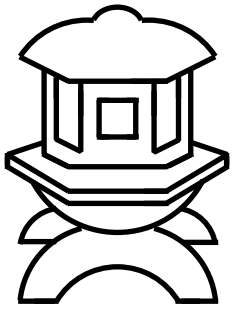
[For HOL Kananaskis-4] January 2, 2007

The HOL System

TUTORIAL



Preface

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

(i) *LOGIC*: a formal description of the higher order logic implemented by the HOL system.

(ii) *TUTORIAL*: a tutorial introduction to HOL, with case studies.

(iii) *DESCRIPTION*: a detailed user's guide for the HOL system;

(iv) *REFERENCE*: the reference manual for HOL.

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

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

Getting started

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

1. The programming meta-language ML, and how to interact with it.
2. The formal logic supported by the HOL system (higher order logic) and its manipulation via ML.
3. Forward proof and derived rules of inference.
4. Goal directed proof, tactics, and tacticals.

Chapters 2 and 3 introduce these topics. Chapter 4 then develops an extended example—Euclid's proof of the infinitude of primes—to illustrate how HOL is used to prove theorems.

Chapter 5 features another worked example: the specification and verification of a simple sequential parity checker. The intention is to accomplish two things: (i) to present another complete piece of work with HOL; and (ii) to give an idea of what it is like to use the HOL system for a tricky proof. Chapter 6 is a more extensive example: the proof of confluence for combinatory logic. Again, the aim is to present a complete piece of non-trivial work.

Chapter 7 gives an example of implementing a proof tool of one's own. This demonstrates the programmability of HOL: the way in which technology for solving specific problems can be implemented on top of the underlying kernel. With high-powered tools to draw on, it is possible to write prototypes very quickly.

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

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

Acknowledgements

The bulk of HOL is based on code written by—in alphabetical order—Hasan Amjad, Richard Boulton, Anthony Fox, Mike Gordon, Elsa Gunter, John Harrison, Peter Homeier, Gérard Huet (and others at INRIA), Joe Hurd, Ken Larsen, Tom Melham, Robin Milner, Lockwood Morris, Malcolm Newey, Michael Norrish, Larry Paulson, Konrad Slind, Don Syme, and Chris Wadsworth. Many others have supplied parts of the system, bug fixes, etc.

Current edition

The current edition of all four volumes (*LOGIC*, *TUTORIAL*, *DESCRIPTION* and *REFERENCE*) has been prepared by Michael Norrish and Konrad Slind. Further contributions to these volumes came from: Hasan Amjad, who developed a model checking library and wrote sections describing its use; Jens Brandt, who developed and documented a library for the rational numbers; Anthony Fox, who formalized and documented new word theories and the associated libraries; Mike Gordon, who documented the libraries for BDDs and SAT; Peter Homeier, who implemented and documented the quotient library; and Joe Hurd, who added material on first order proof search.

The material in the third edition constitutes a thorough re-working and extension of previous editions. The only essentially unaltered piece is the semantics by Andy Pitts (in *LOGIC*), reflecting the fact that, although the HOL system has undergone continual development and improvement, the HOL logic is unchanged since the first edition (1988).

Second edition

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

First edition

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

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

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

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

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

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

Contents

Chapter 1

Getting and Installing HOL

This chapter describes how to get the HOL system and how to install it. It is generally assumed that some sort of Unix system is being used, but the instructions that follow should apply *mutatis mutandis* to other platforms. Unix is not a pre-requisite for using the system. HOL may be run on PCs running Windows operating systems from Windows NT onwards (i.e., Windows 2000 and Windows XP are also supported), as well as Macintoshes running MacOS X.

1.1 Getting HOL

The HOL system can be downloaded from `http://hol.sourceforge.net`. The naming scheme for HOL releases is ⟨*name*⟩-⟨*number*⟩; the release described here is Kananaskis-4.

1.2 The `hol-info` mailing list

The `hol-info` mailing list serves as a forum for discussing HOL and disseminating news about it. If you wish to be on this list (which is recommended for all users of HOL), visit `http://lists.sourceforge.net/lists/listinfo/hol-info`. This web-page can also be used to unsubscribe from the mailing list.

1.3 Installing HOL

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

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

Principal Files on the HOL Distribution Directory

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

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

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

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

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

Before beginning you must have a current version of Moscow ML. In particular, you must have version 2.01. Moscow ML is available on the web from `http://www.dina.kvl.dk/~sestoft/mosml.html`. When you have `mosml` installed, and are in the root directory of the distribution, the next step is to run `smart-configure`:

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

```
$ mosml < tools/smart-configure.sml                                    2
Moscow ML version 2.01 (January 2004)
Enter 'quit();' to quit.
-
HOL smart configuration.

Determining configuration parameters: OS mosmldir holdir
OS:                 linux
mosmldir:           /home/mn200/mosml
holdir:             /home/mn200/hol
dynlib_available:   true

Configuration will begin with above values.  If they are wrong
press Control-C.
```

Assuming you don't interrupt the configuration process, this will build the `Holmake` and `build` programs, and move them into the `hol/bin` directory. If something goes wrong at this stage, consult Section 1.3.1 below.

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

```
$ bin/build                                                            3
  ...
  ...
Uploading files to /home/mn200/hol/sigobj

Hol built successfully.
$
```

1.3.1 Overriding `smart-configure`

If `smart-configure` is unable to guess correct values for the four parameters, `mosmldir`, `holdir`. `OS` and `dynlib_available` then you will need to create a file called `config-override` in the root directory of the HOL distribution. In this file, specify the correct value for the appropriate parameter by providing an ML binding for it. Each of the first three variables must be given a string as a possible value, while `dynlib_available` must be either `true` or `false`. So, one might write

```
val OS = "unix";                                                       4
val holdir = "/local/scratch/myholdir";
val dynlib_available = false;
```

[3] Four HOL executables are produced: hol, hol.noquote, hol.bare and hol.bare.noquote. The first of these will be used for most examples in the *TUTORIAL*.

The `config-override` file need only provide values for those variables that need overriding.

With this file in place, the `smart-configure` program will use the values specified there rather than those it attempts to calculate itself. The value given for the `OS` variable must be one of `"unix"`, `"linux"`, `"solaris"`, `"macosx"` or `"winNT"`.[4]

In extreme circumstances it is possible to edit the file `configure.sml` yourself to set configuration variables directly. At the top of this file three SML declarations are present, but commented out. You will need to uncomment this section (remove the `(*` and `*)` markers), and provide sensible values. The `mosmldir` value must be the name of the directory containing the Moscow ML binaries (`mosmlc`, `mosml`, `mosmllex` etc). The `holdir` value must be the name of the top-level directory listed in the first session above. The `OS` value should be one of the strings specified in the accompanying comment. All three strings must be enclosed in double quotes.

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

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

```
$ more configure.sml                                                    5
  ...
val mosmldir = "/home/mn200/mosml";
val holdir   = "/home/mn200/hol";
val OS       = "linux"       (* Operating system; choices are:
                                "linux", "solaris", "unix", "winNT" *)

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

Now, at either this level (in the `tools` directory) or at the level above, the `configure.sml` script must be piped into the Moscow ML interpreter (called `mosml`).

[4]The string `"winNT"` is used for Microsoft Windows operating systems that are at least as recent as Windows NT. This includes Windows XP and Windows 2000.

6

```
$ mosml < tools/configure.sml
Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
- > val mosmldir = "/home/mn200/mosml" : string
  val holdir = "/home/mn200/hol" : string
  val OS = "linux" : string
- > val CC = "gcc" : string
  ...
Beginning configuration.
- Making bin/Holmake.
  ...
Making bin/build.
- Making hol98-mode.el (for Emacs)
- Setting up the standard prelude.
- Setting up src/0/Globals.sml.
- Generating bin/hol.
- Generating bin/hol.noquote.
- Attempting to compile quote filter ... successful.
- Setting up the muddy library Makefile.
- Setting up the help Makefile.
-
Finished configuration!
-
$
```

Chapter 2

Introduction to ML

This chapter is a brief introduction to the meta-language ML. The aim is just to give a feel for what it is like to interact with the language. A more detailed introduction can be found in numerous textbooks and web-pages; see for example the list of resources on the MoscowML home-page[1], or the `comp.lang.ml` FAQ[2].

2.1 How to interact with ML

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

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

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

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

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

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

The interactions in these boxes should be understood as occurring in sequence. For example, variable bindings made in earlier boxes are assumed to persist to later ones.

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

[2] `http://www.faqs.org/faqs/meta-lang-faq/`

To enter the HOL system one types `hol` or `hol.noquote` to Unix, possibly preceded by path information if the HOL system's `bin` directory is not in one's path. The HOL system then prints a sign-on message and puts one into ML. The ML prompt is –, so lines beginning with – are typed by the user and other lines are the system's responses.

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

1

```
$ bin/hol

------------------------------------------------------------------
       HOL-4 [Kananaskis 4 (built Fri Apr 12 15:34:35 2002)]

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

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

The ML expression `1 :: [2,3,4,5]` has the form $e_1\ op\ e_2$ where e_1 is the expression `1` (whose value is the integer 1), e_2 is the expression `[2,3,4,5]` (whose value is a list of four integers) and op is the infixed operator '`::`' which is like Lisp's *cons* function. Other list processing functions include `hd` (car in Lisp), `tl` (cdr in Lisp) and `null` ($null$ in Lisp). The semicolon ';' terminates a top-level phrase. The system's response is shown on the line starting with the > prompt. It consists of the value of the expression followed, after a colon, by its type. The ML type checker infers the type of expressions using methods invented by Robin Milner [8]. The type `int list` is the type of 'lists of integers'; `list` is a unary type operator. The type system of ML is very similar to the type system of the HOL logic which is explained in Chapter 3.

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

2

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

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

- hd it;
> val it = 2 : int

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

Following standard λ-calculus usage, the application of a function f to an argument x can be written without brackets as $f\ x$ (although the more conventional $f(x)$ is also

allowed). The expression $f\ x_1\ x_2\ \cdots\ x_n$ abbreviates the less intelligible expression $(\cdots((f\ x_1)x_2)\cdots)x_n$ (function application is left associative).

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

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

ML expressions like `"a"`, `"b"`, `"foo"` etc. are *strings* and have type `string`. Any sequence of ASCII characters can be written between the quotes.[3] The function `explode` splits a string into a list of single characters, which are written like single character strings, with a `#` character prepended.

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

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

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

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

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

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

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

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

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

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

```
- fun zip(l1,l2) =                                                    6
    if null l1 orelse null l2 then []
    else (hd l1,hd l2) :: zip(tl l1,tl l2);
> val zip = fn : 'a list * 'b list -> ('a * 'b) list

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

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

```
- fun curried_zip l1 l2 = zip(l1,l2);                                 7
> val curried_zip = fn : 'a list -> 'b list -> ('a * 'b) list

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

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

The evaluation of an expression either *succeeds* or *fails*. In the former case, the evaluation returns a value; in the latter case the evaluation is aborted and an *exception* is raised. This exception passed to whatever invoked the evaluation. This context can either propagate the failure (this is the default) or it can *trap* it. These two possibilities are illustrated below. An exception trap is an expression of the form e_1 `handle _ =>` e_2. An expression of this form is evaluated by first evaluating e_1. If the evaluation succeeds (i.e. doesn't fail) then the value of the whole expression is the value of e_1. If the evaluation of e_1 raises an exception, then the value of the whole is obtained by evaluating e_2.[4]

```
- 3 div 0;                                                            8
! Uncaught exception:
! Div

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

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

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

Chapter 3

The HOL Logic

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

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

In this chapter, we will give a brief overview of the notation used to write expressions of the HOL logic in ML, and also discuss standard HOL proof techniques. It is assumed the reader is familiar with predicate logic. The syntax and semantics of the particular logical system supported by HOL is described in detail in *DESCRIPTION*.

The table below summarizes a useful subset of the notation used in HOL.

Terms of the HOL Logic

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

Terms of the HOL logic are represented in ML by an *abstract type* called `term`. They are normally input between double back-quote marks. For example, the expression

``x /\ y ==> z`` evaluates in ML to a term representing x∧y⇒z. Terms can be manipulated by various built-in ML functions. For example, the ML function `dest_imp` with ML type `term -> term * term` splits an implication into a pair of terms consisting of its antecedent and consequent, and the ML function `dest_conj` of type `term -> term * term` splits a conjunction into its two conjuncts. [1]

1

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

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

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

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

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

2

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

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

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

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

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

[1] All of the examples below assume that the user is running `hol`, the executable for which is in the `bin/` directory.

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

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

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

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

The constructors `mk_conj` and `mk_imp` construct conjunctions and implications respectively. A large collection of term constructors and destructors is available for the core theories in HOL.

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

Type checking There are actually only four different kinds of term in HOL: variables, constants, function applications (``$t_1\ t_2$``), and lambda abstractions (``$\backslash x.t$``). More complex terms, such as those we have already seen above, are just compositions of terms from this simple set. In order to understand the behaviour of the quotation parser, it is necessary to understand how the type checker infers types for the four basic term categories.

Both variables and constants have a name and a type; the difference is that constants cannot be bound by quantifiers, and their type is fixed when they are declared (see below). When a quotation is entered into HOL, the type checking algorithm uses the types of constants to infer the types of variables in the same quotation. For example, in the following case, the HOL type checker used the known type `bool->bool` of boolean negation (`~`) to deduce that the variable `x` must have type `bool`.

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

In the next two cases, the type of `x` and `y` cannot be deduced. (The default 'scope' of type information for type checking is a single quotation, so a type in one quotation cannot affect the type-checking of another. Thus `x` is not constrained to have the the type `bool` in the second quotation.)

```
- ``x``;                                                                    7
<<HOL message: inventing new type variable names: 'a.>>
> val it = ``x`` : Term.term

- type_of it;
> val it = ``:'a`` : hol_type

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

- type_of it;
> val it = ``:'a # 'b`` : hol_type
```

If there is not enough contextually-determined type information to resolve the types of all variables in a quotation, then the system will guess different type variables for all the unconstrained variables.

Type constraints Alternatively, it is possible to explicitly indicate the required types by using the notation ``$term$:$type$``, as illustrated below.

```
- ``x:num``;                                                                8
> val it = ``x`` : term

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

Function application types An application $(t_1\ t_2)$ is well-typed if t_1 is a function with type $\tau_1 \to \tau_2$ and t_2 has type τ_1. Contrarily, an application $(t_1\ t_2)$ is badly typed if t_1 is not a function:

```
- ``1 2``;                                                                  9

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

(1 :num)

to

(2 :num)

unification failure message: unify failed
! Uncaught exception:
! HOL_ERR
```

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

10
```
- ``~1``;

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

$~

to

(1 :num)

unification failure message: unify failed
! Uncaught exception:
! HOL_ERR
```

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

11
```
- ``$==> t1 t2``;
> val it = ``t1 ==> t2`` : term

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

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

12
```
- type_of ``$==>``;
> val it = ``:bool -> bool -> bool`` : hol_type

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

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

13
```
- ``(x + 1, t1 ==> t2)``;
> val it = ``(x + 1,t1 ==> t2)`` : term

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

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

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

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

```
- ``\x. x + 1``;                                                    14
> val it = ``\x. x + 1`` : term

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

Other binding symbols in the logic are its two most important quantifiers: ! and ?, universal and existential quantifiers. For example, the logical statement that every number is either even or odd might be phrased as

```
!n. (n MOD 2 = 1) \/ (n MOD 2 = 0)
```

while a version of Euclid's result about the infinitude of primes is:

```
!n. ?p. prime p /\ p > n
```

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

```
- ``\x y. (x, y * x)``;                                             15
> val it = ``\x y. (x,y * x)`` : term

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

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

3.1 Proof in HOL

This section discusses the nature of proof in HOL. For a logician, one definition of a formal proof is that it is a sequence, each of whose elements is either an *axiom* or follows from earlier members of the sequence by a *rule of inference*. A theorem is the last element of a proof.

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

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

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

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

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

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

This rule is represented in ML by a function `SPEC`,[2] which takes as arguments a term ``` ``a`` ``` and a theorem `|- !`$x.t[x]$ and returns the theorem `|-` $t[a]$, the result of substituting a for x in $t[x]$.

2

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

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

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

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

If the argument to an ML function representing a rule of inference is of the wrong kind, or violates a condition of the rule, then the application fails. For example, `SPEC` $t\ th$ will fail if th is not of the form `|- !`$x.\cdots$ or if it is of this form but the type of t is not the same as the type of x, or if the free variable restriction is not met. When one of the standard `HOL_ERR` exceptions is raised, more information about the failure can often be gained by using the `Raise` function. [3]

3

```
- SPEC ``1=2`` Th2;
! Uncaught exception:
! HOL_ERR

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

Exception raised at Thm.SPEC:

! Uncaught exception:
! HOL_ERR
```

[2] `SPEC` is not a primitive rule of inference in the HOL logic, but is a derived rule. Derived rules are described in Section 3.2.

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

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

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

The general form of a theorem is $t_1, \ldots, t_n$ `|-` t, where t_1, ... , t_n are boolean terms called the *assumptions* and t is a boolean term called the *conclusion*. Such a theorem asserts that if its assumptions are true then so is its conclusion. Its truth conditions are thus the same as those for the single term $(t_1$`/\...\`$t_n)$`==>`t. Theorems with no assumptions are printed out in the form `|-` t.

The five axioms and eight primitive inference rules of the HOL logic are described in detail in the document *DESCRIPTION*. Every value of type `thm` in the HOL system can be obtained by repeatedly applying primitive inference rules to axioms. When the HOL system is built, the eight primitive rules of inference are defined and the five axioms are bound to their ML names, all other predefined theorems are proved using rules of inference as the system is made.[4] This is one of the reasons why building `hol` takes so long.

In the rest of this section, the process of *forward proof*, which has just been sketched, is described in more detail. In Section 3.3 *goal directed proof* is described, including the important notions of *tactics* and *tacticals*, due to Robin Milner.

3.2 Forward proof

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

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

4

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

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

[4] This is a slight over-simplification.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.1 Derived rules

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

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

1. $t' \vdash t'$ [Assumption introduction]
2. $\Gamma \vdash t$ [Hypothesis]

3. $\Gamma \vdash t' \Rightarrow t$ [Discharge t' from line 2]

4. $\Gamma, t' \vdash t$ [Modus Ponens applied to lines 3 and 1]

This proof works for any hypothesis of the form $\Gamma \vdash t$ and any boolean term t' and shows that the result of adding an arbitrary hypothesis to a theorem is another theorem (because the four lines above can be added to any proof of $\Gamma \vdash t$ to get a proof of $\Gamma, t' \vdash t$).[5] For example, the next session uses this proof to add the hypothesis ‘‘t3‘‘ to Th6.

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

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

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

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

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

The ML definition is:

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

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

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

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

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

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

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

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

An ML derived rule that implements this is:

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

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

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

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

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

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

3.2.1.1 Rewriting

An interesting derived rule is REWRITE_RULE. This takes a list of equational theorems of the form:

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

and a theorem $\Delta \vdash t$ and repeatedly replaces instances of u_i in t by the corresponding instance of v_i until no further change occurs. The result is a theorem $\Gamma \cup \Delta \vdash t'$ where t' is the result of rewriting t in this way. The session below illustrates the use of REWRITE_RULE. In it the list of equations is the value rewrite_list containing the pre-proved theorems ADD_CLAUSES and MULT_CLAUSES. These theorems are from the theory

arithmetic, so we must use a fully qualified name with the name of the theory as the first component to refer to them. (Alternatively, we could, as in the Euclid example of section 4, use open to bring declare all of the values in the theory at the top level.)

```
- open arithmeticTheory;                                                    15

   ...

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

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

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

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

REWRITE_RULE is not a primitive in HOL, but is a derived rule. It is inherited from Cambridge LCF and was implemented by Larry Paulson (see his paper [10] for details). In addition to the supplied equations, REWRITE_RULE has some built in standard simplifications:

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

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

3.3 Goal Oriented Proof: Tactics and Tacticals

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

(i) Splits a 'goal' into 'subgoals'.

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

Consider, for example, the rule of $\wedge$-introduction[6] shown below:

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

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

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

This is illustrated in the following new session (note that the session number has been reset to *1*:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

achieves a goal

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

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

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

Tactics are specified using the following notation:

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

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

$$\frac{t_1 \ /\backslash \ t_2}{t_1 \quad t_2}$$

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

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

$$\frac{!n.t[n]}{t[0] \quad \{t[n]\}\ t[\mathtt{SUC}\ n]}$$

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

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

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

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

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

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

3.3.1 Using tactics to prove theorems

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

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

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

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

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

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

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

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

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

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

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

When conducting a proof that involves many subgoals and tactics, it is necessary to keep track of all the justification functions and compose them in the correct order. While this is feasible even in large proofs, it is tedious. HOL provides a package for building and traversing the tree of subgoals, stacking the justification functions and applying them properly; this package was originally implemented for LCF by Larry Paulson. Its use is demonstrated in Chapter 4, and thoroughly documented in *DESCRIPTION*.

3.3.2 Tacticals

A *tactical* is an ML function that takes one or more tactics as arguments, possibly with other arguments as well, and returns a tactic as its result. The various parameters passed to tacticals are reflected in the various ML types that the built-in tacticals have. Some important tacticals in the HOL system are listed below.

3.3.2.1 `THENL : tactic -> tactic list -> tactic`

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

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

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

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

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

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

3.3.2.2 `THEN : tactic -> tactic -> tactic`

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

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

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

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

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

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

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

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

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

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

3.3.2.3 `ORELSE : tactic -> tactic -> tactic`

The tactical `ORELSE` is an ML infix. If T_1 and T_2 are tactics, then T_1 `ORELSE` T_2 evaluates to a tactic which applies T_1 unless that fails; if it fails, it applies T_2. `ORELSE` is defined in ML as a curried infix by[7]

(*T*₁ `ORELSE` *T*₂) *g* = *T*₁ *g* `handle _ =>` *T*₂ *g*

3.3.2.4 `REPEAT : tactic -> tactic`

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

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

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

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

[7] This is a minor simplification.

2

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

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

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

3.3.3 Some tactics built into HOL

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

Before beginning, note that the ML type `thm_tactic` abbreviates `thm->tactic`, and the type `conv`[8] abbreviates `term->thm`.

3.3.3.1 `REWRITE_TAC : thm list -> tactic`

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

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

  where t' is obtained from t by rewriting with

  1. $th_1, \ldots, th_n$ and
  2. the standard rewrites held in the ML variable `basic_rewrites`.

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

- **Other rewriting tactics**:

[8] The type `conv` comes from Larry Paulson's theory of conversions [10].

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

3.3.3.2 CONJ_TAC : tactic

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

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

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

3.3.3.3 EQ_TAC : tactic

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

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

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

3.3.3.4 DISCH_TAC : tactic

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

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

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

3.3.3.5 GEN_TAC : tactic

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

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

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

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

3.3.3.6 PROVE_TAC : thm list -> tactic

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

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

3.3.3.7 STRIP_TAC : tactic

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

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

3.3.3.8 ACCEPT_TAC : thm -> tactic

- **Summary:** ACCEPT_TAC th is a tactic that solves any goal that is achieved by th.

- **Use:** Incorporating forward proofs, or theorems already proved, into goal directed proofs. For example, one might reduce a goal g to subgoals g_1, $\ldots$, g_n using a tactic T and then prove theorems th_1 , $\ldots$, th_n respectively achieving these goals by forward proof. The tactic

  T THENL[ACCEPT_TAC th_1, $\ldots$,ACCEPT_TAC th_n]

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

3.3.3.9 ALL_TAC : tactic

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

3.3.3.10 NO_TAC : tactic

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

Chapter 4

Example: Euclid's Theorem

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

Some tutorial descriptions of proof systems show the system performing amazing feats of automated theorem proving. In this example, we will *not* take this approach; instead, we try to show how one actually goes about the business of proving theorems in HOL: 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' HOL 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, with the command `<holdir>/bin/hol`. Now, we "`open`" the arithmetic theory, and specialize the rewriter provided by `bossLib` to a simplification set that knows about arithmetic. The former means that all of the ML bindings from the HOL theory of arithmetic are available at the top level. The latter is not necessary, but is a convenient abbreviation and serves to make some of the proofs typeset more nicely.

1

```
- open arithmeticTheory;
  ...

- val ARW_TAC = RW_TAC arith_ss;
> val ARW_TAC =
    fn
    : thm list -> term list * term ->
      (term list * term) list * (thm list -> thm)
```

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

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

```
- val divides = Define ‘divides a b = ?x. b = a * x‘;                    2

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

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

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

```
- set_fixity "divides" (Infixr 450);                                     3
```

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 =                                                            4
    Define ‘prime p = ~(p=1) /\ !x. x divides p ==> (x=1) \/ (x=p)‘;

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

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

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

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

the goal to its essence. Sometimes this proves the goal immediately. Often however, we are left with a goal that requires some study before one realizes what lemmas are needed to conclude the proof. Once these lemmas have been proven (or located in ancestor theories), PROVE_TAC can be invoked with them, with the expectation that it will find the right instantiations needed to finish the proof. (These two operations do not suffice to perform all proofs; in particular, our development will also need case analysis and induction.)

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

```
help "arithmeticTheory"
```

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

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

4.1 Divisibility

We start by proving a number of theorems about the divides relation. We will see that each of these initial theorems can be 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. The major reason that PROVE_TAC works so well is that divides is defined by means of an existential quantifier, and PROVE_TAC is quite good at automatically instantiating existentials in the course of proof. For a simple example, consider proving $\forall x.\ x$ divides 0. A new proposition to be proved is entered to the proof manager via "g", which starts a fresh goalstack:

```
- g `!x. x divides 0`;                                                    5

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

     : proofs
```

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

```
- e (ARW_TAC [divides]);                                                  6

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

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

```
- e (EXISTS_TAC ``0``);                                                   7

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

Then a simplification step finishes the proof.

```
- e (ARW_TAC []);                                                         8
OK..

Goal proved.
|- (x = 0) \/ (0 = 0)

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

What just happened here? The application of `ARW_TAC` to the goal decomposed it to an empty list of subgoals; in other words the goal was proved by `ARW_TAC`. Once a goal has been proved, it is popped off the goalstack, prettyprinted to the output, and the theorem becomes available for use by the level of the stack. When all the sub-goals required by *that* level are proven, the corresponding goal at that level can be proven too.

This 'unwinding' process continues until the stack is empty, or until it hits a goal with more than one remaining unproved subgoal. This process may be hard to visualize,[3] but that doesn't matter, since the goalstack was expressly written to allow the user to ignore such details.

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

9

```
- 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, as mentioned earlier, there is another: one can let PROVE_TAC expand the definition of divides and find the required instantiation for x' from the theorem MULT_CLAUSES.[4]

10

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

- e (PROVE_TAC [divides, MULT_CLAUSES]);

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

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

11

```
- val DIVIDES_0 = top_thm();

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

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

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

[4] You might like to try typing MULT_CLAUSES into the interactive loop to see exactly what it states.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
- g `!m n . m divides n ==> m <= n \/ (n = 0)`;                    12
>   ...

- e (ARW_TAC [divides]);

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

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

```
- e (Cases_on `m`);                                               13

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

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

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

```
- e (ARW_TAC []);                                                 14

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 []);                                                 15

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

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

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

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

```
- DB.match ["arithmetic"] ``SUC n * x``;                                    16

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

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

```
- DB.match [] ``SUC n * x = M``;                                                    17

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

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

```
- e (ARW_TAC [MULT_CLAUSES]);                                                       18

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

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

```
- b();                                                                              19
> val it =
    SUC n <= SUC n * x \/ (x = 0)

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

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


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

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

Now we can go forward and do case analysis on x. We will also make a compound tactic invocation, since we already know that we'll have to invoke `ARW_TAC` in both branches of the case split. This can be done using `THEN`. When t_1 `THEN` t_2 is applied

to a goal g, first t_1 is applied to g, giving a list of new subgoals, then t_2 is applied to each member of the list. All goals resulting from these applications of t_2 are gathered together and returned.

20
```
- e (Cases_on `x` THEN ARW_TAC [MULT_CLAUSES]);

OK..

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

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

21
```
- 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]);

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

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

4.1.1 Divisibility and factorial

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

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

A polished proof of *DIVIDES_FACT* is the following[5]:

[5]This and subsequent proofs use the theorems proved on page 40, which we've assumed are now part of the ML environment.

(*DIVIDES_FACT*) `!m n. 0 < m /\ m <= n ==> m divides (FACT n)`

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

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

22

```
- 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
```

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

23

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

- e (ARW_TAC [LESS_EQ_EXISTS]);

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

Now we induct on p:

```
- e (Induct_on `p`);                                                    24

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 []);                                                       25

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

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

```
- e (Cases_on `m`);                                                     26

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

    0 divides FACT 0
    ----------------------------------
      0 < 0
```

Here the first sub-goal goal has an assumption that is false. We can demonstrate this to the system by using the DECIDE function to prove a simple fact about arithmetic (namely, that no number x is less than itself), and then passing the resulting theorem to PROVE_TAC, which can combine this with the contradictory assumption.[6]

[6]Note how the interactive system prints out the proved theorem with [.] before the turnstile. This notation indicates that a theorem has an assumption (the false $0 < 0$ in this case).

```
- e (PROVE_TAC [DECIDE ``!x. ~(x < x)``]);                                27

OK..
Meson search level: ..

Goal proved.
 [.] |- 0 divides FACT 0

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

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

```
- e (ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);                         28

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();                                                                    29

> ...

- 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]);                                    30

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]);                                       31

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

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

```
e (ARW_TAC [LESS_EQ_EXISTS]);
e (Induct_on `p`);
(*1*)
e (ARW_TAC []);
e (Cases_on `m`);
(*1.1*)
e (PROVE_TAC [DECIDE ``!x. ~(x < x)``]);
(*1.2*)
e (ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);
(*2*)
e (ARW_TAC [FACT, ADD_CLAUSES]);
e (PROVE_TAC [DIVIDES_LMUL]);
```

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

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

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

```
- restart();                                                                    32

> ...

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

For many users, this would be the end of dealing with this proof: the tactic can now be packaged into an invocation of `prove`[7] 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]]
```

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

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

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

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

4.1.2 Divisibility and factorial (again!)

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

33

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

```
- b();                                                                    34
  ...

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

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

    m divides FACT n
    ------------------------------------
      0.  0 = n - m
      1.  0 < m
      2.  m <= n
```

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

```
- e (`m = n` by DECIDE_TAC);                                              35
OK..
1 subgoal:
> val it =
    m divides FACT n
    ------------------------------------
      0.  0 = n - m
      1.  0 < m
      2.  m <= n
      3.  m = n
```

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

```
- e (ARW_TAC []);                                                         36

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

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

```
‘?k.  m = SUC k‘ by <tactic>
```

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

When such situations occur, it is often best to start a new proof for the required lemma. This can be done simply by invoking "g" again. A new goalstack will be created and stacked upon the current one[8] and an overview of the extant proof attempts will be printed:

```
- g ‘!m. 0 < m ==> ?k. m = SUC k‘;                                    37

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


         Current goal:
         m divides FACT m
         ------------------------------------
           0.  0 = m - m
           1.  0 < m
           2.  m <= m

    1. Incomplete:
         Initial goal:
         !m. 0 < m ==> ?k. m = SUC k
```

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"[9] function.

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

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

```
- e (Cases THEN ARW_TAC []);                                              38

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

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

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

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

```
- p ();                                                                   39

> 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                                                   40
      PROVE_TAC [lem] THEN ARW_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);

OK..
Meson search level: ...

Goal proved.   ...

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

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]);   41

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]);                                   42

OK..

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

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

4.2 Primality

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

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

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

(*PRIME_2*) `prime 2` —

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

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

4.3 Existence of prime factors

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

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

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

43
```
- g `!n. ~(n = 1) ==> ?p. prime p /\ p divides n`;

- e (completeInduct_on `n`);

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

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

```
- e (ARW_TAC [] THEN Cases_on `prime n`);                                    44
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]);                                             45

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]);           46

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]);                47

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]);                                           48

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

The remaining goal can be clarified by simplification:

49

```
- 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

- 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:

50

```
- 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.

4.4 Euclid's theorem

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

A HOL rendition of the proof may be given as follows:

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

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

51
```
- g ‘!n. ?p. n < p /\ prime p‘;

- e (SPOSE_NOT_THEN STRIP_ASSUME_TAC);

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

Thus we have the hypothesis that all p beyond a certain unspecified n are not prime, and our task is to show that this cannot be. At this point we take advantage of Euclid's great inspiration and we build an explicit term from n. In the informal proof we are asked to 'consider' the term FACT $n + 1$.[10] 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:[11]

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

```
- PRIME_FACTOR;                                                    52

> val it = |- !n. ~(n = 1) ==> (?p. prime p /\ p divides n) : 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 HOL, one could do this in a so-called *forward* proof style (by proving $\vdash \neg(\texttt{FACT}\ n + 1 = 1)$ and then applying *modus ponens*, the result of which can then be used in the proof), or one could bring `th` into the proof and simplify it *in situ*. We choose the latter approach.

```
- e (MP_TAC (SPEC ‘‘FACT n + 1‘‘ PRIME_FACTOR));                   53

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 []);                                                  54

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

    ~(FACT n = 0)
    ------------------------------------
      !p. n < p ==> ~prime p
```

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

```
- e (ARW_TAC [FACT_LESS, DECIDE ‘‘!x. ~(x=0) = 0 < x‘‘]);          55

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

Notice the 'on-the-fly' use of `DECIDE` to provide an *ad hoc* rewrite. Looking at the remaining goal, one might think that our aim, to prove falsity, has been lost. But this

is not so: a goal $\neg M$ is equivalent to $M \supset$ `F`. We can quickly proceed to show that p `divides` (`FACT` n), and thus that $p = 1$, hence that p is not prime, at which point we are done. This can all be packaged into a single invocation of `PROVE_TAC`:

56

```
- 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 can allow a readable linear presentation that mirrors the informal proof. The following proves Euclid's theorem in the assertional style. We think it is fairly readable, certainly much more so than the standard tactic proof just given.[12]

(*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 []
```

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

4.5 Turning the script into a theory

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

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

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

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

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

1

```
$ cd examples/
$ ../bin/hol < euclid.sml
  ...

> val EUCLID = |- !n. ?p. n < p /\ prime p : thm
  ...

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

However, we now want to create a `euclidTheory` that we can load in other interactive sessions. So, our first step is to create a file `euclidScript.sml`, and to copy the body of `euclid.sml` into it.

The first non-comment line opens `arithmeticTheory`. However, when writing for the compiler, we need to explicitly mention the other HOL modules that we depend on. We must add

```
open HolKernel boolLib Parse bossLib
```

The next line that poses a difficulty is

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

While it is legitimate to type expressions directly into the interactive system, the compiler requires that every top-level phrase be a declaration. We satisfy this requirement

by altering this line into a "do nothing" declaration that does not record the result of the expression:

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

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

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

and at the end of the file:

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

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

2

```
$ mkdir euclid
$ mv euclidScript.sml euclid
$ cd euclid
$ ../../bin/Holmake
Analysing euclidScript.sml
Trying to create directory .HOLMK for dependency files
Compiling euclidScript.sml
Linking euclidScript.uo to produce theory-builder executable
<<HOL message: Created theory "euclid".>>
Definition has been stored under "divides_def".
Definition has been stored under "prime_def".
Meson search level: .....
Meson search level: ................
 ...
Exporting theory "euclid" ... done.
Analysing euclidTheory.sml
Analysing euclidTheory.sig
Compiling euclidTheory.sig
Compiling euclidTheory.sml
```

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

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

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

4.6 Summary

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

- Even though the proof of Euclid's theorem is short and easy to understand when presented informally, a perhaps surprising amount of support development was required to set the stage for Euclid's classic argument.
- The proof support offered by `bossLib` (`RW_TAC`, `PROVE_TAC`, `DECIDE_TAC`, `DECIDE`, `Cases_on`, `Induct_on`, and the "`by`" construct) was nearly complete for this example: it was rarely necessary to resort to lower-level tactics.
- Simplification is a workhorse tactic; even when an automated reasoner like `PROVE_TAC` is used, its application has often been set up by some exploratory simplifications. It therefore pays to become familiar with the strengths and weaknesses of the simplifier.
- A common problem with interactive proof systems is dealing with hypotheses. Often `PROVE_TAC` and the "`by`" construct allow the use of hypotheses without directly resorting to indexing into them (or naming them, which amounts to the same thing). This is desirable, since the hypotheses are notionally a *set*, and moreover, experience has shown that profligate indexing into hypotheses results in hard-to-maintain proof scripts. However, it can be clumsy to work with a large set of hypotheses, in which case the following approaches may be useful.

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

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

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

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

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

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

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

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

Chapter 5

Example: a Simple Parity Checker

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

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

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

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

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

5.1 Introduction

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

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

The file `PARITY.sml` contains the HOL sessions in this chapter; the files `RESET_REG.sml` and `RESET_PARITY.sml` contain the solutions to the exercises described in Section 5.5.

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

The sessions of this example comprise the specification and verification of a device that computes the parity of a sequence of bits. More specifically, a detailed verification is given of a device with an input `in`, an output `out` and the specification that the nth

output on out is T if and only if there have been an even number of T's input on in. A theory named PARITY is constructed; this contains the specification and verification of the device. All the ML input in the boxes below can be found in the file parity/PARITY.sml. It is suggested that the reader interactively input this to get a 'hands on' feel for the example.

5.2 Specification

The first step is to start up the HOL system. We again use <holdir>/bin/hol. The ML prompt is -, so lines beginning with - are typed by the user and other lines are the system's response.

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

1
```
- val PARITY_def = Define‘
    (PARITY 0 f = T) /\
    (PARITY(SUC n) f = if f(SUC n) then ~(PARITY n f) else PARITY n f)‘;
Definition has been stored under "PARITY_def".
> val PARITY_def =
    |- (!f. PARITY 0 f = T) /\
       !n f. PARITY (SUC n) f =
             (if f (SUC n) then ~PARITY n f else PARITY n f)
    : thm
```

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

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

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

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

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

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

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

and

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

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

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

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

2

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

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

3

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

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

4

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

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

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

5

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

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

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

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

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

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

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

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

The ML function

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

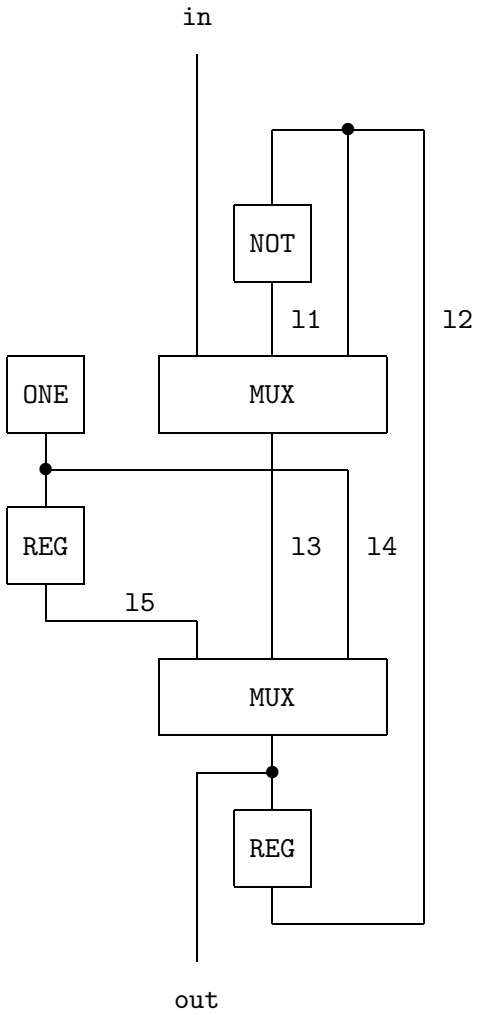
returns the theorem just proved (i.e. on the top of the theorem stack) in the current theory, and we bind this to the ML name `UNIQUENESS_LEMMA`.

```
- val UNIQUENESS_LEMMA = top_thm();                                       7
> val UNIQUENESS_LEMMA =
    |- !inp out.
         (out 0 = T) /\
         (!t. out (SUC t) = (if inp (SUC t) then ~out t else out t)) ==>
         !t. out t = PARITY t inp
    : thm
```

5.3 Implementation

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

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



The devices making up this schematic will be modelled with predicates [5]. For example, the predicate `ONE` is true of a signal `out` if for all times `t` the value of `out` is `T`.

```
- val ONE_def = Define ‘ONE(out:num->bool) = !t. out t = T‘;                    8
Definition stored under "ONE_def".
> val ONE_def = |- !out. ONE out = !t. out t = T : thm
```

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

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

```
- val NOT_def = Define‘NOT(inp, out:num->bool) = !t. out t = ~(inp t)‘;        9
Definition stored under "NOT_def".
> val NOT_def = |- !inp out. NOT (inp,out) = !t. out t = ~inp t : Thm.thm
```

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

```
- val MUX_def = Define‘                                                        10
    MUX(sw,in1,in2,out:num->bool) =
      !t. out t = if sw t then in1 t else in2 t‘;
Definition stored under "MUX_def".
> val MUX_def =
    |- !sw in1 in2 out.
         MUX (sw,in1,in2,out) = !t. out t = (if sw t then in1 t else in2 t)
    : thm
```

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

```
- val REG_def =                                                                11
    Define ‘REG(inp,out:num->bool) =
              !t. out t = if (t=0) then F else inp(t-1)‘;
Definition stored under "REG_def".
> val REG_def =
    |- !inp out. REG (inp,out) = !t. out t =
                  (if t = 0 then F else inp (t - 1))
    : thm
```

The schematic diagram above can be represented as a predicate by conjoining the relations holding between the various signals and then existentially quantifying the internal lines. This technique is explained elsewhere (e.g. see [3, 5]).

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

12
```
- val PARITY_IMP_def = Define
   ‘PARITY_IMP(inp,out) =
      ?l1 l2 l3 l4 l5.
        NOT(l2,l1) /\ MUX(inp,l1,l2,l3) /\ REG(out,l2) /\
        ONE l4     /\ REG(l4,l5)        /\ MUX(l5,l3,l4,out)‘;
Definition stored under "PARITY_IMP_def".
> val PARITY_IMP_def =
    |- !inp out.
         PARITY_IMP (inp,out) =
         ?l1 l2 l3 l4 l5.
           NOT (l2,l1) /\ MUX (inp,l1,l2,l3) /\ REG (out,l2) /\ ONE l4 /\
           REG (l4,l5) /\ MUX (l5,l3,l4,out)
    : thm
```

5.4 Verification

The following theorem will eventually be proved:

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

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

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

13
```
- g ‘!inp out.
        PARITY_IMP(inp,out) ==>
        (out 0 = T) /\
        !t. out(SUC t) = if inp(SUC t) then ~(out t) else out t‘;
> val it =
    Proof manager status: 2 proofs.
    2. Completed: ...
    1. Incomplete:
         Initial goal:
         !inp out.
           PARITY_IMP (inp,out) ==>
           (out 0 = T) /\
           !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

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

14

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

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

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

15

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

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

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

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

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

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

16

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

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

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

17
```
- e (RW_TAC arith_ss []);
OK..

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

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

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

18
```
- val PARITY_LEMMA = top_thm ();
> val PARITY_LEMMA =
    |- !inp out.
         PARITY_IMP (inp,out) ==>
         (out 0 = T) /\
         !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

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

19
```
- restart()
> ...
- e (PURE_REWRITE_TAC [PARITY_IMP_def, ONE_def, NOT_def,
                       MUX_def, REG_def] THEN
     REPEAT STRIP_TAC THENL [
       PROVE_TAC [],
       PAT_ASSUM ``!t. out t = X t``
                 (fn th => REWRITE_TAC [SPEC ``SUC t`` th]) THEN
       RW_TAC arith_ss []
     ]);
<<HOL message: inventing new type variable names: 'a, 'b.>>
OK..
Meson search level: .....
> val it =
    Initial goal proved.
    |- !inp out.
         PARITY_IMP (inp,out) ==>
         (out 0 = T) /\
         !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

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

```
- val PARITY_CORRECT = prove(                                              20
    ‘‘!inp out. PARITY_IMP(inp,out) ==> (!t. out t = PARITY t inp)‘‘,
    REPEAT STRIP_TAC THEN MATCH_MP_TAC UNIQUENESS_LEMMA THEN
    MATCH_MP_TAC PARITY_LEMMA THEN ASM_REWRITE_TAC []);
> val PARITY_CORRECT =
    |- !inp out. PARITY_IMP (inp,out) ==> !t. out t = PARITY t inp
```

This completes the proof of the parity checking device.

5.5 Exercises

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

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

5.5.1 Exercise 1

Using *only* the devices ONE, NOT, MUX and REG defined in Section 5.3, design and verify a register RESET_REG with an input in, reset line reset, output out and behaviour specified as follows.

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

This is formalized in HOL by the definition:

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

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

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

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

[4] i.e. a definition of the same form as that of PARITY_IMP on page 74.

5.5.2 Exercise 2

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

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

2. Design an implementation of this specification built using *only* the devices `ONE`, `NOT`, `MUX` and `REG` defined in Section 5.3.

3. Verify the correctness of your implementation in HOL.

Chapter 6

Example: Combinatory Logic

6.1 Introduction

This small case study is a formalisation of (variable-free) combinatory logic. This logic is of foundational importance in theoretical computer science, and has a very rich theory. The example builds principally on a development done by Tom Melham. The complete script for the development is available as `clScript.sml` in the `examples/ind_def` directory of the distribution. It is self-contained and so includes the answers to the exercises set at the end of this document.

6.2 The type of combinators

The first thing we need to do is define the type of *combinators*. There are just two of these, K and S, but we also need to be able to *combine* them, and for this we need to introduce the notion of application. For lack of a better ASCII symbol, we will use the hash (#) to represent this in the logic:

```
- Hol_datatype `cl = K | S | # of cl => cl`;                    1
> val it = () : unit
```

We also want the # to be an infix, so we set its fixity to be a tight left-associative infix:

```
- set_fixity "#" (Infixl 1100);                                 2
> val it = () : unit
```

Finally, there's one last piece of book-keeping to be done for our new type. The datatype package defines the constructors in theorems of their own, and the name of the theorem stored to disk is the same as the name of the constructor. SML doesn't allow # to be an identifier so we must change the name of the theorem. We do this with the function `set_MLname`. The first parameter to the function is the old name, and the second is the new name.

```
- set_MLname "#" "HASH";                                        3
> val it = () : unit
```

6.3 Combinator reductions

Combinatory logic is the study of how values of this type can evolve given various rules describing how they change. Therefore, our next step is to define the reductions that combinators can undergo. There are two basic rules:

$$\begin{array}{lcl} \mathsf{K}\ x\ y & \rightarrow & x \\ \mathsf{S}\ f\ g\ x & \rightarrow & (fx)(gx) \end{array}$$

Here, in our description outside of HOL, we use juxtaposition instead of the #. Further, juxtaposition is also left-associative, so that $\mathsf{K}\ x\ y$ should be read as $\mathsf{K}\ \#\ x\ \#\ y$ which is in turn $(\mathsf{K}\ \#\ x)\ \#\ y$.

Given a term in the logic, we want these reductions to be able to fire at any point, not just at the top level, so we need two further congruence rules:

$$\frac{x \rightarrow x'}{x\ y \rightarrow x'\ y}$$

$$\frac{y \rightarrow y'}{x\ y \rightarrow x\ y'}$$

In HOL, we can capture this relation with an inductive definition. First we set our arrow symbol up as an infix to make everything that bit prettier:

```
- set_fixity "-->" (Infix(NONASSOC, 510));                                4
> val it = () : unit
```

(By choosing to make our arrow symbol non-associative, we make it a parse error to write `x --> y --> z`. It would be nice to be able to write this and have it mean `x --> y /\ y --> z`, but this is not presently possible with the HOL parser.)

Our next step is to actually define the relation. The function for doing this returns three separate theorems, so we bind each separately:

5
```
val (redn_rules, redn_ind, redn_cases) =
  Hol_reln
    `(!x y f. x --> y   ==>    f # x --> f # y) /\
     (!f g x. f --> g   ==>    f # x --> g # x) /\
     (!x y.   K # x # y --> x) /\
     (!f g x. S # f # g # x --> (f # x) # (g # x))`;
> val redn_rules =
    |- (!x y f. x --> y ==> f # x --> f # y) /\
       (!f g x. f --> g ==> f # x --> g # x) /\
       (!x y. K # x # y --> x) /\
       !f g x. S # f # g # x --> f # x # (g # x) : thm
  val redn_ind =
    |- !-->'.
         (!x y f. -->' x y ==> -->' (f # x) (f # y)) /\
         (!f g x. -->' f g ==> -->' (f # x) (g # x)) /\
         (!x y. -->' (K # x # y) x) /\
         (!f g x. -->' (S # f # g # x) (f # x # (g # x))) ==>
         !a0 a1. a0 --> a1 ==> -->' a0 a1 : thm
  val redn_cases =
    |- !a0 a1.
         a0 --> a1 =
         (?x y f. (a0 = f # x) /\ (a1 = f # y) /\ x --> y) \/
         (?f g x. (a0 = f # x) /\ (a1 = g # x) /\ f --> g) \/
         (?y. a0 = K # a1 # y) \/
         ?f g x. (a0 = S # f # g # x) /\ (a1 = f # x # (g # x))
    : thm
```

The induction theorem `redn_ind` looks a little strange because the induction predicate is given the name `-->'`. We can change the name to make things prettier with the function `RENAME_VARS_CONV`, a conversion:

6
```
- val redn_ind = CONV_RULE (RENAME_VARS_CONV ["P"]) redn_ind;
> val redn_ind =
    |- !P.
         (!x y f. P x y ==> P (f # x) (f # y)) /\
         (!f g x. P f g ==> P (f # x) (g # x)) /\
         (!x y. P (K # x # y) x) /\
         (!f g x. P (S # f # g # x) (f # x # (g # x))) ==>
         !a0 a1. a0 --> a1 ==> P a0 a1 : thm
```

In addition to proving these three theorems for us, the inductive definitions package has also saved them to disk. Unfortunately, it does so in a way that generates names that will be unacceptable. Only experience tells you this at this stage, but when (if) we later export and compile the theory we will get nasty errors. So, we need to set the ML names for the theorem defining the constant `-->` and for the theorems that have been given the names `-->_rules`, `-->_ind` and `-->_cases`.[1]

[1]Normally, `-->` would be a fine name for an ML identifier, but the problem here is that when the theory is compiled, the identifier `-->` is already declared as an infix. Names like `-->_rules` are always

```
- app (uncurry set_MLname) [                                              7
    ("-->", "redn"), ("-->_rules", "redn_rules"),
    ("-->_ind", "redn_ind"), ("-->_cases", "redn_cases")
  ];
> val it = () : unit
```

Now, using our theorem `redn_rules` we can demonstrate single steps of our reduction relation:

```
- PROVE [redn_rules] ``S # (K # x # x) --> S # x``;                       8
Meson search level: ...
> val it = |- S # (K # x # x) --> S # x : thm
```

The system we have just defined is as powerful as the λ-calculus, Turing machines, and all the other standard models of computation.

One useful result about the combinatory logic is that it is *confluent*. Consider the term $\mathsf{S}\ z\ (\mathsf{K}\ \mathsf{K})\ (\mathsf{K}\ y\ x)$. It can make two reductions, to $\mathsf{S}\ z\ (\mathsf{K}\ \mathsf{K})\ y$ and also to $(z\ (\mathsf{K}\ y\ x))\,(\mathsf{K}\ \mathsf{K}\ (\mathsf{K}\ y\ x))$. Do these two choices of reduction mean that from this point on the terms have two completely separate histories? Roughly speaking, to be confluent means that the answer to this question is *no*.

6.4 Transitive closure and confluence

A notion crucial to that of confluence is that of *transitive closure*. We have defined a system that evolves by specifying how an algebraic value can evolve into possible successor values in one step. The natural next question is to ask for a characterisation of evolution over one or more steps of the $\rightarrow$ relation.

In fact, we will define a relation that holds between two values if the second can be reached from the first in zero or more steps. This is the *reflexive, transitive closure* of our original relation. However, rather than tie our new definition to our original relation, we will develop this notion independently and prove a variety of results that are true of any system, not just our system of combinatory logic.

So, we begin our abstract digression with another inductive definition. Our new constant is RTC, such that $\mathsf{RTC}\ R\ x\ y$ is true if it is possible to get from x to y with zero or more "steps" of the R relation. (The standard notation for $\mathsf{RTC}\ R$ is R^*.) We can express this idea with just two rules. The first

$$\frac{}{\mathsf{RTC}\ R\ x\ x}$$

says that it's always possible to get from x to x in zero or more steps. The second

$$\frac{R\ x\ y \qquad \mathsf{RTC}\ R\ y\ z}{\mathsf{RTC}\ R\ x\ z}$$

bad because they attempt to mix symbolic and alpha-numeric characters.

says that if you can take a single step from x to y, and then take zero or more steps to get y to z, then it's possible to take zero or more steps to get between x and z. The realisation of these rules in HOL is again straightforward.

(As it happens, RTC is already a defined constant in the context we're working in (it is found in `relationTheory`), so we'll hide it from view before we begin. We thus avoid messages telling us that we are inputting ambiguous terms. The ambiguities would always be resolved in the favour of more recent definition, but the warnings are annoying.)

9

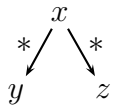
```
val _ = hide "RTC";

val (RTC_rules, RTC_ind, RTC_cases) =
  Hol_reln ‘
    (!x.     RTC R x x) /\
    (!x y z. R x y /\ RTC R y z ==> RTC R x z)‘;
<<HOL message: inventing new type variable names: 'a>>
> val RTC_rules =
    |- !R. (!x. RTC R x x) /\
           !x y z. R x y /\ RTC R y z ==> RTC R x z : thm
  val RTC_ind =
    |- !R RTC'.
         (!x. RTC' x x) /\
         (!x y z. R x y /\ RTC' y z ==> RTC' x z) ==>
         !a0 a1. RTC R a0 a1 ==> RTC' a0 a1 : thm
  val RTC_cases =
    |- !R a0 a1. RTC R a0 a1 = (a1 = a0) \/
                               ?y. R a0 y /\ RTC R y a1 : thm
```

Now let us go back to the notion of confluence. We want this to mean something like: "though a system may take different paths in the short-term, those two paths can always end up in the same place". This suggests that we define confluent thus:

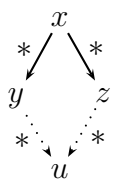
10

```
- val confluent_def = Define
    ‘confluent R =
       !x y z. RTC R x y /\ RTC R x z ==>
               ?u. RTC R y u /\ RTC R z u‘;
```

This property states of R that we can "complete the diamond"; if we have



then there must be a u such that



One nice property of confluent relations is that from any one starting point they produce no more than one *normal form*, where a normal form is a value from which no further steps can be taken.

```
- val normform_def = Define‘normform R x = !y. ~(R x y)‘;                    11
<<HOL message: inventing new type variable names: 'a, 'b>>
Definition has been stored under "normform_def".
> val normform_def = |- !R x. normform R x = !y. ~R x y : thm
```

In other words, a system has an R-normal form at x if there are no connections via R to any other values. (We could have written `~?y. R x y` as our RHS for the definition above.)

We can now prove the following:

```
- g ‘!R. confluent R ==>                                                     12
         !x y z.
           RTC R x y /\ normform R y /\
           RTC R x z /\ normform R z ==> (y = z)‘;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !R.
           confluent R ==>
           !x y z.
             RTC R x y /\ normform R y /\
             RTC R x z /\ normform R z ==> (y = z)
```

We rewrite with the definition of confluence:

```
- e (RW_TAC std_ss [confluent_def]);                                         13
OK..
1 subgoal:
> val it =
    y = z
    ------------------------------------
      0.  !x y z. RTC R x y /\ RTC R x z ==>
                  ?u. RTC R y u /\ RTC R z u
      1.  RTC R x y
      2.  normform R y
      3.  RTC R x z
      4.  normform R z
```

Our confluence property is now assumption 0, and we can use it to infer that there is a u at the base of the diamond:

```
- e (‘?u. RTC R y u /\ RTC R z u‘ by PROVE_TAC []);                    14
OK..
Meson search level: .........
1 subgoal:
> val it =
    y = z
    ------------------------------------
      0.  !x y z. RTC R x y /\ RTC R x z ==>
                  ?u. RTC R y u /\ RTC R z u
      1.  RTC R x y
      2.  normform R y
      3.  RTC R x z
      4.  normform R z
      5.  RTC R y u
      6.  RTC R z u
```

So, from y we can take zero or more steps to get to u and similarly from z. But, we also know that we're at an R-normal form at both y and z. We can't take any steps at all from these values. We can conclude both that $u = y$ and $u = z$, and this in turn means that $y = z$, which is our goal. So we can finish with

```
- e (PROVE_TAC [normform_def, RTC_cases]);                            15
OK..
Meson search level: ..........

Goal proved. [...]
> val it =
    Initial goal proved.
    |- !R.
         confluent R ==>
         !x y z.
           RTC R x y /\ normform R y /\
           RTC R x z /\ normform R z ==> (y = z)
```

Packaged up so as to remove the sub-goal package commands, we can prove and save the theorem for future use by:

```
val confluent_normforms_unique = store_thm(                           16
  "confluent_normforms_unique",
  ‘‘!R. confluent R ==>
        !x y z. RTC R x y /\ normform R y /\
                RTC R x z /\ normform R z ==> (y = z)‘‘,
  RW_TAC std_ss [confluent_def] THEN
  ‘?u. RTC R y u /\ RTC R z u‘ by PROVE_TAC [] THEN
  PROVE_TAC [normform_def, RTC_cases]);
```

$\cdots\diamond\cdots$

Clearly confluence is a nice property for a system to have. The question is how we might manage to prove it. Let's start by defining the diamond property that we used in the definition of confluence.

17
```
- val diamond_def = Define
    ‘diamond R = !x y z. R x y /\ R x z ==>
                         ?u. R y u /\ R z u‘;
<<HOL message: inventing new type variable names: ’a>>
Definition has been stored under "diamond_def".
> val diamond_def =
    |- !R.
         diamond R = !x y z. R x y /\ R x z ==>
                             ?u. R y u /\ R z u
     : thm
```

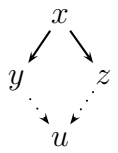
Now we clearly have that confluence of a relation is equivalent to the reflexive, transitive closure of that relation having the diamond property.

18
```
val confluent_diamond_RTC = store_thm(
  "confluent_diamond_RTC",
  ‘‘!R. confluent R = diamond (RTC R)‘‘,
  RW_TAC std_ss [confluent_def, diamond_def]);
```

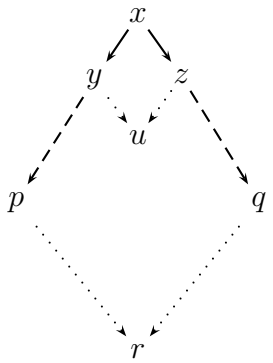
So far so good. How then do we show the diamond property for RTC R? The answer that leaps to mind is to hope that if the original relation has the diamond property, then maybe the reflexive and transitive closure will too. The theorem we want is

$$\mathsf{diamond}\ R \supset \mathsf{diamond}\ (\mathsf{RTC}\ R)$$

Graphically, this is hoping that from

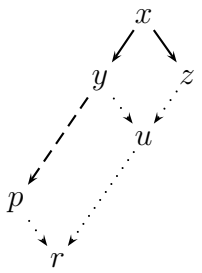


we will be able to conclude



where the dashed lines indicate that these steps (from x to p, for example) are using RTC R. The presence of two instances of RTC R is an indication that this proof will

require two inductions. With the first we will prove



In other words, we want to show that if we take one step in one direction (to z) and many steps in another (to p), then the diamond property for R will guarantee us the existence of r, to which will we be able to take many steps from both p and z.

We take some care to state the goal so that after stripping away the outermost assumption (that R has the diamond property), it will match the induction principle for RTC.[2]

19

```
- g `!R. diamond R ==>
         !x p. RTC R x p ==>
               !z. R x z ==>
                   ?u. RTC R p u /\ RTC R z u`;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !R.
           diamond R ==>
           !x p. RTC R x p ==> !z. R x z ==>
                                  ?u. RTC R p u /\ RTC R z u
```

First, we strip away the diamond property assumption (two things need to be stripped: the outermost universal quantifier and the antecedent of the implication):

20

```
- e (GEN_TAC THEN STRIP_TAC);
OK..
1 subgoal:
> val it =
    !x p. RTC R x p ==> !z. R x z ==> ?u. RTC R p u /\ RTC R z u
    ------------------------------------
      diamond R
```

Now we can use the induction principle. We use the higher-order backward chaining rule, `HO_MATCH_MP_TAC`, which takes a theorem of the form $\vdash P \supset Q$, tries to instantiate

[2] In this and subsequent proofs using the sub-goal package, we will present the proof manager as if the goal to be proved is the first ever on this stack. In other words, we have done a `dropn 1;` after every successful proof to remove the evidence of the old goal. In practice, there is no harm in leaving these goals on the proof manager's stack.

it to make it $\vdash \ P' \supset Q'$, such that Q' is the same as the goal to be proved, and then requires the user to prove P'.

```
- e (HO_MATCH_MP_TAC RTC_ind);                                          21
OK..
1 subgoal:
> val it =
    (!x z. R x z ==> ?u. RTC R x u /\ RTC R z u) /\
    !x y z.
      R x y /\ (!z'. R y z' ==> ?u. RTC R z u /\ RTC R z' u) ==>
      !z'. R x z' ==> ?u. RTC R z u /\ RTC R z' u
    ------------------------------------
      diamond R
```

Let's strip the goal as much as possible with the aim of making what remains to be proved easier to see:

```
- e (REPEAT STRIP_TAC);                                                 22
OK..
2 subgoals:
> val it =
    ?u. RTC R z u /\ RTC R z' u
    ------------------------------------
      0.  diamond R
      1.  R x y
      2.  !z'. R y z' ==> ?u. RTC R z u /\ RTC R z' u
      3.  R x z'

    ?u. RTC R x u /\ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  R x z
```

This first goal is easy. It corresponds to the case where the many steps from x to p are actually no steps at all, and p and x are actually the same place. In the other direction, x has taken one step to z, and we need to find somewhere reachable in zero or more steps from both x and z. Given what we know so far, the only candidate is z itself. In fact, we don't even need to provide this witness explicitly. PROVE_TAC will find it for us, as long as we tell it what the rules governing RTC are:

```
- e (PROVE_TAC [RTC_rules]);                                          23
OK..
Meson search level: .....

Goal proved. [...]
Remaining subgoals:
> val it =
    ?u. RTC R z u /\ RTC R z' u
    ------------------------------------
      0.  diamond R
      1.  R x y
      2.  !z'. R y z' ==> ?u. RTC R z u /\ RTC R z' u
      3.  R x z'
```

And what of this remaining goal? Assumptions one and three between them are the top of an R-diamond. Let's use the fact that we have the diamond property for R and infer that there exists a v to which y and z' can both take single steps:

```
- e (`?v. R y v /\ R z' v` by PROVE_TAC [diamond_def]);               24
OK..
Meson search level: ............
1 subgoal:
> val it =
    ?u. RTC R z u /\ RTC R z' u
    ------------------------------------
      0.  diamond R
      1.  R x y
      2.  !z'. R y z' ==> ?u. RTC R z u /\ RTC R z' u
      3.  R x z'
      4.  R y v
      5.  R z' v
```

Now we can apply our induction hypothesis (assumption 2) to complete the long, lopsided strip of the diamond. We will conclude that there is a u such that RTC $R\ z\ u$ and RTC $R\ v\ u$. We actually need a u such that RTC $R\ z'\ u$, but because there is a single R-step between z' and v we have that as well. All we need to provide PROVE_TAC is the rules for RTC:

```
- e (PROVE_TAC [RTC_rules]);                                          25
OK..
Meson search level: .......

Goal proved. [...]
> val it =
    Initial goal proved.
    |- !R.
         diamond R ==> !x p. RTC R x p ==>
                        !z. R x z ==> ?u. RTC R p u /\ RTC R z u
```

Again we can (and should) package up the lemma, avoiding the sub-goal package commands:

```
val R_RTC_diamond = store_thm(                                          26
  "R_RTC_diamond",
  ‘‘!R. diamond R ==>
        !x p. RTC R x p ==>
              !z. R x z ==>
                  ?u. RTC R p u /\ RTC R z u‘‘,
  GEN_TAC THEN STRIP_TAC THEN HO_MATCH_MP_TAC RTC_ind THEN
  REPEAT STRIP_TAC THENL [
    PROVE_TAC [RTC_rules],
    ‘?v. R y v /\ R z’ v‘ by PROVE_TAC [diamond_def] THEN
    PROVE_TAC [RTC_rules]
  ]);
```

$\cdots\diamond\cdots$

Now we can move on to proving that if R has the diamond property, so too does RTC R. We want to prove this by induction again. It's very tempting to state the goal as the obvious

$$\mathsf{diamond}\ R \supset \mathsf{diamond}\,(\mathsf{RTC}\ R)$$

but doing so will actually make it harder to apply the induction principle when the time is right. Better to start out with a statement of the goal that is very near in form to the induction princple. So, we manually expand the meaning of diamond and state our next goal thus:

```
- g ‘!R. diamond R ==> !x y. RTC R x y ==>                              27
                             !z. RTC R x z ==>
                                 ?u. RTC R y u /\ RTC R z u‘;
<<HOL message: inventing new type variable names: ’a>>
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !R.
           diamond R ==>
           !x y. RTC R x y ==> !z. RTC R x z ==>
                                  ?u. RTC R y u /\ RTC R z u
```

Again we strip the diamond property assumption, apply the induction principle, and strip repeatedly:

28
```
- e (GEN_TAC THEN STRIP_TAC THEN HO_MATCH_MP_TAC RTC_ind THEN
     REPEAT STRIP_TAC);
OK..
2 subgoals:
> val it =
    ?u. RTC R z u /\ RTC R z' u
    ------------------------------------
      0.  diamond R
      1.  R x y
      2.  !z'. RTC R y z' ==> ?u. RTC R z u /\ RTC R z' u
      3.  RTC R x z'

    ?u. RTC R x u /\ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  RTC R x z
```

The first goal is again an easy one, corresponding to the case where the trip from x to y has been one of no steps whatsoever.

29
```
- e (PROVE_TAC [RTC_rules]);
OK..
Meson search level: ...

Goal proved. [...]

Remaining subgoals:
> val it =
    ?u. RTC R z u /\ RTC R z' u
    ------------------------------------
      0.  diamond R
      1.  R x y
      2.  !z'. RTC R y z' ==> ?u. RTC R z u /\ RTC R z' u
      3.  RTC R x z'
```

This goal is very similar to the one we saw earlier. We have the top of a ("lop-sided") diamond in assumptions 1 and 3, so we can infer the existence of a common destination for y and z':

```
- e (‘?v. RTC R y v /\ RTC R z’ v‘                                    30
       by PROVE_TAC [R_RTC_diamond]);
OK..
Meson search level: ............
1 subgoal:
> val it =
    ?u. RTC R z u /\ RTC R z’ u
    ------------------------------------
      0.  diamond R
      1.  R x y
      2.  !z’. RTC R y z’ ==> ?u. RTC R z u /\ RTC R z’ u
      3.  RTC R x z’
      4.  RTC R y v
      5.  RTC R z’ v
```

At this point in the last proof we were able to finish it all off by just appealing to the rules for RTC. This time it is not quite so straightforward. When we use the induction hypothesis (assumption 2), we can conclude that there is a u to which both z and v can connect in zero or more steps, but in order to show that this u is reachable from z', we need to be able to conclude RTC R z' u when we know that RTC R z' v (assumption 5 above) and RTC R v u (our consequence of the inductive hypothesis). We leave the proof of this general result as an exercise, and here assume that it is already proved as the theorem RTC_RTC.

```
- e (PROVE_TAC [RTC_rules, RTC_RTC]);                                31
Meson search level: .......

Goal proved. [...]
> val it =
    Initial goal proved.
    |- !R.
         diamond R ==>
         !x y. RTC R x y ==> !z. RTC R x z ==>
                                 ?u. RTC R y u /\ RTC R z u
```

We can package this result up as a lemma and then prove the prettier version directly:

```
val diamond_RTC_lemma = prove(                                          32
  ‘‘!R.
       diamond R ==>
       !x y. RTC R x y ==> !z. RTC R x z ==>
                              ?u. RTC R y u /\ RTC R z u‘‘,
  GEN_TAC THEN STRIP_TAC THEN HO_MATCH_MP_TAC RTC_ind THEN
  REPEAT STRIP_TAC THENL [
    PROVE_TAC [RTC_rules],
    ‘?v. RTC R y v /\ RTC R z’ v‘
       by PROVE_TAC [R_RTC_diamond] THEN
    PROVE_TAC [RTC_RTC, RTC_rules]
  ]);
val diamond_RTC = store_thm(
  "diamond_RTC",
  ‘‘!R. diamond R ==> diamond (RTC R)‘‘,
  PROVE_TAC [diamond_def,diamond_RTC_lemma]);
```

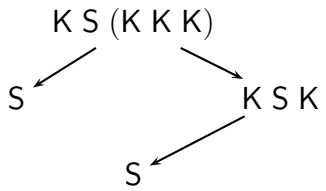
6.5 Back to combinators

Now, we are in a position to return to the real object of study and prove confluence for combinatory logic. We have done an abstract development and established that

$$\begin{array}{rcl} \mathsf{diamond}\ R & \supset & \mathsf{diamond}\,(\mathsf{RTC}\ R) \\ & & \wedge \\ & & \mathsf{diamond}\,(\mathsf{RTC}\ R) \quad \equiv \quad \mathsf{confluent}\ R \end{array}$$

(We have also established a couple of other useful results along the way.)

Sadly, it just isn't the case that $\rightarrow$, our one-step relation for combinators, has the diamond property. A counter-example is $\mathsf{K}\ \mathsf{S}\ (\mathsf{K}\ \mathsf{K}\ \mathsf{K})$. Its possible evolution can be described graphically:



If we had the diamond property, it should be possible to find a common destination for $\mathsf{K}\ \mathsf{S}\ \mathsf{K}$ and S. However, S doesn't admit any reductions whatsoever, so there isn't a common destination.[3]

This is a problem. We are going to have to take another approach. We will define another reduction strategy (*parallel reduction*), and prove that its reflexive, transitive closure is actually the same relation as our original's reflexive and transitive closure.

[3] In fact our counter-example is more complicated than necessary. The fact that $\mathsf{K}\ \mathsf{S}\ \mathsf{K}$ has a reduction to the normal form S also acts as a counter-example. Can you see why?

Then we will also show that parallel reduction has the diamond property. This will establish that its reflexive, transitive closure has it too. Then, because they are the same relation, we will have that the reflexive, transitive closure of our original relation has the diamond property, and therefore, our original relation will be confluent.

6.5.1 Parallel reduction

Our new relation allows for any number of reductions to occur in parallel. We use the `-||->` symbol to indicate parallel reduction because of its own parallel lines:

```
- set_fixity "-||->"  (Infix(NONASSOC, 510));                    33
> val it = () : unit
```

Then we can define parallel reduction itself. The rules look very similar to those for $\rightarrow$. The difference is that we allow the reflexive transition, and say that an application of $x\ u$ can be transformed to $y\ v$ if there are transformations taking x to y and u to v. This is why we must have reflexivity incidentally. Without it, a term like $(\mathsf{K}\ x\ y)\,\mathsf{K}$ couldn't reduce because while the LHS of the application $(\mathsf{K}\ x\ y)$ can reduce, its RHS (K) can't.

34
```
- val (predn_rules, predn_ind, predn_cases) =
    Hol_reln
      `(!x. x -||-> x) /\
       (!x y u v. x -||-> y /\ u -||-> v
                          ==>
                  x # u -||-> y # v) /\
       (!x y. K # x # y -||-> x) /\
       (!f g x. S # f # g # x -||-> (f # x) # (g # x))`;
> val predn_rules =
    |- (!x. x -||-> x) /\
       (!x y u v. x -||-> y /\ u -||-> v ==> x # u -||-> y # v) /\
       (!x y. K # x # y -||-> x) /\
       !f g x. S # f # g # x -||-> f # x # (g # x) : thm
  val predn_ind =
    |- !-||->'.
         (!x. -||->' x x) /\
         (!x y u v. -||->' x y /\ -||->' u v ==>
                    -||->' (x # u) (y # v)) /\
         (!x y. -||->' (K # x # y) x) /\
         (!f g x. -||->' (S # f # g # x) (f # x # (g # x))) ==>
         !a0 a1. a0 -||-> a1 ==> -||->' a0 a1 : thm
  val predn_cases =
    |- !a0 a1.
         a0 -||-> a1 =
         (a1 = a0) \/
         (?x y u v. (a0 = x # u) /\ (a1 = y # v) /\
                    x -||-> y /\ u -||-> v) \/
         (?y. a0 = K # a1 # y) \/
         ?f g x. (a0 = S # f # g # x) /\ (a1 = f # x # (g # x))
    : thm
```

We again have an induction principle that looks bizarre because of the choice of variable name, so we rename the bound variables.

35
```
- val predn_ind =
    CONV_RULE (RENAME_VARS_CONV ["P"]) predn_ind;
> val predn_ind =
    |- !P.
         (!x. P x x) /\
         (!x y u v. P x y /\ P u v ==> P (x # u) (y # v)) /\
         (!x y. P (K # x # y) x) /\
         (!f g x. P (S # f # g # x) (f # x # (g # x))) ==>
         !a0 a1. a0 -||-> a1 ==> P a0 a1 : thm
```

Again, we have to change the names that the inductive definitions package has chosen for our theorems:

```
- app (uncurry set_MLname) [                                        36
    ("-||->_rules", "predn_rules"), ("-||->_ind", "predn_ind"),
    ("-||->_cases", "predn_cases")
  ];
> val it = () : unit
```

6.5.2 Using RTC

Now we can define the reflexive and transitive closures of our two relations. We will use ASCII symbols for both that consist of the original symbol followed by an asterisk. Note also how, in defining the two relations, we have to use the `$` character to "escape" the symbols' usual fixities. This is exactly analogous to the way in which ML's `op` keyword is used. Finally, because we are defining a constant whose name is symbolic, we have to use `xDefine` rather than `Define`. This is because the latter function likes to try and guess an appropriate name for the definitions that it stores to disk. With symbolic names it doesn't know how to do this. The first parameter to `xDefine` is an alpha-numeric "stem" which provides the name to use.

```
- set_fixity "-->*" (Infix(NONASSOC, 510));                         37
> val it = () : unit

- val RTCredn_def = xDefine "RTCredn" `$-->* = RTC $-->`;
Definition has been stored under "RTCredn_def".
> val RTCredn_def = |- $-->* = RTC $--> : thm
```

We do exactly the same thing for the reflexive and transitive closure of our parallel reduction.

```
- set_fixity "-||->*" (Infix(NONASSOC, 510));                       38
> val it = () : unit

- val RTCpredn_def = xDefine "RTCpredn" `$-||->* = RTC $-||->`;
Definition has been stored under "RTCpredn_def".
> val RTCpredn_def = |- $-||->* = RTC $-||-> : thm
```

Finally, before doing some real proof, let's generate specialised versions of the RTC theorems for our new constants. This is a straightforward process; we just specialise the R in those theorems with `-->` and `-||->` and then rewrite with the two defining equations above in the RHS-LHS orientation. This will replace instances of RTC R with our new constants.

```
- val RTCredn_rules =                                                    39
    REWRITE_RULE [SYM RTCredn_def] (Q.ISPEC `$-->` RTC_rules)
  val RTCredn_ind =
    REWRITE_RULE [SYM RTCredn_def] (Q.ISPEC `$-->` RTC_ind)
  val RTCpredn_rules =
    REWRITE_RULE [SYM RTCpredn_def] (Q.ISPEC `$-||->` RTC_rules)
  val RTCpredn_ind =
    REWRITE_RULE [SYM RTCpredn_def] (Q.ISPEC `$-||->` RTC_ind);
> val RTCredn_rules =
    |- (!x. x -->* x) /\
       !x y z. x --> y /\ y -->* z ==> x -->* z : thm
  val RTCredn_ind =
    |- !RTC'.
         (!x. RTC' x x) /\
         (!x y z. x --> y /\ RTC' y z ==> RTC' x z) ==>
         !a0 a1. a0 -->* a1 ==> RTC' a0 a1 : thm
  val RTCpredn_rules =
    |- (!x. x -||->* x) /\
       !x y z. x -||-> y /\ y -||->* z ==> x -||->* z : thm
  val RTCpredn_ind =
    |- !RTC'.
         (!x. RTC' x x) /\
         (!x y z. x -||-> y /\ RTC' y z ==> RTC' x z) ==>
         !a0 a1. a0 -||->* a1 ==> RTC' a0 a1 : thm
```

Incidentally, in conjunction with PROVE we can now automatically demonstrate relatively long chains of reductions:

```
- PROVE [RTCredn_rules, redn_rules] ``S # K # K # x -->* x``;            40
Meson search level: ......
> val it = |- S # K # K # x -->* x : thm

- PROVE [RTCredn_rules, redn_rules]
    ``S # (S # (K # S) # K) # (S # K # K) # f # x -->*
      f # (f # x)``;
Meson search level: ..........................
> val it = |- S # (S # (K # S) # K) # (S # K # K) # f # x -->*
              f # (f # x) : thm
```

(The latter sequence is seven reductions long.)

6.5.3 Proving the RTCs are the same

We start with the easier direction, and show that everything in RTC $\rightarrow$ is in RTC $-|\!\!\rightarrow$. Because RTC is monotone (which fact is left to the reader to prove), we can reduce this to showing that $x \rightarrow y \supset x -|\!\!\rightarrow y$.

Our goal:

```
- g `!x y. x -->* y ==> x -||->* y`;                                    41
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !x y. x -->* y ==> x -||->* y
```

Now we rewrite with the definitions of our two symbols to expose the fact that they are reflexive, transitive closures:

```
- e (SIMP_TAC std_ss [RTCredn_def, RTCpredn_def]);                      42
OK..
1 subgoal:
> val it =
    !x y. RTC $--> x y ==> RTC $-||-> x y
```

We back-chain using our monotonicity result:

```
- e (HO_MATCH_MP_TAC RTC_monotone);                                     43
OK..
1 subgoal:
> val it =
    !x y. x --> y ==> x -||-> y
```

Now we can induct over the rules for $\rightarrow$:

```
- e (HO_MATCH_MP_TAC redn_ind);                                         44
OK..
1 subgoal:
> val it =
    (!x y f. x -||-> y ==> f # x -||-> f # y) /\
    (!f g x. f -||-> g ==> f # x -||-> g # x) /\
    (!x y. K # x # y -||-> x) /\
    !f g x. S # f # g # x -||-> f # x # (g # x)
```

We could split the 4-way conjunction apart into four goals, but there is no real need. It is quite clear that each follows immediately from the rules for parallel reduction.

```
- e (PROVE_TAC [predn_rules]);                                          45
OK..
Meson search level: ............

Goal proved. [...]
> val it =
    Initial goal proved.
    |- !x y. x -->* y ==> x -||->* y : goalstack
```

Packaged into a tidy little sub-goal-package-free parcel, our proof is

```
val RTCredn_RTCpredn = store_thm(
  "RTCredn_RTCpredn",
  ‘‘!x y. x -->* y   ==>   x -||->* y‘‘,
  SIMP_TAC std_ss [RTCredn_def, RTCpredn_def] THEN
  HO_MATCH_MP_TAC RTC_monotone THEN
  HO_MATCH_MP_TAC redn_ind THEN
  PROVE_TAC [predn_rules]);
```
46

...◇...

Our next proof is in the other direction. It should be clear that we will not just be able to appeal to the monotonicity of RTC this time; one step of the parallel reduction relation can not be mirrored with one step of the original reduction relation. It's clear that mirroring one step of the parallel reduction relation might take many steps of the original relation. Let's prove that then:

```
- g ‘!x y. x -||-> y   ==>   x -->* y‘;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !x y. x -||-> y ==> x -->* y
```
47

This time our induction will be over the rules defining the parallel reduction relation.

```
- e (HO_MATCH_MP_TAC predn_ind);
OK..
1 subgoal:
> val it =
    (!x. x -->* x) /\
    (!x y u v. x -->* y /\ u -->* v ==> x # u -->* y # v) /\
    (!x y. K # x # y -->* x) /\
    !f g x. S # f # g # x -->* f # x # (g # x)
```
48

There are four conjuncts here, and it should be clear that all but the second can be proved immediately by appeal to the rules for the transitive closure and for $\rightarrow$ itself. We could split apart the conjunctions and enter a THENL branch. However, we'd need to repeat the same tactic three times to quickly close three of the four branches. Instead, we use the TRY tactical to try applying the same tactic to all four branches. If our tactic fails on branch #2, as we expect, TRY will protect us against this failure and let us proceed.

```
e (REPEAT CONJ_TAC THEN                                                    49
   TRY (PROVE_TAC [RTCredn_rules, redn_rules]);
OK..
Meson search level: ....
Meson search level: ....
Meson search level: ............................
Meson search level: ..
1 subgoal:
> val it =
    !x y u v. x -->* y /\ u -->* v ==> x # u -->* y # v
```

Note that wrapping TRY around PROVE_TAC is not always wise. It can often take PROVE_TAC an extremely long time to exhaust its search space, and then give up with a failure. Here, "we got lucky".

Anyway, what of this latest sub-goal? If we look at it for long enough, we should see that it is another monotonicity fact. In this form, it's not quite right for easy proof. Let's go away and prove RTCredn_ap_monotonic separately. (Another exercise!) Our new theorem should state

```
val RTCredn_ap_monotonic = store_thm(                                      50
  "RTCredn_ap_monotonic",
  ``!x y. x -->* y ==> !z. x # z -->* y # z /\ z # x -->* z # y``,
  ...);
```

Now that we have this, our sub-goal is almost immediately provable. Using it, we know that

$$x\ u \rightarrow^* y\ u$$
$$y\ u \rightarrow^* y\ v$$

All we need to do is "stitch together" the two transitions above and go from $x\ u$ to $y\ v$. We can do this by appealing to our earlier RTC_RTC result and reminding PROVE_TAC that $\rightarrow^*$ is really just RTC $\rightarrow$.

```
e (PROVE_TAC [RTCredn_def, RTC_RTC, RTCredn_ap_monotonic]);                51
OK..
Meson search level: ............................

Goal proved. [...]
> val it =
    Initial goal proved.
    |- !x y. x -||-> y ==> x -->* y : goalstack
```

Odds are that you found that this last step took noticably longer than previous invocations of PROVE_TAC. This is because of the equality in the theorem RTCredn_def. (Equality reasoning always slows PROVE_TAC down.) Better performance is possible if you instead prove an appropriately specialised version of RTC_RTC and use this in place of both RTC_RTC and RTCredn_def. Let's go back and do this.

```
- b();                                                                    52
> val it =
    !x y u v. x -->* y /\ u -->* v ==> x # u -->* y # v
```

We need our specialised version of RTC_RTC.

```
- val RTCredn_RTCredn = save_thm(                                         53
    "RTCredn_RTCredn",
    SIMP_RULE std_ss [SYM RTCredn_def] (Q.ISPEC `$-->` RTC_RTC));
> val RTCredn_RTCredn =
    |- !x y z. x -->* y /\ y -->* z ==> x -->* z : thm
```

Now we can finish with:

```
- e (PROVE_TAC [RTCredn_RTCredn, RTCredn_ap_monotonic])                   54
OK..
Meson search level: .......

Goal proved.[...]
> val it =
    Initial goal proved.
    |- !x y. x -||-> y ==> x -->* y : goalstack
```

But given that we can finish off what we thought was an awkward branch with just another application of PROVE_TAC, we don't need to use our fancy TRY-footwork at the stage before. Instead, we can just merge the theorem lists passed to both invocations, dispense with the REPEAT CONJ_TAC and have a very short tactic proof indeed:

```
val predn_RTCredn = store_thm(                                            55
  "predn_RTCredn",
  ``!x y. x -||-> y  ==>  x -->* y``,
  HO_MATCH_MP_TAC predn_ind THEN
  PROVE_TAC [RTCredn_rules, redn_rules, RTCredn_RTCredn,
             RTCredn_ap_monotonic]);
```

...◇...

Now it's time to prove that if a number of parallel reduction steps are chained together, then we can mirror this with some number of steps using the original reduction relation. Our goal:

```
- g `!x y. x -||->* y  ==> x -->* y`;                                     56
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !x y. x -||->* y ==> x -->* y
```

We use the appropriate induction principle to get to:

```
- e (HO_MATCH_MP_TAC RTCpredn_ind);                                    57
OK..
1 subgoal:
> val it =
    (!x. x -->* x) /\ !x y z. x -||-> y /\ y -->* z ==> x -->* z
```

This we can finish off in one step. The first conjunct is obvious, and in the second the `x -||-> y` and our last result combine to tell us that `x -->* y`. Then this can be chained together with the other assumption in the second conjunct and we're done.

```
- e (PROVE_TAC [RTCredn_rules, predn_RTCredn,                          58
                RTCredn_RTCredn]);
OK..
Meson search level: .......

Goal proved.[...]
> val it =
    Initial goal proved.
    |- !x y. x -||->* y ==> x -->* y : goalstack
```

Packaged up, this proof is:

```
val RTCpredn_RTCredn = store_thm(                                      59
  "RTCpredn_RTCredn",
  ``!x y. x -||->* y   ==>  x -->* y``,
  HO_MATCH_MP_TAC RTCpredn_ind THEN
  PROVE_TAC [predn_RTCredn, RTCredn_RTCredn, RTCredn_rules]);
```

$\ldots\diamond\ldots$

Our final act is to use what we have so far to conclude that $\rightarrow^*$ and $\dashv\!\!\mid\!\!\mapsto^*$ are equal. We state our goal:

```
- g `$-||->* = $-->*`;                                                 60
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         $-||->* = $-->*
```

We want to now appeal to extensionality. This is best done with the conversion `FUN_EQ_CONV`, thus:

```
- e (CONV_TAC FUN_EQ_CONV);                                            61
OK..
1 subgoal:
> val it =
    !c. $-||->* c = $-->* c
```

This is progress but both "arrows" need another argument. We repeat ourselves (getting rid of extra universal quantifiers along the way):

```
- e (GEN_TAC THEN CONV_TAC FUN_EQ_CONV THEN GEN_TAC);                    62
OK..
1 subgoal:
> val it =
    c -||->* c' = c -->* c'
```

(You might be wondering why it is our variables are suddenly `c` and `c'`. This is because they are of type `:cl`, and the code that chooses the name thinks that it's reasonable to use variables named after the type.)

This goal is an easy consequence of our two earlier implications.

```
- e (PROVE_TAC [RTCpredn_RTCredn, RTCredn_RTCpredn]);                    63
OK..
Meson search level: ......

Goal proved. [...]
> val it =
    Initial goal proved.
    |- $-||->* = $-->* : goalstack
```

Packaged, the proof is:

```
val RTCpredn_EQ_RTCredn = store_thm(                                    64
  "RTCpredn_EQ_RTCredn",
  ``$-||->* = $-->*``,
  CONV_TAC FUN_EQ_CONV THEN GEN_TAC THEN
  CONV_TAC FUN_EQ_CONV THEN GEN_TAC THEN
  PROVE_TAC [RTCpredn_RTCredn, RTCredn_RTCpredn]);
```

6.5.4 Proving a diamond property for parallel reduction

Now we just have one substantial proof to go. Before we can even begin, there are a number of minor lemmas we will need to prove first. These are basically specialisations of the theorem `predn_cases`. We want exhaustive characterisations of the possibilities when the following terms undergo a parallel reduction: $x\ y$, K, S, K x, S x, K $x\ y$, S $x\ y$ and S $x\ y\ z$.

To do this, we will write a little function that derives characterisations automatically:

```
- fun characterise t = SIMP_RULE (srw_ss()) [] (SPEC t predn_cases);    65
> val characterise = fn : term -> thm
```

The `characterise` function specialises the theorem `predn_cases` with the input term, and then simplifies. The `srw_ss()` simpset includes information about the injectivity and disjointness of constructors and eliminates obvious impossibilities. For example,

```
- val K_predn = characterise ``K``;                                    66
<<HOL message: more than one resolution of overloading was possible>>
> val K_predn = |- !a1. K -||-> a1 = (a1 = K) : thm

- val S_predn = characterise ``S``;
<<HOL message: more than one resolution of overloading was possible>>
> val S_predn = |- !a1. S -||-> a1 = (a1 = S) : thm
```

Unfortunately, what we get back from other inputs is not so good:

```
- val Sx_predn0 = characterise ``S # x``;                              67
> val Sx_predn0 =
    |- !a1.
         S # x -||-> a1 =
         (a1 = S # x) \/
         ?y v. (a1 = y # v) /\ S -||-> y /\ x -||-> v : thm
```

That first disjunct is redundant, as the following demonstrates:

```
val Sx_predn = prove(                                                  68
  ``!x y. S # x -||-> y = ?z. (y = S # z) /\ (x -||-> z)``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [Sx_predn0, predn_rules, S_predn]);
```

Our `characterise` function will just have to help us in the proofs that follow.

```
val Kx_predn = prove(                                                  69
  ``!x y. K # x -||-> y = ?z. (y = K # z) /\ (x -||-> z)``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ``K # x``, predn_rules, K_predn]);
```

What of K x y? A little thought demonstrates that there really must be two cases this time.

```
val Kxy_predn = prove(                                                 70
  ``!x y z.
       K # x # y -||-> z =
       (?u v. (z = K # u # v) /\ (x -||-> u) /\ (y -||-> v)) \/
       (z = x)``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ``K # x # y``, predn_rules,
                 Kx_predn]);
```

By way of contrast, there is only one case for S x y because it is not yet a "redex" at the top-level.

```
val Sxy_predn = prove(                                                 71
  ``!x y z. S # x # y -||-> z =
            ?u v. (z = S # u # v) /\ (x -||-> u) /\ (y -||-> v)``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ``S # x # y``, predn_rules,
                 Sx_predn]);
```

Next, the characterisation for S $x\ y\ z$:

```
val Sxyz_predn = prove(                                                    72
  ``!w x y z. S # w # x # y -||-> z =
              (?p q r. (z = S # p # q # r) /\
                       w -||-> p /\ x -||-> q /\ y -||-> r) \/
              (z = (w # y) # (x # y))``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ``S # w # x # y``, predn_rules,
                 Sxy_predn]);
```

Last of all, we want a characterisation for $x\ y$. What `characterise` gives us this time can't be improved upon, for all that we might look upon the four disjunctions and despair.

```
- val x_ap_y_predn = characterise ``x # y``;                               73
> val x_ap_y_predn =
    |- !a1.
         x # y -||-> a1 =
         (a1 = x # y) \/
         (?y' v. (a1 = y' # v) /\ x -||-> y' /\ y -||-> v) \/
         (x = K # a1) \/
         ?f g. (x = S # f # g) /\ (a1 = f # y # (g # y)) : thm
```

Our last preliminary before we begin is to derive what is known as the *strong induction principle* for the inductive relation defining `-||->`. This gives us an induction principle where the application case changes from

```
!x y u v. P x y /\ P u v ==> P (x # u) (y # v)
```

where we can only assume `P x y` and `P u v` in trying to prove the application case, to the often more useful:

```
!x y u v.
     x -||-> y /\ P x y /\ u -||-> v /\ P u v ==>
     P (x # u) (y # v)
```

Deriving strong induction can be done automatically by the function `derive_strong_induction` found in the `IndDefRules` module. It takes a pair of a list of theorems and another theorem. The list of theorems consists of the rules of the relation split up into individual conjuncts, and the second argument is the normal induction principle.

Thus:

74
```
val predn_strong_ind =
  IndDefRules.derive_strong_induction (CONJUNCTS predn_rules, predn_ind);
> val predn_strong_ind =
    |- !P.
         (!x. P x x) /\
         (!x y u v.
            x -||-> y /\ P x y /\ u -||-> v /\ P u v ==>
            P (x # u) (y # v)) /\
         (!x y. P (K # x # y) x) /\
         (!f g x. P (S # f # g # x) (f # x # (g # x))) ==>
         !a0 a1. a0 -||-> a1 ==> P a0 a1 : thm
```

...◇...

Now we are ready to prove the final goal. It is

75
```
- g `!x y. x -||-> y ==>
           !z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         !x y. x -||-> y ==> !z. x -||-> z ==>
               ?u. y -||-> u /\ z -||-> u
```

We now apply the strong induction principle and split the goal into its individual conjuncts:

76
```
- e (HO_MATCH_MP_TAC predn_strong_ind THEN REPEAT CONJ_TAC);
OK..
4 subgoals:
> val it =
    !f g x z. S # f # g # x -||-> z ==>
              ?u. f # x # (g # x) -||-> u /\ z -||-> u


    !x y z. K # x # y -||-> z ==> ?u. x -||-> u /\ z -||-> u


    !x y u v.
      x -||-> y /\
      (!z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u) /\
      u -||-> v /\
      (!z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u) ==>
      !z. x # u -||-> z ==> ?u. y # v -||-> u /\ z -||-> u


    !x z. x -||-> z ==> ?u. x -||-> u /\ z -||-> u
```

The first goal is easily disposed of. The witness we would provide for this case is simply z, but PROVE_TAC will do the work for us:

```
- e (PROVE_TAC [predn_rules]);                                        77
OK..
Meson search level: ...

Goal proved. [...]
```

The next goal includes two instances of terms of the form `x # y -||-> z`. We can use our `x_ap_y_predn` theorem here. However, if we rewrite indiscriminately with it, we will really confuse the goal. We want to rewrite just the assumption, not the instance underneath the existential quantifier. Starting everything by repeatedly stripping can't lead us too far astray.

```
- e (REPEAT STRIP_TAC);                                               78
OK..
1 subgoal:
> val it =
    ?u. y # v -||-> u /\ z -||-> u
    ------------------------------------
      0.  x -||-> y
      1.  !z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u
      2.  u -||-> v
      3.  !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u
      4.  x # u -||-> z
```

We need to split up assumption 4. We can get it out of the assumption list using the `Q.PAT_ASSUM` theorem-tactical. We will write

```
Q.PAT_ASSUM `x # y -||-> z`
  (STRIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn])
```

The quotation specifies the pattern that we want to match. The second argument specifies how we are going to transform the theorem. Reading the compositions from right to left, first we will simplify with the `x_ap_y_predn` theorem and then we will assume the result back into the assumptions, stripping disjunctions and existentials as we go.[4]

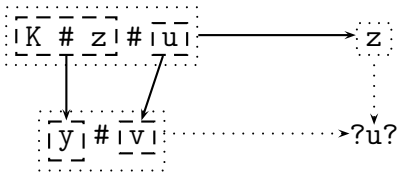
We already know that doing this is going to produce four new sub-goals (there were four disjuncts in the `x_ap_y_predn` theorem). At least one of these should be trivial because it will correspond to the case when the parallel reduction is just a "do nothing" step. Let's try eliminating the simple cases with a "speculative" call to `PROVE_TAC` wrapped inside a `TRY`. And before doing that, we should do some rewriting to make sure that equalities in the assumptions are eliminated.

So:

[4] An alternative to using `PAT_ASSUM` is to use `by` instead: you would have to state the four-way disjunction yourself, but the proof would be more "declarative" in style, and though wordier, might be more maintainable.

```
- e (Q.PAT_ASSUM `x # y -||-> z`                                        79
      (STRIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn]) THEN
     RW_TAC std_ss [] THEN
     TRY (PROVE_TAC [predn_rules]));
OK..
Meson search level: ............................
Meson search level: ............................
Meson search level: ..............
Meson search level: .....
2 subgoals:
> val it =
    ?u'. y # v -||-> u' /\ f # u # (g # u) -||-> u'
    ------------------------------------
      0.  S # f # g -||-> y
      1.  !z. S # f # g -||-> z ==> ?u. y -||-> u /\ z -||-> u
      2.  u -||-> v
      3.  !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u

    ?u. y # v -||-> u /\ z -||-> u
    ------------------------------------
      0.  K # z -||-> y
      1.  !z'. K # z -||-> z' ==> ?u. y -||-> u /\ z' -||-> u
      2.  u -||-> v
      3.  !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u
```

Brilliant! We've eliminated two of the four disjuncts already. Now our next goal features a term `K # z -||-> y` in the assumptions. We have a theorem that pertains to just this situation. But before applying it willy-nilly, let us try to figure out exactly what the situation is. A diagram of the current situation might look like



Our theorem tells us that `y` must actually be of the form `K # w` for some `w`, and that there must be an arrow between `z` and `w`. Thus:

```
- e (‘?w. (y = K # w) /\ (z -||-> w)‘ by PROVE_TAC [Kx_predn]);                80
OK..
Meson search level: ......
1 subgoal:
> val it =
    ?u. y # v -||-> u /\ z -||-> u
    ------------------------------------
      0.  K # z -||-> y
      1.  !z'. K # z -||-> z' ==> ?u. y -||-> u /\ z' -||-> u
      2.  u -||-> v
      3.  !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u
      4.  y = K # w
      5.  z -||-> w
```

On inspection, it becomes clear that the `u` must be `w`. The first conjunct requires `K # w # v -||-> w`, which we have because this is what Ks do, and the second conjunct is already in the assumption list. Rewriting (eliminating that equality in the assumption list first will make `PROVE_TAC`'s job that much easier), and then first order reasoning will solve this goal:

```
- e (RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]);                          81
OK..
Meson search level: ...

Goal proved. [...]
Remaining subgoals:
> val it =
    ?u'. y # v -||-> u' /\ f # u # (g # u) -||-> u'
    ------------------------------------
      0.  S # f # g -||-> y
      1.  !z. S # f # g -||-> z ==> ?u. y -||-> u /\ z -||-> u
      2.  u -||-> v
      3.  !z. u -||-> z ==> ?u. v -||-> u /\ z -||-> u
```

This case involving S is analogous. Here's the tactic to apply:

```
- e (‘?p q. (y = S # p # q) /\ (f -||-> p) /\ (g -||-> q)‘                    82
       by PROVE_TAC [Sxy_predn] THEN
     RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]);
OK..
Meson search level: ........
Meson search level: ...........

Goal proved.[...]
Remaining subgoals:
> val it =
    !f g x z. S # f # g # x -||-> z ==>
              ?u. f # x # (g # x) -||-> u /\ z -||-> u


    !x y z. K # x # y -||-> z ==> ?u. x -||-> u /\ z -||-> u
```

This next goal features a `K # x # y -||-> z` term that we have a theorem for already. And again, let's speculatively use a call to `PROVE_TAC` to eliminate the simple cases immediately (`Kxy_predn` is a disjunct so we'll get two sub-goals if we don't eliminate anything).

83

```
- e (RW_TAC std_ss [Kxy_predn] THEN
     TRY (PROVE_TAC [predn_rules]));
OK..
Meson search level: ..
Meson search level: ...

Goal proved. [...]
Remaining subgoals:
> val it =
    !f g x z. S # f # g # x -||-> z ==>
              ?u. f # x # (g # x) -||-> u /\ z -||-> u
```

Better yet! We got both cases immediately, and have moved onto the last case. We can try the same strategy.

84

```
- e (RW_TAC std_ss [Sxyz_predn] THEN PROVE_TAC [predn_rules]);
OK..
Meson search level: ..
Meson search level: ..........

Goal proved.[...]
> val it =
    Initial goal proved.
    |- !x y. x -||-> y ==> !z. x -||-> z ==>
             ?u. y -||-> u /\ z -||-> u : goalstack
```

The final goal proof can be packaged into:

```
val predn_diamond_lemma = prove(                                        85
  ‘‘!x y. x -||-> y ==>
          !z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u‘‘,
  HO_MATCH_MP_TAC predn_strong_ind THEN REPEAT CONJ_TAC THENL [
    PROVE_TAC [predn_rules],
    REPEAT STRIP_TAC THEN
    Q.PAT_ASSUM ‘x # y -||-> z‘
      (STRIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn]) THEN
    RW_TAC std_ss [] THEN
    TRY (PROVE_TAC [predn_rules]) THENL [
      ‘?w. (y = K # w) /\ (z -||-> w)‘ by PROVE_TAC [Kx_predn] THEN
      RW_TAC std_ss [] THEN PROVE_TAC [predn_rules],
      ‘?p q. (y = S # p # q) /\ (f -||-> p) /\ (g -||-> q)‘ by
         PROVE_TAC [Sxy_predn] THEN
      RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]
    ],
    RW_TAC std_ss [Kxy_predn] THEN PROVE_TAC [predn_rules],
    RW_TAC std_ss [Sxyz_predn] THEN PROVE_TAC [predn_rules]
  ]);
```

. . . ◇ . . .

We are on the home straight. The lemma can be turned into a statement involving the diamond constant directly:

```
val predn_diamond = store_thm(                                          86
  "predn_diamond",
  ‘‘diamond $-||->‘‘,
  PROVE_TAC [diamond_def, predn_diamond_lemma]);
```

And now we can prove that our original relation is confluent in similar fashion:

```
val confluent_redn = store_thm(                                         87
  "confluent_redn",
  ‘‘confluent $-->‘‘,
  PROVE_TAC [predn_diamond, RTCpredn_def,
             RTCredn_def, confluent_diamond_RTC,
             RTCpredn_EQ_RTCredn, diamond_RTC]);
```

6.6 Exercises

If necessary, answers to the first three exercises can be found by examining the source file in `examples/ind_def/clScript.sml`.

1. Prove that

$$\mathsf{RTC}\ R\ x\ y\ \wedge\ \mathsf{RTC}\ R\ y\ z\ \supset\ \mathsf{RTC}\ R\ x\ z$$

You will need to prove the goal by induction, and will probably need to massage it slightly first to get it to match the appropriate induction principle. Store the theorem under the name `RTC_RTC`.

2. Another induction. Show that

$$(\forall x\, y.\ R_1\ x\ y \supset R_2\ x\ y) \supset (\forall x\, y.\ \mathsf{RTC}\ R_1\ x\ y \supset \mathsf{RTC}\ R_2\ x\ y)$$

   Call the resulting theorem `RTC_monotone`.

3. Yet another RTC induction, but where R is no longer abstract, and is instead the original reduction relation. Prove

$$x \rightarrow^* y \quad \supset \quad \forall z.\ x\ z \rightarrow^* y\ z \wedge z\ x \rightarrow^* z\ y$$

   Call it `RTCredn_ap_monotonic`.

4. Come up with a counter-example for the following property:

$$\left(\begin{array}{l} \forall x\, y\, z.\quad R\ x\ y \ \wedge\ R\ x\ z \ \supset \\ \qquad\quad \exists u.\ \mathsf{RTC}\ R\ y\ u \ \wedge\ \mathsf{RTC}\ R\ z\ u \end{array}\right)$$
$$\supset$$
$$\mathsf{diamond}\ (\mathsf{RTC}\ R)$$

Chapter 7

Proof Tools: Propositional Logic

Users of HOL can create their own theorem proving tools by combining predefined rules and tactics. The ML type-discipline ensures that only logically sound methods can be used to create values of type `thm`. In this chapter, a real example is described.

Two implementations of the tool are given to illustrate various styles of proof programming. The first implementation is the obvious one, but is inefficient because of the 'brute force' method used. The second implementation attempts to be a great deal more intelligent. Extensions to the tools to allow more general applicability are also discussed.

The problem to be solved is that of deciding the truth of a closed formula of propositional logic. Such a formula has the general form

$$\begin{array}{rcl} \varphi & ::= & v \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \varphi = \varphi \\ \mathit{formula} & ::= & \forall \vec{v}.\ \varphi \end{array}$$

where the variables v are all of boolean type, and where the universal quantification at the outermost level captures all of the free variables.

7.1 Method 1: Truth Tables

The first method to be implemented is the brute force method of trying all possible boolean combinations. This approach's only real virtue is that it is exceptionally easy to implement. First we will prove the motivating theorem:

```
val FORALL_BOOL = prove(
  ``(!v. P v) = P T /\ P F``,
  SRW_TAC [][EQ_IMP_THM] THEN Cases_on `v` THEN SRW_TAC [][]);
```

The proof proceeds by splitting the goal into two halves, showing

$$(\forall v.\ P(v)) \Rightarrow P(\top) \wedge P(\bot)$$

(which goal is automatically shown by the simplifier), and

$$P(\top) \wedge P(\bot) \Rightarrow P(v)$$

for an arbitrary boolean variable v. After case-splitting on v, the assumptions are then enough to show the goal. (This theorem is actually already proved in the theory `bool`.)

The next, and final, step is to rewrite with this theorem:

```
val tautDP = SIMP_CONV bool_ss [FORALL_BOOL]
```

This enables the following

1

```
- tautDP ‘‘!p q. p /\ q /\ ~p‘‘;
> val it = |- (!p q. p /\ q /\ ~p) = F : thm

- tautDP ‘‘!p. p \/ ~p‘‘
> val it = |- (!p. p \/ ~p) = T : thm
```

and even the marginally more intimidating

2

```
- time tautDP
    ‘‘!p q c a. ~(((~a \/ p /\ ~q \/ ~p /\ q) /\
                  (~(p /\ ~q \/ ~p /\ q) \/ a)) /\
                 (~c \/ p /\ q) /\ (~(p /\ q) \/ c)) \/
                ~(p /\ q) \/ c /\ ~a‘‘;
runtime: 0.147s,    gctime: 0.012s,     systime: 0.000s.
> val it =
    |- (!p q c a.
          ~(((~a \/ p /\ ~q \/ ~p /\ q) /\ (~(p /\ ~q \/ ~p /\ q) \/ a)) /\
            (~c \/ p /\ q) /\ (~(p /\ q) \/ c)) \/ ~(p /\ q) \/ c /\ ~a) =
       T : thm
```

This is a dreadful algorithm for solving this problem. The system's built-in function, `tautLib.TAUT_CONV` solves the problem above ten times faster, even though it is also using a truth-table technique.[1] The only real merit in this solution is that it took one line to write. This is a general illustration of the truth that HOL's high-level tools, particularly the simplifier, can provide fast prototypes for a variety of proof tasks.

7.2 Method 2: the DPLL Algorithm

The Davis-Putnam-Loveland-Logemann method [4] for deciding the satisfiability of propositional formulas in CNF (Conjunctive Normal Form) is a powerful technique, still used in state-of-the-art solvers today. If we strip the universal quantifiers from our input formulas, our task can be seen as determining the validity of a propositional formula. Testing the negation of such a formula for satisfiability is a test for validity: if the formula's negation is satisfiable, then it is not valid (the satisfying assignment will make the original false); if the formula's negation is unsatisfiable, then the formula is valid (no assignment can make it false).

(The source code for this example is available in the file `examples/dpll.sml`.)

[1] The main difference is that `TAUT_CONV` simplifies the body of the universally quantified formula after each case-split. Our function does all of the case-splits and then simplifies. The simplifier, which works over a term top-down, can't implement `TAUT_CONV`'s algorithm directly.

Preliminaries

To begin, assume that we have code already to convert arbitrary formulas into CNF, and to then decide the satisfiability of these formulas. Assume further that if the input to the latter procedure is unsatisfiable, then it will return with a theorem of the form

$$\vdash \varphi = \mathtt{F}$$

or if it is satisfiable, then it will return a satisfying assignment, a map from variables to booleans. This map will be a function from HOL variables to one of the HOL terms `T` or `F`. Thus, we will assume

```
datatype result = Unsat of thm | Sat of term -> term
val toCNF : term -> thm
val DPLL : term -> result
```

(The theorem returned by `toCNF` will equate the input term to another in CNF.)

Before looking into implementing these functions, we will need to consider

- how to transform our inputs to suit the function; and
- how to use the outputs from the functions to produce our desired results

We are assuming our input is a universally quantified formula. Both the CNF and DPLL procedures expect formulas without quantifiers. We also want to pass these procedures the negation of the original formula. Both of the required term manipulations required can be done by functions found in the structure `boolSyntax`. (In general, important theories (such as `bool`) are accompanied by `Syntax` modules containing functions for manipulating the term-forms associated with that theory.)

In this case we need the functions

```
strip_forall : term -> term list * term
mk_neg       : term -> term
```

The function `strip_forall` strips a term of all its outermost universal quantifications, returning the list of variables stripped and the body of the quantification. The function `mk_neg` takes a term of type `bool` and returns the term corresponding to its negation.

Using these functions, it is easy to see how we will be able to take $\forall \vec{v}.\ \varphi$ as input, and pass the term $\neg\varphi$ to the function `toCNF`. A more significant question is how to use the results of these calls. The call to `toCNF` will return a theorem

$$\vdash \neg\varphi = \varphi'$$

The formula φ' is what will then be passed to `DPLL`. (We can extract it by using the `concl` and `rhs` functions.) If `DPLL` returns the theorem $\vdash \varphi' = \mathtt{F}$, an application of `TRANS` to this and the theorem displayed above will derive the formula $\vdash \neg\varphi = F$. In order to derive the final result, we will need to turn this into $\vdash \varphi$. This is best done by proving a bespoke theorem embodying the equality (there isn't one such already in the system):

```
val NEG_EQ_F = prove(``(~p = F) = p``, REWRITE_TAC []);
```

To turn $\vdash \varphi$ into $\vdash (\forall \vec{v}.\ \varphi) = \mathtt{T}$, we will perform the following proof:

$$\dfrac{\dfrac{\vdash \varphi}{\vdash \forall \vec{v}.\ \varphi}\ \mathtt{GENL}(\vec{v})}{\vdash (\forall \vec{v}.\ \varphi) = \mathtt{T}}\ \mathtt{EQT\_INTRO}$$

The other possibility is that DPLL will return a satisfying assignment demonstrating that φ' is satisfiable. If this is the case, we want to show that $\forall \vec{v}.\ \varphi$ is false. We can do this by assuming this formula, and then specialising the universally quantified variables in line with the provided map. In this way, it will be possible to produce the theorem

$$\forall \vec{v}.\ \varphi \vdash \varphi[\vec{v} := \textit{satisfying assignment}]$$

Because there are no free variables in $\forall \vec{v}.\ \varphi$, the substitution will produce a completely ground boolean formula. This will straightforwardly rewrite to F (if the assignment makes $\neg\varphi$ true, it must make φ false). Turning $\phi \vdash \mathtt{F}$ into $\vdash \phi = \mathtt{F}$ is a matter of calling DISCH and then rewriting with the built-in theorem IMP_F_EQ_F:

$$\vdash \forall t.\ t \Rightarrow \mathtt{F} = (t = \mathtt{F})$$

Putting all of the above together, we can write our wrapper function, which we will call DPLL_UNIV, with the UNIV suffix reminding us that the input must be universally quantified.

```
fun DPLL_UNIV t = let
  val (vs, phi) = strip_forall t
  val cnf_eqn = toCNF (mk_neg phi)
  val phi' = rhs (concl cnf_eqn)
in
  case DPLL phi' of
    Unsat phi'_eq_F => let
      val negphi_eq_F = TRANS cnf_eqn phi'_eq_F
      val phi_thm = CONV_RULE (REWR_CONV NEG_EQ_F) negphi_eq_F
    in
      EQT_INTRO (GENL vs phi_thm)
    end
  | Sat f => let
      val t_assumed = ASSUME t
      fun spec th =
          spec (SPEC (f (#1 (dest_forall (concl th)))) th)
          handle HOL_ERR _ => REWRITE_RULE [] th
    in
      CONV_RULE (REWR_CONV IMP_F_EQ_F) (DISCH t (spec t_assumed))
    end
end
```

The auxiliary function `spec` that is used in the second case relies on the fact that `dest_forall` will raise a `HOL_ERR` exception if the term it is applied to is not universally quantified. When `spec`'s argument is not universally quantified, this means that the recursion has bottomed out, and all of the original formula's universal variables have been specialised. Then the resulting formula can be rewritten to false (`REWRITE_RULE`'s built-in rewrites will handle all of the necessary cases).

The `DPLL_UNIV` function also uses `REWR_CONV` in two places. The `REWR_CONV` function applies a single (first-order) rewrite at the top of a term. These uses of `REWR_CONV` are done within calls to the `CONV_RULE` function. This lifts a conversion c (a function taking a term t and producing a theorem $\vdash t = t'$), so that `CONV_RULE` c takes the theorem $\vdash t$ to $\vdash t'$.

7.2.1 Conversion to Conjunctive Normal Form

A formula in Conjunctive Normal Form is a conjunction of disjunctions of literals (either variables, or negated variables). It is possible to convert formulas of the form we are expecting into CNF by simply rewriting with the following theorems

$$\begin{aligned}
\neg(\phi \wedge \psi) &= \neg\phi \vee \neg\psi \\
\neg(\phi \vee \psi) &= \neg\phi \wedge \neg\psi \\
\phi \vee (\psi \wedge \xi) &= (\phi \vee \psi) \wedge (\phi \vee \xi) \\
(\psi \wedge \xi) \vee \phi &= (\phi \vee \psi) \wedge (\phi \vee \xi) \\
\\
\phi \Rightarrow \psi &= \neg\phi \vee \psi \\
(\phi = \psi) &= (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)
\end{aligned}$$

Unfortunately, using these theorems as rewrites can result in an exponential increase in the size of a formula. (Consider using them to convert an input in Disjunctive Normal Form, a disjunction of conjunctions of literals, into CNF.)

A better approach is to convert to what is known as "definitional CNF". HOL includes functions to do this in the structure `defCNF`. Unfortunately, this approach adds extra, existential, quantifiers to the formula. For example

3

```
- defCNF.DEF_CNF_CONV ``p \/ (q /\ r)``;
> val it =
    |- p \/ q /\ r =
       ?x. (x \/ ~q \/ ~r) /\ (r \/ ~x) /\ (q \/ ~x) /\ (p \/ x) : thm
```

Under the existentially-bound `x`, the code has produced a formula in CNF. With an example this small, the formula is actually bigger than that produced by the naïve translation, but with more realistic examples, the difference quickly becomes significant. The last example used with `tautDP` is 20 times bigger when translated naïvely than when using `defCNF`, and the translation takes 150 times longer to perform.

But what of these extra existentially quantified variables? In fact, we can ignore the quantification when calling the core DPLL procedure. If we pass the unquantified body to DPLL, we will either get back an unsatisfiable verdict of the form $\vdash \varphi' = \mathtt{F}$, or a satisfying assignment for all of the free variables. If the latter occurs, the same satisfying assignment will also satisfy the original. If the former, we will perform the following proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\vdash \varphi' = \mathtt{F}}{\vdash \varphi' \Rightarrow \mathtt{F}}}{\vdash \forall \vec{x}.\ \varphi' \Rightarrow \mathtt{F}}}{\vdash (\exists \vec{x}.\ \varphi') \Rightarrow \mathtt{F}}}{\vdash (\exists \vec{x}.\ \varphi') = \mathtt{F}}$$

producing a theorem of the form expected by our `wrapper` function.

In fact, there is an alternative function in the `defCNF` API that we will use in preference to `DEF_CNF_CONV`. The problem with `DEF_CNF_CONV` is that it can produce a big quantification, involving lots of variables. We will rather use `DEF_CNF_VECTOR_CONV`. Instead of output of the form

$$\vdash \varphi = (\exists \vec{x}.\ \varphi')$$

this second function produces

$$\vdash \varphi = (\exists (v : \mathsf{num} \to \mathsf{bool}).\ \varphi')$$

where the individual variables x_i of the first formula are replaced by calls to the v function $v(i)$, and there is just one quantified variable, v. This variation will not affect the operation of the proof sketched above. And as long as we don't require literals to be variables or their negations, but also allow them to be terms of the form $v(i)$ and $\neg v(i)$ as well, then the action of the DPLL procedure on the formula φ' won't be affected either.

Unfortunately for uniformity, in simple cases, the definitional CNF conversion functions may not result in any existential quantifications at all. This makes our implementation of `DPLL` somewhat more complicated. We calculate a `body` variable that will be passed onto the `CoreDPLL` function, as well as a `transform` function that will transform an unsatisfiability result into something of the desired form. If the result of conversion to CNF produces an existential quantification, we use the proof sketched above. Otherwise, the transformation can be the identity function, `I`:

```
fun DPLL t = let
  val (transform, body) = let
    val (vector, body) = dest_exists t
    fun transform body_eq_F = let
      val body_imp_F = CONV_RULE (REWR_CONV (GSYM IMP_F_EQ_F)) body_eq_F
      val fa_body_imp_F = GEN vector body_imp_F
      val ex_body_imp_F = CONV_RULE FORALL_IMP_CONV fa_body_imp_F
    in
      CONV_RULE (REWR_CONV IMP_F_EQ_F) ex_body_imp_F
    end
  in
    (transform, body)
  end handle HOL_ERR _ => (I, t)
in
  case CoreDPLL body of
    Unsat body_eq_F => Unsat (transform body_eq_F)
  | x => x
end
```

where we have still to implement the core DPLL procedure (called `CoreDPLL` above). The above code uses `REWR_CONV` with the `IMP_F_EQ_F` theorem to affect two of the proof's transformations. The `GSYM` function is used to flip the orientation a theorem's top-level equalities. Finally, the `FORALL_IMP_CONV` conversion takes a term of the form

$$\forall x.\ P(x) \Rightarrow Q$$

and returns the theorem

$$\vdash (\forall x.\ P(x) \Rightarrow Q) = ((\exists x.\ P(x)) \Rightarrow Q)$$

7.2.2 The Core DPLL Procedure

The DPLL procedure can be seen as a slight variation on the basic "truth table" technique we have already seen. As with that procedure, the core operation is a case-split on a boolean variable. There are two significant differences though: DPLL can be seen as a search for a satisfying assignment, so that if picking a variable to have a particular value results in a satisfying assignment, we do not need to also check what happens if the same variable is given the opposite truth-value. Secondly, DPLL takes some care to pick good variables to split on. In particular, *unit propagation* is used to eliminate variables that will not cause branching in the search-space.

Our implementation of the core DPLL procedure is a function that takes a term and returns a value of type `result`: either a theorem equating the original term to false, or a satisfying assignment (in the form of a function from terms to terms). As the DPLL search for a satisfying assignment proceeds, an assignment is incrementally constructed. This suggests that the recursive core of our function will need to take a term (the current

formula) and a context (the current assignment) as parameters. The assignment can be naturally represented as a set of equations, where each equation is either $v = \mathtt{T}$ or $v = \mathtt{F}$.

This suggests that a natural representation for our program state is a theorem: the hypotheses will represent the assignment, and the conclusion can be the current formula. Of course, HOL theorems can't just be wished into existence. In this case, we can make everything sound by also assuming the initial formula. Thus, when we begin our initial state will be $\phi \vdash \phi$. After splitting on variable v, we will generate two new states $\phi, (v = \mathtt{T}) \vdash \phi_1$, and $\phi, (v = \mathtt{F}) \vdash \phi_2$, where the ϕ_i are the result of simplifying ϕ under the additional assumption constraining v.

The easiest way to add an assumption to a theorem is to use the derived rule `ADD_ASSUM`. But in this situation, we also want to simplify the conclusion of the theorem with the same assumption. This means that it will be enough to rewrite with the theorem $\psi \vdash \psi$, where ψ is the new assumption. The action of rewriting with such a theorem will cause the new assumption to appear among the assumptions of the result.

The `casesplit` function is thus:

```
fun casesplit v th = let
  val eqT = ASSUME (mk_eq(v, boolSyntax.T))
  val eqF = ASSUME (mk_eq(v, boolSyntax.F))
in
  (REWRITE_RULE [eqT] th, REWRITE_RULE [eqF] th)
end
```

A case-split can result in a formula that has been rewritten all the way to true or false. These are the recursion's base cases. If the formula has been rewritten to true, then we have found a satisfying assignment, one that is now stored for us in the hypotheses of the theorem itself. The following function, `mk_satmap`, extracts those hypotheses into a finite-map, and then returns the lookup function for that finite-map:

```
fun mk_satmap th = let
  val hyps = hypset th
  fun foldthis (t,acc) = let
    val (l,r) = dest_eq t
  in
    Binarymap.insert(acc,l,r)
  end handle HOL_ERR _ => acc
  val fmap = HOLset.foldl foldthis (Binarymap.mkDict Term.compare) hyps
in
  Sat (fn v => Binarymap.find(fmap,v)
               handle Binarymap.NotFound => boolSyntax.T)
end
```

The `foldthis` function above adds the equations that are stored as hypotheses into the finite-map. The exception handler in `foldthis` is necessary because one of the hypotheses will be the original formula. The exception handler in the function that looks

up variable bindings is necessary because a formula may be reduced to true without every variable being assigned a value at all. In this case, it is irrelevant what value we give to the variable, so we arbitrarily map such variables to `T`.

If the formula has been rewritten to false, then we can just return this theorem directly. Such a theorem is not quite in the right form for the external caller, which is expecting an equation, so if the final result is of the form $\phi \vdash \mathtt{F}$, we will have to transform this to $\vdash \phi = \mathtt{F}$.

The next question to address is what to do with the results of recursive calls. If a case-split returns a satisfying assignment this can be returned unchanged. But if a recursive call returns a theorem equating the input to false, more needs to be done. If this is the first call, then the other branch needs to be checked. If this also returns that the theorem is unsatisfiable, we will have two theorems:

$$\phi_0, \Delta, (v = \mathtt{T}) \vdash \mathtt{F} \qquad \phi_0, \Delta, (v = \mathtt{F}) \vdash \mathtt{F}$$

where ϕ_0 is the original formula, Δ is the rest of the current assignment, and v is the variable on which a split has just been performed. To turn these two theorems into the desired

$$\phi_0, \Delta \vdash \mathtt{F}$$

we will use the rule of inference `DISJ_CASES`:

$$\frac{\Gamma \vdash \psi \vee \xi \quad \Delta_1 \cup \{\psi\} \vdash \phi \quad \Delta_2 \cup \{\xi\} \vdash \phi}{\Gamma \cup \Delta_1 \cup \Delta_2 \vdash \phi}$$

and the theorem `BOOL_CASES_AX`:

$$\vdash \forall t.\ (t = \mathtt{T}) \vee (t = \mathtt{F})$$

We can put these fragments together and write the top-level `CoreDPLL` function, in Figure 7.1.

All that remains to be done is to figure out which variable to case-split on. The most important variables to split on are those that appear in what are called “unit clauses”, a clause containing just one literal. If there is a unit clause in a formula then it is of the form

$$\phi \wedge v \wedge \phi'$$

or

$$\phi \wedge \neg v \wedge \phi'$$

In either situation, splitting on v will always result in a branch that evaluates directly to false. We thus eliminate a variable without increasing the size of the problem. The process of eliminating unit clauses is usually called “unit propagation”. Unit propagation

```
fun CoreDPLL form = let
  val initial_th = ASSUME form
  fun recurse th = let
    val c = concl th
  in
    if c = boolSyntax.T then
      mk_satmap th
    else if c = boolSyntax.F then
      Unsat th
    else let
        val v = find_splitting_var c
        val (l,r) = casesplit v th
      in
        case recurse l of
          Unsat l_false => let
          in
            case recurse r of
              Unsat r_false =>
                Unsat (DISJ_CASES (SPEC v BOOL_CASES_AX) l_false r_false)
            | x => x
          end
        | x => x
      end
  end
in
  case (recurse initial_th) of
    Unsat th => Unsat (CONV_RULE (REWR_CONV IMP_F_EQ_F) (DISCH form th))
  | x => x
end
```

Figure 7.1: The core of the DPLL function

is not usually thought of as a case-splitting operation, but doing it this way makes our code simpler.

If a formula does not include a unit clause, then choice of the next variable to split on is much more of a black art. Here we will implement a very simple choice: to split on the variable that occurs most often. Our function `find_splitting_var` takes a formula and returns the variable to split on.

```
fun find_splitting_var phi = let
  fun recurse acc [] = getBiggest acc
    | recurse acc (c::cs) = let
        val ds = strip_disj c
      in
        case ds of
          [lit] => (dest_neg lit handle HOL_ERR _ => lit)
        | _ => recurse (count_vars ds acc) cs
      end
in
  recurse (Binarymap.mkDict Term.compare) (strip_conj phi)
end
```

This function works by handing a list of clauses to the inner `recurse` function. This strips each clause apart in turn. If a clause has only one disjunct it is a unit-clause and the variable can be returned directly. Otherwise, the variables in the clause are counted and added to the accumulating map by `count_vars`, and the recursion can continue.

The `count_vars` function has the following implementation:

```
fun count_vars ds acc =
  case ds of
    [] => acc
  | lit::lits => let
      val v = dest_neg lit handle HOL_ERR _ => lit
    in
      case Binarymap.peek (acc, v) of
        NONE => count_vars lits (Binarymap.insert(acc,v,1))
      | SOME n => count_vars lits (Binarymap.insert(acc,v,n + 1))
    end
```

The use of a binary tree to store variable data makes it efficient to update the data as it is being collected. Extracting the variable with the largest count is then a linear scan of the tree, which we can do with the `foldl` function:

```
fun getBiggest acc =
  #1 (Binarymap.foldl(fn (v,cnt,a as (bestv,bestcnt)) =>
                         if cnt > bestcnt then (v,cnt) else a)
                     (boolSyntax.T, 0)
                     acc
```

7.2.3 Performance

Once inputs get even a little beyond the clearly trivial, the function we have written (at the top-level, DPLL_UNIV) performs considerably better than the truth table implementation. For example, the generalisation of the following term, with 29 variables, takes wrapper two and a half minutes to demonstrate as a tautology:

```
(s0_0 = (x_0 = ~y_0)) /\ (c0_1 = x_0 /\ y_0) /\
(s0_1 = ((x_1 = ~y_1) = ~c0_1)) /\
(c0_2 = x_1 /\ y_1 \/ (x_1 \/ y_1) /\ c0_1) /\
(s0_2 = ((x_2 = ~y_2) = ~c0_2)) /\
(c0_3 = x_2 /\ y_2 \/ (x_2 \/ y_2) /\ c0_2) /\
(s1_0 = ~(x_0 = ~y_0)) /\ (c1_1 = x_0 /\ y_0 \/ x_0 \/ y_0) /\
(s1_1 = ((x_1 = ~y_1) = ~c1_1)) /\
(c1_2 = x_1 /\ y_1 \/ (x_1 \/ y_1) /\ c1_1) /\
(s1_2 = ((x_2 = ~y_2) = ~c1_2)) /\
(c1_3 = x_2 /\ y_2 \/ (x_2 \/ y_2) /\ c1_2) /\
(c_3 = ~c_0 /\ c0_3 \/ c_0 /\ c1_3) /\
(s_0 = ~c_0 /\ s0_0 \/ c_0 /\ s1_0) /\
(s_1 = ~c_0 /\ s0_1 \/ c_0 /\ s1_1) /\
(s_2 = ~c_0 /\ s0_2 \/ c_0 /\ s1_2) /\ ~c_0 /\
(s2_0 = (x_0 = ~y_0)) /\ (c2_1 = x_0 /\ y_0) /\
(s2_1 = ((x_1 = ~y_1) = ~c2_1)) /\
(c2_2 = x_1 /\ y_1 \/ (x_1 \/ y_1) /\ c2_1) /\
(s2_2 = ((x_2 = ~y_2) = ~c2_2)) /\
(c2_3 = x_2 /\ y_2 \/ (x_2 \/ y_2) /\ c2_2) ==>
(c_3 = c2_3) /\ (s_0 = s2_0) /\ (s_1 = s2_1) /\ (s_2 = s2_2)
```

The truth table implementation in tautLib takes over 100 times as long to prove the tautology. (But if you want real speed, the SAT_TAUT_PROVE function in the HolSatLib library does the above in less than a second, by using an external tool to generate the proof of unsatisfiability, and then translating that proof back into HOL.)

7.3 Extending our Procedure's Applicability

The function DPLL_UNIV requires its input to be universally quantified, with all free variables bound, and for each literal to be a variable or the negation of a variable. This makes DPLL_UNIV a little unfriendly when it comes to using it as part of the proof of a goal. In this section, we will write one further "wrapper" layer to wrap around DPLL_UNIV, producing a tool that can be applied to many more goals.

Relaxing the Quantification Requirement The first step is to allow formulas that are not closed. In order to hand on a formula that *is* closed to DPLL_UNIV, we can simply generalise over the formula's free variables. If DPLL_UNIV then says that the new, ground

formula is true, then so too will be the original. On the other hand, if DPLL_UNIV says that the ground formula is false, then we can't conclude anything further and will have to raise an exception.

Code implementing this is shown below:

```
fun nonuniv_wrap t = let
  val fvs = free_vars t
  val gen_t = list_mk_forall(fvs, t)
  val gen_t_eq = DPLL_UNIV gen_t
in
  if rhs (concl gen_t_eq) = boolSyntax.T then let
      val gen_th = EQT_ELIM gen_t_eq
    in
      EQT_INTRO (SPECL fvs gen_th)
    end
  else
    raise mk_HOL_ERR "dpll" "nonuniv_wrap" "No conclusion"
end
```

Allowing Non-Literal Leaves We can do better than nonuniv_wrap: rather than quantifying over just the free variables (which we have conveniently assumed will only be boolean), we can turn any leaf part of the term that is not a variable or a negated variable into a fresh variable. We first extract those boolean-valued leaves that are not the constants true or false.

```
fun var_leaves acc t = let
  val (l,r) = dest_conj t handle HOL_ERR _ =>
              dest_disj t handle HOL_ERR _ =>
              dest_imp t handle HOL_ERR _ =>
              dest_bool_eq t
in
  var_leaves (var_leaves acc l) r
end handle HOL_ERR _ =>
  if type_of t <> bool then
    raise mk_HOL_ERR "dpll" "var_leaves" "Term not boolean"
  else if t = boolSyntax.T then acc
  else if t = boolSyntax.F then acc
  else HOLset.add(acc, t)
```

Note that we haven't explicitly attempted to pull apart boolean negations (which one might do with dest_neg). This is because dest_imp also destructs terms ~p, returning p and F as the antecedent and conclusion. We have also used a function dest_bool_eq designed to pull apart only those equalities which are over boolean values. Its definition is

```
fun dest_bool_eq t = let
  val (l,r) = dest_eq t
  val _ = type_of l = bool orelse
          raise mk_HOL_ERR "dpll" "dest_bool_eq" "Eq not on bools"
in
  (l,r)
end
```

Now we can finally write our final DPLL_TAUT function:

```
fun DPLL_TAUT tm =
  let val (univs,tm') = strip_forall tm
      val insts = HOLset.listItems (var_leaves empty_tmset tm')
      val vars = map (fn t => genvar bool) insts
      val theta = map2 (curry (op |->)) insts vars
      val tm'' = list_mk_forall (vars,subst theta tm')
  in
      EQT_INTRO (GENL univs
                   (SPECL insts (EQT_ELIM (DPLL_UNIV tm''))))
  end
```

Note how this code first pulls off all external universal quantifications (with strip_forall), and then re-generalises (with list_mk_forall). The calls to GENL and SPECL undo these manipulations, but at the level of theorems. This produces a theorem equating the original input to true. (If the input term is not an instance of a valid propositional formula, the call to EQT_ELIM will raise an exception.)

Exercises

1. Extend the procedure so that it handles conditional expressions (both arms of the terms must be of boolean type).

Chapter 8

More Examples

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

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

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

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

`ind_def` This directory contains some examples of an inductive definition package in action. Featured are an operational semantics for a small imperative programming language, a small process algebra, and combinatory logic with its type system. The files were originally developed by Tom Melham and Juanito Camilleri and are extensively commented. The last is the basis for Chapter 6.

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

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

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

`parity` This sub-directory contains the files used in the parity example of Chapter 5.

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

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

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

References

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

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

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

[4] M. Davis, G. Logemann and D. Loveland, 'A machine program for theorem proving', *Communications of the ACM*, Vol. 5 (1962), pp. 394–397.

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

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

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

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

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

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

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

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

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