ reasoning to be mixed with ordinary tactic proof³

`SPOSE_NOT_THEN` initiates a proof by contradiction by assuming the negation of the goal and driving the negation inwards through quantifiers. It provides the resulting theorem as an argument to the supplied function, which will use the theorem to build and apply a tactic.

Note. When the library `bossLib` is loaded, the infix parsing status of `&&` and “by” must be re-asserted by the user.

³Proofs in the Mizar system are readable documents, unlike almost all tactic-based proofs.

Examples

Eventually, this chapter will have a complete set of examples showing Hol98 in action, so that it could be used as a tutorial on the system. Currently, however, it is incomplete. The reader who wishes to see more can browse the `examples` directory of the distribution where the following small examples can 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 a record-definition package due to Mike Norrish (building on work of Phil Windley), as well as illustrating some aspects of the automation available in Hol98.

euclid.sml This example is 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.

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

MLsyntax This example shows the use 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.

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

7.1 Euclid's Theorem

In this section, we will 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` and is intended to serve as an introduction to performing high-level interactive proofs in Hol98.

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.enquote`.

```
> Moscow ML version 1.43 (April 1998)
> Enter 'quit();' to quit.
> For HOL help, type: help "hol";
>
>           HHH           LL
>           HHH           LL
>           HHH           LL
>           HHH           LL
>           HHH      0000      LL
>           HHHHHHHH      00 00      LL
>           HHHHHHHH      00 00      LLL
>           HHH      0000      LLLL
>           HHH           LL  LL
>           HHH           LL  LL
>           HHH           LL  LL
>           HHH           LL      LL98 [Athabasca 2]
>
> [closing file "/home/kxs/hol98/std.prelude"]
> [opening file "/home/kxs/hol98/tools/use.sml"]
> Rebinding "use" for quotation pre-processing.
> [closing file "/home/kxs/hol98/tools/use.sml"]
```

Then we load and open `bossLib`. This library provides high-level support for interactive proof, as described in Chapter 6. We also open the theory of arithmetic, since we will use some of its theorems.

```
load "bossLib";
open bossLib;
infix &&; infix 8 by;
open arithmeticTheory;
```

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
```

The ML type of `ARW_TAC` is *thm list* \rightarrow *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 a *prime* number, we first need to define the *divisibility* relation:

```
val divides = Define 'divides a b = ?x. b = a * x'
```

The definition is silently added to the current theory (see Section 5), 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 ML binding is made for each definition and (useful) proved theorem: the ML environment is being used as a convenient place to hold definitions and theorems for later reference.

We want to treat `divides` as an infix:

```
set_fixity "divides" (Infix 450)
```

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 =
  Define 'prime p = ~(p=1) /\ !x. x divides p ==> (x=1) \/ (x=p)'
```

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 found 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 a couple of possibilities: the help system can be used to look up definition and theorems, as well as proof procedures; for example, an invocation of `help`

¹Linear arithmetic especially: purely universal statements involving the operators `SUC`, `+`, `-`, numeric literals, `<`, `≤`, `>`, `≥`, `=`, and multiplication by numeric literals.

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

"`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 more easy 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 key 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. 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.

7.1.1 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 $\forall x. x \text{ divides } 0$. A new proposition to be proved is entered to the proof manager via "g", which starts a fresh goalstack:³

```
g '!x. x divides 0';

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

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]);

> OK..
> 1 subgoal:
>   val it =
>     ?x'. 0 = x * x'
```

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

³System output is indicated by a `>` in the first column.

```

    e (EXISTS_TAC ''0'');
> OK..
> 1 subgoal:
>   val it =
>     0 = x * 0

```

Then a simplification step finishes the proof.

```

    e (ARW_TAC []);
> OK..
> Goal proved.
> |- 0 = x * 0
>
> Goal proved.
> |- ?x'. 0 = x * x'
> val it =
>   Initial goal proved.
> |- !x. x divides 0

```

What has happened here? The application of `ARW_TAC` to the goal decomposed it to an empty list of subgoals, plus a justification function. The system then applied the justification function to the empty list, and proved the goal. 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 previous justification function on the stack. When all the theorems required by *that* justification function are available, it can then evaluate, and prove its corresponding goal. 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,⁴ 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 and this time it will take just one step:

```

    restart();
    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`.

⁴Perhaps since we have used a stack to implement what is notionally a tree!

```

restart();
e (PROVE_TAC [divides, MULT_CLAUSES]);
> OK..
> Meson search level: .....
> val it =
>   Initial goal proved.
>   |- !x. x divides 0

```

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`.

$$(DIVIDES_0) \frac{!x. x \text{ divides } 0}{\text{PROVE_TAC [divides, MULT_CLAUSES]}}$$

$$(DIVIDES_ZERO) \frac{!x. 0 \text{ divides } x = (x = 0)}{\text{PROVE_TAC [divides, MULT_CLAUSES]}}$$

$$(DIVIDES_ONE) \frac{!x. x \text{ divides } 1 = (x = 1)}{\text{PROVE_TAC [divides, MULT_CLAUSES, MULT_EQ_1]}}$$

$$(DIVIDES_REFL) \frac{!x. x \text{ divides } x}{\text{PROVE_TAC [divides, MULT_CLAUSES]}}$$

$$(DIVIDES_TRANS) \frac{!a b c. a \text{ divides } b \wedge b \text{ divides } c \implies a \text{ divides } c}{\text{PROVE_TAC [divides, MULT_ASSOC]}}$$

$$(DIVIDES_ADD) \frac{!d a b. d \text{ divides } a \wedge d \text{ divides } b \implies d \text{ divides } (a+b)}{\text{PROVE_TAC [divides, LEFT_ADD_DISTRIB]}}$$

$$(DIVIDES_SUB) \frac{!d a b. d \text{ divides } a \wedge d \text{ divides } b \implies d \text{ divides } (a-b)}{\text{PROVE_TAC [divides, LEFT_SUB_DISTRIB]}}$$

$$(DIVIDES_ADDL) \frac{!d a b. d \text{ divides } a \wedge d \text{ divides } (a+b) \implies d \text{ divides } b}{\text{PROVE_TAC [ADD_SUB, ADD_SYM, DIVIDES_SUB]}}$$

$$(DIVIDES_LMUL) \frac{!d a x. d \text{ divides } a \implies d \text{ divides } (x * a)}{\text{PROVE_TAC [divides, MULT_ASSOC, MULT_SYM]}}$$

$$(DIVIDES_RMUL) \frac{!d a x. d \text{ divides } a \implies d \text{ divides } (a * x)}{\text{PROVE_TAC [MULT_SYM, DIVIDES_LMUL]}}$$

Now we encounter a lemma about divisibility that doesn't succumb to a single invocation of `PROVE_TAC`:

```
(DIVIDES_LE)  $\frac{!m\ n.\ m\ \text{divides}\ n\ ==>\ m\ <= n\ \ \backslash/\ (n = 0)}{\text{ARW\_TAC [divides]}}$ 
              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:

```
g '!m n . m divides n ==> m <= n \/\ (n = 0)';
e (ARW_TAC [divides]);
> OK..
> 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.

```
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 is trivial:

```
e (ARW_TAC []);
> 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 []);
> 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 seemed to make some proofs *more* complicated, rather than simpler. OK, so let's manually add `MULT_CLAUSES` in.

```
e (ARW_TAC [MULT_CLAUSES]);
> 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) three times:

```
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)
```

And 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`. Recall that 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]);
> OK..
>
> Goal proved. ...
> 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. The actual ML is the following:⁵

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

7.1.1.1 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 $\text{FACT } n = 1 * \dots * m * \dots * n$. To show $m \text{ divides } (\text{FACT } n)$ means exhibiting a q such that $q * m = \text{FACT } n$. Thus $q = \text{FACT } n \div m$. 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 $n \text{ divides } (\text{FACT } n)$, 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) ' ;
> val it =
> Proof manager status: 1 proof.
> 1. Incomplete:
>   Initial goal:
>   !m n. 0 < m /\ m <= n ==> m divides FACT n
```

⁵`store_thm` 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.

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;
> val it = |- !m n. m <= n = (?p. n = m + p)

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');
> 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 []);
> OK..
> 1 subgoals:
> 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');
> OK..
> 2 subgoals:
> val it =
>   SUC n divides FACT (SUC n)
>   -----
>   0 < SUC n

```

```

>
>   0 divides FACT 0
>   -----
>   0 < 0

e (PROVE_TAC [DECIDE '!x. ~(x < x)']);
> OK..
> Meson search level: ..
> Goal proved. ....
>
> Remaining subgoals:
> val it =
>   SUC n divides FACT (SUC n)
>   -----
>   0 < SUC n

e (ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);
> 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();
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]);
> 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]);
> 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 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();
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 would be packaged into an invocation of `prove` 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]
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.

7.1.1.2 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();
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 slightly hard to read, so we use `STRIP_TAC` 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();
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:num = n' by DECIDE_TAC);
> OK..
> 1 subgoal:
> val it =
>   m divides FACT n
>   -----

```

```

>      0.  0 = n - m
>      1.  0 < m
>      2.  m <= n
>      3.  m = n

```

Notice that we needed to constrain the type of m (and thus that of n) because otherwise they would be assigned a polymorphic type and then they would not be the same as the m and n already occurring in the goal, which have type *num*.⁶ We can now use `ARW_TAC` to propagate the newly derived equality throughout the goal.

```

      e (ARW_TAC []);
> 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 had to do a case analysis on m . However, in the current proof, we already have the hypothesis that m is positive. Thus we know that m is the successor of some number k . We might like 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 one⁷ and an overview of the extant proof attempts will be printed:

```

g '!m. 0 < m ==> ?k. m = SUC k';

> val it =
>   Proof manager status: 2 proofs.
>   2. Incomplete:
>     Initial goal:
>     !m n. 0 < m /\ m <= n ==> m divides FACT n
>
>

```

⁶The quotation in “*quotation by tactic*” is currently parsed in ignorance of the assignment of types to terms in the goal.

⁷This stacking of proof attempts (goalstacks) is different than the stacking of goals and justifications inside a particular goalstack.

```

>      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)

```

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”⁸ function.

```

      e (Cases THEN ARW_TAC []);

> 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)

      p ();
> 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 PROVE_TAC [lem]
        THEN ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);

> OK..
> Meson search level: ...
>
> Goal proved. ....
> Remaining subgoals:

```

⁸An infix version of the K combinator, defined by `fun (x before y) = x.`

```

> 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

```

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]);

> 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]);

> 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.

7.1.2 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.

$$\frac{(NOT_PRIME_0) \quad \sim\text{prime } 0}{ARW_TAC \quad [prime, DIVIDES_0]}$$

$$(NOT_PRIME_1) \frac{\sim \text{prime } 1}{ARW_TAC \text{ [prime]}}$$

$$(PRIME_2) \frac{\text{prime } 2}{ARW_TAC \text{ [prime]}}$$

THEN PROVE_TAC [DIVIDES_LE, DIVIDES_ZERO,
DECIDE ' $\sim(2=1)$ ', DECIDE ' $\sim(2=0)$ ',
DECIDE ' $x \leq 2 = (x=0) \vee (x=1) \vee (x=2)$ ']

$$(PRIME_POS) \frac{!p. \text{prime } p \implies 0 < p}{Cases \text{ THEN } ARW_TAC[NOT_PRIME_0]}$$

7.1.3 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:

$$(PRIME_FACTOR) \frac{!n. \sim(n = 1) \implies ?p. \text{prime } p \wedge p \text{ divides } n}{\text{completeInduct_on 'n'}}$$

THEN STRIP_TAC
THEN Cases 'prime n' THENL
[PROVE_TAC [DIVIDES_REFL],
' $?x. x \text{ divides } n \wedge \sim(x=1) \wedge \sim(x=n)$ '
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 :

```
e (completeInduct_on 'n');
> OK..
> 1 subgoal:
> val it =
>    $\sim(n = 1) \implies (?p. \text{prime } p \wedge p \text{ divides } n)$ 
>   -----
>    $!m. m < n \implies \sim(m = 1) \implies (?p. \text{prime } p \wedge p \text{ divides } m)$ 
```

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 (STRIP_TAC THEN Cases_on 'prime n');
> 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 n
>   -----
>   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]);
> 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]);
> 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]);
> 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]);
> OK..
> Meson search level: .....
> Goal proved. ....

```

The remaining goal can be clarified by simplification:

```

e (ARW_TAC []);
> 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]);
> 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.

7.1.4 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 $\text{FACT}(n) + 1$: it's not equal to 1 so, by *PRIME_FACTOR*, there's a prime p that divides it. Note that p also divides $\text{FACT}(n)$ 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 $\neg(\forall n. \exists p. n < p \wedge \text{prime } p)$ is simplified a bit into $\exists n. \forall p. n < p \supset \neg \text{prime } p$ 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 .

```
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`.⁹ 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:¹⁰

```
PRIME_FACTOR;
> 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 $\vdash \neg(\text{FACT } n + 1 = 1)$ 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));
> 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 []);
> OK..
```

⁹The HOL parser thinks `FACT n + 1` is equivalent to $(\text{FACT } n) + 1$.

¹⁰The function `SPEC` implements the rule of universal specialization.

```

> 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 ']);
> 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 $(\text{FACT } n)$, 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,
              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 allows 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.¹¹

```
(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 []
```

7.1.5 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

¹¹Note 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.

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). 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: (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.

The Programmer's Interface

References

- [1] Peter Andrews. *An Introduction to Mathematical Logic and Type Theory: To Truth Through Proof*. Academic Press, 1986.
- [2] Alonzo Church. A formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–68, 1940.
- [3] Michael Gordon, Robin Milner, and Christopher Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.
- [4] Mike Gordon and Tom Melham. *Introduction to HOL, a theorem proving environment for higher order logic*. Cambridge University Press, 1993.
- [5] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. The MIT Press, 1997.
- [6] Lawrence Paulson. *ML for the working programmer*. Cambridge University Press, second edition, 1996.
- [7] J. Ullman. *Elements of ML Programming*. Prentice-Hall, second edition, 1997.