ProofPower

TUTORIAL

PPTex-2.9.1w2.rda.110727

Information on the current status of ProofPower is available on the World-Wide Web, at URL:

`http://www.lemma-one.demon.co.uk/ProofPower/index.html`

This document is published by:

Lemma 1 Ltd.
2nd Floor
31A Chain Street
Reading
Berkshire
UK
RG1 2HX
e-mail: `pp@lemma-one.com`

PPTex-2.9.1w2.rda.110727 13:54:16 27/07/2011

CONTENTS

ABOUT THIS PUBLICATION

0.1 Purpose

This document, one of several making up the user documentation for the ProofPower system, contains a tutorial introduction to the system.

0.2 Readership

This document is intended to be the first to be read by new users of ProofPower. It is a tutorial for learning the basic use of the system. The reader is assumed to be familiar with predicate logic.

0.3 Related Publications

A bibliography is given at the end of this document. Publications relating specifically to ProofPower are:

1. ProofPower HOL Tutorial Notes *[10]*, tutorial notes for the ProofPower-HOL course.
2. ProofPower Z Tutorial *[9]*, a tutorial covering the ProofPower Z support option.
3. ProofPower Description Manual *[7]*;
4. ProofPower Reference Manual *[12]*;
5. ProofPower Installation and Operation *[8]*;
6. ProofPower Document Preparation *[6]*.

0.4 Area Covered

This tutorial is an introductory ProofPower course which gives an idea of the way ProofPower is used, but which does not systematically explain the underlying principles. After working through this tutorial, the reader should be capable of using ProofPower for simple tasks, and should also be in a position to approach the ProofPower *Reference Manual* [12].

Once the ProofPower system is installed on the user's workstation, by following the procedure described in the Installation Guide, this tutorial should enable the potential ProofPower user to become familiar with the following subjects:

1. The metalanguage ProofPower-ML, and how to interact with the metalanguage compiler. The description of ProofPower-ML given here is very brief, only intended to be sufficient to support

the exposition of ProofPower. ProofPower-ML is an extension of the programming language Standard ML. For a more complete introduction to Standard ML the reader is referred to [3], [5], or [11].

2. The formal logic supported by the ProofPower system (higher order logic) and its manipulation via the metalanguage.
3. Forward proof and derived rules of inference.
4. Goal directed proof, and tactics and tacticals.

The sections that follow cover these topics in the sequence shown above.

0.5 Assumptions

Though this tutorial can be read independently, it is most beneficially read while running ProofPower so that the features described can be observed at first hand. The instructions for running ProofPower assume that the reader has available an installed ProofPower system, and that the reader is following the tutorial at a workstation and trying out the examples interactively. Basic familiarity with using the X Windows System is assumed.

Other tutorial manuals are available with ProofPower, which are best attempted after reading this tutorial, which is the tutorial most suitable for absolute beginners. After reading this tutorial, a more thorough knowledge of ProofPower may be obtained by working through ProofPower *HOL Tutorial Notes* [10], which contains exercises and solutions, and covers a wider range of facilities than those described here. Those interested in the Z support facilities of ProofPower may then work through ProofPower *Z Tutorial* [9] which describes how to use ProofPower for specification and proof in ProofPower-Z.

0.6 Acknowledgements

ICL gratefully acknowledges its debt to the many researchers (both academic and industrial) who have provided the intellectual capital on which ICL has drawn in the development of ProofPower.

We are particularly indebted to Mike Gordon of Cambridge, both for his leading role in the research on which ProofPower is based, and for the text, [13], which formed the starting point for the development of this tutorial.

The ProofPower system is a proof tool for Higher Order Logic which builds upon ideas arising from research carried out at the Universities of Cambridge and Edinburgh, and elsewhere.

In particular the logic supported by the system is (at an abstract level) identical to that implemented in the Cambridge HOL system [1], and the paradigm adopted for implementation of proof support for the language follows that adopted by Cambridge HOL, originating with the LCF system developed at Edinburgh [2]. The functional language "standard ML" used both for the implementation and as a interactive metalanguage for proof development, originates in work at Edinburgh, and has been developed to its present state by an international group of academics and industrial researchers. The implementation of Standard ML on which ProofPower is based was itself implemented by David Matthews at the University of Cambridge, and is now commercially marketed by Abstract Hardware Limited.

GETTING STARTED

This manual is intended to serve as a tutorial introduction to the use of ProofPower for simple specification and proof using the ProofPower-HOL language. It contains numerous examples of interaction with ProofPower. Many readers will wish to try out the examples interactively as they read. The remainder of this chapter explains how to do this.

The instructions which follow assume that you are working in a user name which has been set up to run ProofPower as described in the installation instructions supplied with the software. If you have problems with the instructions, this may well be because your user name has not been set up to run ProofPower or because there has been some problem with the installation of ProofPower. If you have difficulty, you are advised first to consult the installation instructions or the person responsible for installing the ProofPower software on your system.

1.1 Interaction with ProofPower

The most convenient way to use ProofPower for developing both specifications and proofs involves two parallel activities:

- Using an editor to develop a LaTeX source document called a 'literate script' (see ProofPower *Document Preparation* [6]) in which ProofPower commands are recorded.
- Executing ProofPower-ML commands, typically extracted from the script. The various object languages, ProofPower-HOL, ProofPower-Z, etc., supported by the tool are all embedded in ProofPower-ML, and execution of ProofPower-ML commands is how fragments of specification are checked and how proof steps are conducted.

Under the X Windows System, the recommended way of carrying out these two activities is to use the program `xpp` which integrates a general purpose editor with the ProofPower-ML compiler and gives easy ways of carrying out many of the common tasks (e.g., entering mathematical symbols). Sections 1.2 to 1.7 below describe how to use `xpp` to work through the examples in this tutorial. The instructions in these sections assume that you have started up X in the way appropriate for your system and that you have an 'xterm', 'command tool' or other UNIX terminal emulator on the work station screen.

An advantage of `xpp` is that it supports the use of mathematical symbols on the screen. It is possible to run ProofPower without using `xpp`, but this advantage is then lost: mathematical symbols have to be handled as ASCII keywords. Users who are obliged to work the tutorial examples in ASCII are referred to section 1.9 below.

1.2 Setting Up

The results of work with the ProofPower system are stored in what is called a *database*. A database stores objects of different kinds: definitions, axioms, theorems and theories. Work with ProofPower will commonly result in updating a database.

The ProofPower system is issued with a relatively large database of predefined objects, and it is desirable both to avoid casual modification to this issued database and to avoid making unnecessary copies of it. Consequently the user is recommended to create a new empty database, having the issued database as its 'parent', and to use this 'child' database thereafter while working through the examples. The objects stored in the parent are available through use of the child.

The file-name for a database is derived from the name you use to name the database on the command line. The conventions for the file-names depend on the operating system and hardware you are using and on the Standard ML compiler used to build ProofPower. For example, with the Standard ML of New Jersey compiler running with the Linux operating system on Intel x86 hardware, a database named 'demo' would be held in a file named 'demo-x86-linux'.

The following UNIX commands create a new database named 'demo' (which is a child of the issued database and is held in a file in the current directory):

```
pp_make_database -p hol demo
```

The instructions in the rest of this chapter assume you have changed to a directory in which you wish to store your work on this tutorial and that you have successfully executed the above command.

1.3 Entering ProofPower

The ProofPower system is entered by a command at the UNIX prompt, invoking the `xpp` program and giving it arguments identifying, amongst other things, the name of a file you wish to edit and the name of a *database*.

Following the instructions in the previous section, you will have a new database called 'demo'. The source text of the tutorial is in the file '$PPHOME/doc/usr004.doc'. The following UNIX command will start up `xpp` to edit that file and to run ProofPower on the new database.

```
xpp -f $PPHOME/doc/usr004.doc -d demo
```

See the entry on `xpp` in ProofPower *Reference Manual* [12] for more information.)

When `xpp` starts up, you will see that the its display has four main features as follows, (working from top to bottom).

Menu Bar This contains menus which you use to select the main functions of `xpp`. The menu at the right-hand end is used to give on-line help with `xpp`.

File Name Bar This contains the name of the file you are editing.

Script Window This is the text area in which you carry out your editing work. It will come up containing the tutorial script.

Journal Window This displays a journal of your transactions with ProofPower. It will come up displaying start-up messages from the ProofPower-ML compiler followed by a prompt: ':)'.

1.4 Using the Editor

The Script Window in xpp together with the Menu Bar comprise a general purpose text editor providing several features which are convenient for editing ProofPower literate scripts; The File and Edit Menu provide standard editing features, and the Tools Menu is used to pop up tools to perform various functions, e.g., the Palette Tool which gives an easy way to enter mathematical symbols (both into the Script Window and into other parts of xpp — see the Help Menu entry about the Tools Menu for more information).

1.5 Executing ProofPower Commands

The basic way of executing ProofPower commands using xpp is with the 'Execute Selection' item in the Command Menu in response to the ProofPower-ML prompt: ':)'.

In this manual, lines of input for ProofPower will be shown in the following style:

```
SML
|"This is a line of Standard ML Input";
```

To execute a line of ProofPower-ML, select it in the xpp Script Window (e.g., by double clicking with the left mouse button) and then chose 'Execute Selection' from the Command Menu. The line will be copied to the Journal Window for processing by the ProofPower-ML compiler.

The output which subsequently appears in the Journal Window is shown in this manual thus:

```
|val it = "This is a line of Standard ML Input" : string
```

There are several shortcuts to make interaction easier; for example, you can type 'Control-X' instead of selecting 'Execute Selection'. A more powerful shortcut is the Command Line Tool, which you can invoke from the Tools menu; this helps you type in and execute commands which you do not want to keep in the script, and can be used to remember and recall commands which you use frequently. Consult the Help Menu for further information.

1.6 ProofPower-ML Prompts and Interrupts

The ProofPower-ML compiler uses the prompt ':)' to invite you to input a command. If you have typed in a syntactically incomplete line of input, the compiler will expect you to complete the command on subsequent 'continuation lines'. For example, the following command is spread over three input lines.

```
SML
|1
|+
|2;
```

If you suspect that the compiler is waiting for you to complete a command, but you wish to abort the command instead, you can use the 'Interrupt' item in the Command Menu to do so. The 'Abandon' item that you will also see in the Command Menu is for backwards compatibility with earlier versions of ProofPower only.

1.7 Ending The Session

To leave xpp, use the 'Quit' item in its File Menu. You will be prompted if you have not saved all your changes to the file you are editing or if the ProofPower-ML compiler is still running.

It is recommended that you always quit the ProofPower-ML compiler before quitting xpp. (Quitting xpp without quitting the ProofPower-ML compiler generally works, but may bypass some tidying up such as removal of temporary files.)

If you wish the work you have carried out to be saved in the database before your quit, you should execute the following ProofPower-ML command before quitting xpp:

SML
```
save_and_quit();
```

This command will cause the database to be updated by storing in it the results of the work done during the session.

To quit from the ProofPower system without updating the database, execute the ProofPower-ML command before quitting xpp:

SML
```
quit();
```

This will ask you for confirmation (which you can conveniently send using the Command Line Tool).

1.8 Input from a File

Within a ProofPower session, the ProofPower system may be directed to take input from a file, rather than interactively, by executing, for example, the command

SML
```
use_file "myfile";
```

After reading and executing the last line in the file, the ProofPower system returns to taking interactive input.

There is an option on the UNIX command-line to read and execute such a 'script' file immediately on entry to the ProofPower system:

```
xpp -command pp -d demo -i myfile
```

1.9 Working In ASCII

To work through this tutorial in ASCII, first copy the source document into a local directory and then convert it to ASCII as follows:

```
cp installdir/docs/usr004.doc .
conv_ascii usr004
textedit usr004.doc
```

Create a database as described in section 1.2 and then begin an interactive ProofPower session directly rather than via xpp by executing the UNIX command:

```
pp -d demo
```

ProofPower will then come up and prompt for input from your terminal. You should immediately set ProofPower into ASCII mode by entering the following command:

SML

```
set_flag ("use_extended_chars", false);
```

When a flag is set the previous value of the flag is returned, so the above command will respond:

```
val it = true : bool
```

Now type directly in response to the ProofPower prompt, or use cut-and-paste from the text editor of your choice.

The ASCII keywords used to represent mathematical symbols in ASCII mode are documented in ProofPower *Document Preparation* [6].

CONVENTIONS

This chapter describes some conventions followed in this manual.

2.1 Sessions

Throughout this tutorial, the sequences of user's interactions with the system and the system's responses are called 'sessions'. All sessions in this documentation are displayed in numbered boxes. 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. In particular, variable bindings made in earlier boxes are assumed to persist to later ones in the same sequence.

2.2 Input and Output

As already mentioned, input to ProofPower-ML will be marked by a vertical line on the left, with 'SML' in small letters, thus:

```
SML
│"This is a line of ProofPower Input";
```

This is, in fact, the usual appearance of ProofPower commands in a printed literate script.

The output resulting from the above input is shown in this manual marked by a vertical line alone, thus:

```
│val it = "This is a line of Standard ML Input" : string
```

A FIRST EXAMPLE

In this section a very small example is presented and briefly explained in order to give the reader some idea of what to expect in the following sections. The purpose of the ProofPower system is to support proof. A style of proof which is favoured by ProofPower (but not the only one possible) is called **goal-oriented proof**. In this style of proof:

- firstly, a conjecture is stated. The conjecture is called the **goal** of the proof process.
- then a proof is conducted in one or more steps, each step being specified by the user. The steps are progressive transformations of the goal, aimed at transforming the goal to the logical value 'true'. When this has been achieved the conjecture is proved, yielding a **theorem**.

To illustrate the actual mechanics of the process, here is an example which shows three lines of user input to state a conjecture $p \lor \neg\ p$, perform a one-step proof, and then record the proved conjecture as a theorem. Each line of user input is followed by a system response, (which is not reproduced in full here). Inputs and responses are annotated with comments between the symbols (* and *).

```
SML                                                                    1
│ set_goal([ ], ⌜p ∨ ¬p⌝ );             (* 1: state the conjecture *)

│ ... ⌜p ∨ ¬ p⌝  ...                    (* response echoes goal *)
│

SML
│ apply_tactic (REPEAT strip_tac);      (* 2: perform one−step proof *)

│ ... goal achieved ...                 (* response  *)

SML
│ val example_theorem = top_thm();      (* 3:  save resulting theorem *)

│ val example_theorem = ⊢ p ∨ ¬ p : THM (* response *)
```

The following points may be noted:

- Each line of user input is in the **metalanguage**, which is called ProofPower-ML. The conjecture $p \lor \neg\ p$ is a term in the **object language**, which is called ProofPower-HOL. The object language term occurs, surrounded by the special quotation symbols ⌜ and ⌝, embedded in the metalanguage command.

- Stating the conjecture is accomplished by initialising a **stack** of goals. The proved theorem is extracted from the top of the stack at the end of the process.
- A proof step is accomplished by applying what is called a **tactic** to the goal at the top of the stack. At each step the user must **choose** an appropriate tactic.
- A tactic is a procedure which attempts to find a sequence of inferences in the ProofPower-HOL logic such that the goal can be inferred to be true. A tactic may be
  - wholly successful, as in this example, or
  - partly successful, in which case the goal is reduced to a simpler goal, so that a further tactic must be chosen and applied, or
  - wholly unsuccesful, leaving the goal unchanged, in which case a different tactic must be chosen.
- The proof system makes available a set of predefined tactics. Different tactics are available to exploit different features of the goal. Users can construct new tactics as programmed application of existing tactics.
- In this example the predefined tactic denoted by *REPEAT strip_tac* was chosen, on the ground that this tactic is a standard opening gambit, capable of achieving many useful simplifications of the goal, and indeed achieving simple goals by itself.
- The system, **not the user**, is responsible for the soundness of the process of logical inference performed by any tactic, whether the tactic is predefined or user-defined. Thus to choose an inappropriate tactic at any step does not risk an unsound inference, but merely failure to make progress.
- Finally, note that what achieves the proof of the conjecture $\ulcorner p \lor \neg\ p \urcorner$ is a metalanguage expression *apply_tactic* (*REPEAT strip_tac*), and the latter does not have the conventional appearance of a proof. It should be regarded, not as a proof itself, but rather as a program which, when executed, will perform a formal proof of $\ulcorner p \lor \neg\ p \urcorner$ and many other such propositions.

INTRODUCTION TO THE METALANGUAGE

This chapter is a brief introduction to the metalanguage ProofPower-ML. ProofPower-ML is an extension of the programming language Standard ML. The extensions are:

- An extended character set to include symbols of logic and mathematics.
- An additional form of quotation, analogous to the quotation of ASCII strings, for the quotation of object language expressions.
- A collection of predefined functions.

The aim of this chapter is to explain only enough of ProofPower-ML to make the following chapters comprehensible. The rest of this chapter applies equally to Standard ML and to ProofPower-ML. For a more complete introduction to Standard ML the reader is referred to [3], [5], or [11].

Throughout the rest of this document, ProofPower-ML will be referred to simply as ML. ML is an interactive programming language. When interacting directly with the system, (which is called 'at the top level') one can evaluate expressions and perform declarations.

4.1 Expressions

```
SML                                                        1
│   1+1;

│   val it = 2 : int
```

This box shows an example of entering an ML expression through the keyboard (that is, 'at the top level'), which is then evaluated and the result displayed. The semicolon ';' is used to terminate a top-level phrase. The display of the result can be seen to consist of:

- The letters *val*, indicating that a value is to follow.
- A name for the value. In this case the user has not supplied any name, having merely typed in the anonymous expression *1+1*, and so the system supplies the name *it*. The value of the most-recently-entered anonymous expression at the top level can always be referred to as *it*.
- The symbol =.
- The value, in this case *2*.
- A colon followed by an indication of the type of the value. In this case, the value 2 is of type integer, abbreviated to *int*. The ML type checker infers the type of expressions using methods invented by Robin Milner, [4].

4.2 Lists and Strings

. The ML expression [*2*,*3*,*4*,*5*] is a list of four integers.

```
SML                                                          2
│   [2,3,4,5];

│   val it = [2, 3, 4, 5] : int list
```

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 ProofPower logic which is explained in Chapter 5.

Expressions such as "*a*", "*b*", "*foo*" are *strings* and have type *string*. Any sequence of ASCII characters can be written between the quotes. The infix function ^ concatenates two strings to form a single string.

```
SML                                                          3
│   "tog" ^ "ether";

│   val it = "together" : string
```

4.3 Declarations

A declaration may have the form *val n* = *e*, which results in the value of the expression *e* being bound to the name *n*.

```
SML                                                          4
│     val x = 42;

│     val x = 42 : int
│

SML
│     x + 1;

│     val it = 43 : int
```

4.4 Function Applications

The application of a function f to an argument x can be written as $f\ x$. The more conventional f (x) is also allowed. The expression $f\ x_1\ x_2\ \cdots\ x_n$ abbreviates the less intelligible expression $(\cdots((f\ x_1)x_2)\cdots)x_n$. That is, function application is left associative.

Functions may be infix, as in the case of +. Another infix function is :: which constructs a list which is the left argument followed by the right argument. Other list processing functions include *hd*, which yields the head – the first element – of a list), *tl*, which yields the tail – all but the first element – of a list) and *null*, which tests for an empty list.

```
SML                                                           5
│ val L = 1:: [2, 3];

│ val L = [1, 2, 3] : int list

SML
│ hd L;

│ val it = 1 : int

SML
│ tl L;

│ val it = [2, 3] : int list

SML
│ tl (tl it);

│ val it = [ ] : int list
```

4.5 Pairs and Tuples

An expression of the form (e_1, e_2) evaluates to a pair, with first component and second component having respectively 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 $\sigma_1 * \sigma_2$. A tuple $(e_1, ..., e_n)$ is NOT equivalent to $(e_1, (e_2, ..., e_n))$, unless n = 2. The first and second components of a pair (but not a tuple of length greater than two) can be extracted with the ML functions *fst* and *snd* respectively. The i-th component of a tuple can be extracted with the function $\#i$.

```
SML                                                              6
│  (1, 2, (true, "abc"));

│  val it = (1, 2, (true, "abc")) : int * int * (bool * string)

SML
│  #3 it;

│  val it = (true, "abc") : bool * string

SML
│  snd it;

│  val it = "abc" : string
```

The ML expressions *true* and *false* denote the two truth values, being of type *bool*.

4.6 Polymorphic Types

ML types can contain the *type variables 'a*, *'b*, 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. An example of a function with polymorphic type is *hd* (head of a list), which is applicable to lists of any type:

```
SML                                                              7
│   hd;

│   val it = fn : 'a list -> 'a
```

This example also shows that, in the system's response, the value of a function is not displayed in full, but only symbolized by the letters *fn*. This is true of all function-values.

4.7 Declarations of Functions

The function which, for example, computes $x + 1$ from x can be defined and given a name, say, *step*, as follows:

8
```
SML
│ fun step x = x + 1;

│ val step = fn : int -> int

SML
│ step 6;

│ val it = 7 : int
```

The declaration *fun step x = x + 1* is a convenient abbreviation for *val step = fn x => x + 1*.

9
```
SML
│ val step = fn x => x + 1;

│ val step = fn : int -> int

SML
│ step 6;

│ val it = 7 : int
```

Here *fn x => x + 1* is an expression the value of which is a function. In what follows, it will be common for the arguments or results of functions themselves to be functions. In the following example *twice* is a function which takes a function as argument and returns another as a result, such that applying the result-function is equivalent to applying the argument-function twice.

```
SML                                                        10
│     fun twice f = fn x => f (f x);

│     val twice = fn : ('a -> 'a) -> 'a -> 'a

SML
│     val hop = twice step;

│     val hop = fn : int -> int

SML
│     hop 6;

│     val it = 8 : int

SML
│     (twice tl) [1, 3, 5, 7];

│     val it = [5, 7] : int list
```

Again the syntactic abbreviation may be employed to give a neater definition of twice:

```
SML                                                        11
│     fun twice f x = f (f x);

│     val twice = fn : ('a -> 'a) -> 'a -> 'a

SML
│     twice step 6;

│     val it = 8 : int
```

Note particularly that the expression *twice step 6* is equivalent to (*twice step*) *6*. The declaration above, *fun twice f x = f (f x)*, is an example of a more general form of declaration of a function, *fun f* v_1 ... v_n = *e* where each v_i is an argument and *e* is an expression.

As a final example, a useful built-in function is *map* which applies its function-argument to each member of a list to produce a list:

SML 12

```
map step [1, 3, 5];


val it = [2, 4, 6] : int list
```

The sessions above are enough to give an idea of ML. In the next sections, the logic supported by the ProofPower system (higher order logic) will be introduced, together with the tools in ML for manipulating it.

INTRODUCTION TO THE ProofPower LOGIC

The ProofPower 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*. Instead, there are terms of a boolean type.

5.1 Overview of higher order logic

It is assumed that the reader is familiar with predicate logic. The table below summarizes the notation used. In what follows the logic supported by ProofPower will be called the HOL logic, or simply HOL.

Terms of the HOL Logic

Kind of term	*HOL notation*	*Description*
Truth	T	*true*
Falsity	F	*false*
Negation	$\neg t$	*not t*
Disjunction	$t_1 \vee t_2$	t_1 *or* t_2
Conjunction	$t_1 \wedge t_2$	t_1 *and* t_2
Implication	$t_1 \Rightarrow t_2$	t_1 *implies* t_2
Equality	$t_1 = t_2$	t_1 *equals* t_2
$\forall$-quantification	$\forall x \bullet t$	*for all* x, t
$\exists$-quantification	$\exists x \bullet t$	*for some* x, t
ε-term	$\varepsilon\ x \bullet t$	*an* x *such that* t
Conditional	*if t then* t_1 *else* t_2	*if t then* t_1 *else* t_2

Terms of the HOL logic are represented in ML by an *abstract type*[1] called *TERM*. They are represented as character strings which are input, not between the usual quotation symbols but rather between the symbols ⌜ and ⌝. For example, the expression ⌜ $x \wedge y \Rightarrow z$⌝ evaluates in ML to a term representing $x \wedge y \Rightarrow z$. Terms can be manipulated by various built-in ML functions. For example, the ML function *dest_*$\Rightarrow$ 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_*$\wedge$ of type *TERM −> TERM ∗ TERM* splits a conjunction into its two conjuncts.

[1] Abstract types appear to the user as primitive types with a collection of operations

```
SML                                                          1
│    ⌜x∧y⇒z⌝;

│    val it = ⌜x ∧ y ⇒ z⌝ : TERM

SML
│    dest_⇒ it;

│    val it = (⌜x ∧ y⌝, ⌜z⌝) : TERM * TERM

SML
│    dest_∧ (fst it);

│    val it = (⌜x⌝, ⌜y⌝) : TERM * TERM
```

Terms of the HOL logic are quite similar in appearance to ML expressions, but the distinction must be carefully observed. Indeed, terms of the logic have types in a way which is similar to the way in which ML expressions have types. For example, ⌜*1*⌝ is an ML expression with ML type *TERM*. The HOL type of this term is :ℕ, the type of the natural numbers.

The types of HOL terms form an ML type called *TYPE*. Expressions having the form ⌜: ....⌝ evaluate to logical (that is, HOL) types. The built-in function *type_of* has ML type *TERM −> TYPE* and returns the logical type of a term.

```
SML                                                          2
│    ⌜(1,2)⌝;

│    val it = ⌜(1, 2)⌝ : TERM

SML
│    type_of it;

│    val it = ⌜:ℕ × ℕ⌝ : TYPE

SML
│    (⌜1⌝, ⌜2⌝);

│    val it = (⌜1⌝, ⌜2⌝) : TERM * TERM

SML
│    type_of (fst it);

│    val it = ⌜:ℕ⌝ : TYPE
```

To emphasise the distinction between between the ML types of ML expressions and the logical types of HOL terms , the former will be referred to as *metalanguage types* and the latter as *object language types*.

HOL terms can be input using explicit *quotation*, as above, using ⌜ and ⌝ for quotation marks, or they can be constructed using ML constructor functions. The function *mk_var* constructs a variable from a string and a type. In the example below, three terms are constructed, each representing a single object-language variable of type *BOOL*, and metalanguage names are chosen for the terms to coincide with the names of the object-language variables. These are used later.

```
SML                                                   3
│    val x = mk_var("x",ⓣ:BOOL⌝);
│    val y = mk_var("y",ⓣ:BOOL⌝);
│    val z = mk_var("z",ⓣ:BOOL⌝);

│    val x = ⌜x⌝ : TERM
│    val y = ⌜y⌝ : TERM
│    val z = ⌜z⌝ : TERM
```

The constructors *mk_∧* and *mk_⇒* construct conjunctions and implications respectively.

```
SML                                                   4
│    val t = mk_⇒(mk_∧(x,y),z);

│    val t = ⌜x ∧ y ⇒ z⌝ : TERM
```

5.2 Terms

There are only four different kinds of terms:

1. Variables.
2. Constants.
3. Function applications: ⌜t_1 t_2⌝
4. λ-abstractions: ⌜λ x • t ⌝.

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

```
SML                                                          5
│     ⌜¬x⌝;

│     val it = ⌜¬ x⌝ : TERM

SML
│     dest_¬ it;

│     val it = ⌜x⌝ : TERM

SML
│     type_of it;

│     val it = ⌜:BOOL⌝ : TYPE

SML
│     ⌜x⌝;

│     val it = ⌜x⌝ : TERM

SML
│     type_of it;

│     val it = ⌜:'a⌝ : TYPE
```

In the first case, the HOL type checker used the known type *BOOL* → *BOOL* of ¬ to deduce that the variable x must have type *BOOL*. In the second case, it assigns the most general type to x. The 'scope' of type information for type checking is a single quotation, so a type in one quotation cannot affect the type checking of another. If there is not enough contextually-determined type information to resolve the types of all variables in a quotation, then it may be necessary to explicitly indicate the required types by using ⌜*term*:*type*⌝ as illustrated below.

```
SML                                                          6
│     ⌜x:ℕ⌝;

│     val it = ⌜x⌝ : TERM

SML
│     type_of it;

│     val it = ⌜:ℕ⌝ : TYPE
```

Functions have types of the form $\sigma_1 \rightarrow \sigma_2$, where σ_1 and σ_2 are the types of elements of the domain and range of the function, respectively.

Before considering an example of the types of functions, an aside is appropriate on a purely syntactic matter. Functions may be defined with a special lexical status, such as being an infix operator, in the case of + or ∧. In such cases, putting $ in front of the name of the function causes the parser to ignore any special syntactic status it may have. This means that the naked symbol ∧ is not in itself a syntactically well-formed expression, because it denotes the application of the function to arguments which are missing. However the expression $∧ is well-formed in itself, denoting a function, and it can be applied to arguments.

```
SML                                                        7
│    ⌜∧⌝;

│    Syntax error in: ⌜  <?> ∧
│    ∧ is not expected after ⌜
│    Exception− Fail * Syntax error [HOL−Parser.19000] * raised

SML
│    ⌜$∧⌝;

│    val it = ⌜$∧⌝ : TERM

SML
│    type_of it;

│    val it = ⌜:BOOL → BOOL → BOOL⌝ : TYPE

SML
│    ⌜$∧ t1 t2⌝;

│    val it = ⌜t1 ∧ t2⌝ : TERM
```

After that aside, we return now to the subject of the types of functions. Functions can be denoted by Lambda-terms (or λ-terms). For example, ⌜$\lambda x\bullet\ x{+}1$⌝ is a term that denotes the function which maps a number x to a number $x + 1$, and is thus of type $\mathbb{N} \rightarrow \mathbb{N}$.

8

SML
```
│    ⌜λx • x+1⌝;
```

```
│    val it = ⌜λ x• x + 1⌝ : TERM
```

SML
```
│    type_of it;
```

```
│    val it = ⌜:ℕ → ℕ⌝ : TYPE
```

The next box provides further examples of metalanguage and object-language types.

9

SML
```
│    ⌜(x+1), (t1⇒t2)⌝;
```

```
│    val it = ⌜(x + 1, t1 ⇒ t2)⌝ : TERM
```

SML
```
│    type_of it;
```

```
│    val it = ⌜:ℕ × BOOL⌝ : TYPE
```

SML
```
│    (⌜x=1⌝, ⌜t1⇒t2⌝);
```

```
│    val it = (⌜x = 1⌝, ⌜t1 ⇒ t2⌝) : TERM * TERM
```

SML
```
│    (type_of(fst it), type_of(snd it));
```

```
│    val it = (⌜:BOOL⌝, ⌜:BOOL⌝) : TYPE * TYPE
```

The types of constants are declared in *theories*; this is described in Section 5.4.

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

```
SML                                                              10
│    ⌜1 2⌝;

│    Type error in ⌜1 2⌝
│    The operator must have type σ → τ
│    Cannot apply ⌜1:ℕ⌝
│             to ⌜2:ℕ⌝
│    Exception− Fail ∗ Type error [HOL−Parser.16000] ∗ raised
```

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

```
SML                                                              11
│    ⌜¬1⌝;

│    Type error in ⌜¬ 1⌝
│    The operator and the operand have incompatible types
│    Cannot apply ⌜¬:(BOOL→BOOL)⌝
│             to ⌜1:ℕ⌝
│    Exception− Fail ∗ Type error [HOL−Parser.16000] ∗ raised
```

5.3 Boolean Terms, Theorems and Sequents

So far, in the language of HOL terms, we have seen terms of different object-language types, including those of object-language type ⌜:*BOOL*⌝. The ProofPower system supports a process of inference which results in the production of *theorems*. Theorems are objects of metalanguage type *THM*. Terms are not theorems, that is, the metalanguage types *TERM* and *THM* are distinct. The form taken by a theorem in this system of inference is not simply a boolean-valued term but rather a composite of:

- a list of assumptions, each of which is a boolean-valued term
- a conclusion, which is a single boolean-valued term.

The following session produces an example of a theorem to illustrate this structure of assumptions and conclusion. The example is produced by means which are yet to be described, but will be covered in following sections.

```
SML                                                              12
│    tac_proof (([⌜x=y⌝, ⌜y=z⌝], ⌜x=z⌝), (asm_rewrite_tac[ ]));

│    val it = x = y, y = z ⊢ x = z : THM
```

It can be seen that the turnstile symbol, ⊢, separates assumptions from conclusion. This theorem can be understood as meaning: on the assumption that x=y and the further assumption that y=z, it may be concluded that x=z. The theorem is about the relationship between assumptions and conclusion (that the latter follows from the former). The "truth" of the theorem is the truth of an assertion about what follows from what.

Strictly speaking, all theorems in this system are about the relationship between assumptions and conclusions, but in practice many theorems have no assumptions. Here is another example of a theorem produced by means yet to be described:

```
SML                                                                  13
│     refl_conv ⌜x⌝;

│     val it = ⊢ x = x : THM
```

This theorem can be understood as meaning "without making any assumptions, it may be concluded that x=x". Here the list of assumptions mentioned above is present, but is empty and so nothing is displayed for it.

Terms can be constructed at will, (subject only to the constraint of being well-typed.) On the other hand, theorems can be constructed only by a proof which appeals to the rules of inference supported by the system. The soundness of the system of inference and the correctness of the implementation guarantee the "truth" of any theorems produced, and ensure that theorems can only be produced by the prescribed system of inference.

Objects structured according to the pattern described above as a list of assumptions followed by a conclusion are called "sequents". In this sense, theorems may be called sequents, so that the ProofPower system of inference is described as a sequent calculus; see e.g. [1].

The system supports "sequents" by providing, as an abbreviation for *TERM list ∗ TERM*, the name *SEQ*. Sequents in this other sense are NOT theorems, just data-structures. Their usefulness is (as shown in the example of producing the first theorem above) in convenience in stating goals for a proof process, so much so that the system also supports the abbreviation `GOAL` for the same type. This is illustrated in the next session, where the same object is ascribed a type which is reported in three different ways.

```
SML                                                                  14
│     val s =([⌜x=y⌝, ⌜y=z⌝], ⌜x=z⌝);

│     val s = ([⌜x = y⌝, ⌜y = z⌝], ⌜x = z⌝) : TERM list * TERM

SML
│     s:SEQ;

│     val it = ([⌜x = y⌝, ⌜y = z⌝], ⌜x = z⌝) : SEQ

SML
│     s:GOAL;

│     val it = ([⌜x = y⌝, ⌜y = z⌝], ⌜x = z⌝) : GOAL
```

5.4 The Development of Theories

5.4.1 Theories

The objects generated by work with ProofPower – definitions, types, constants, axioms and theorems – are organised into larger units called *theories*. A theory in ProofPower is similar to what a logician would call a theory, but there are some differences arising from the needs of mechanical proof. A HOL theory, like a logician's theory, contains sets of types, constants, definitions and axioms. In addition, however, a HOL theory may contain an explicit list of theorems that have been proved from the axioms and definitions. Logicians normally do not need to distinguish theorems that have actually been proved from those that could be proved, hence they do not normally consider sets of proven theorems as part of a theory; rather, they take the theorems of a theory to be the (often infinite) set of all consequences of the axioms and definitions. Another difference between logicians' theories and HOL theories is that, for logicians, theories are relatively static objects, but in ProofPower they can be developed over a period of time. For example, further theorems can be proved to produce a new version of a theory which replaces the previous version.

The purpose of the ProofPower system may be described as to provide tools to enable well-formed theories to be constructed. All the theorems of such theories are logical consequences of the definitions and axioms of the theory. The ProofPower system ensures that only well-formed theories can be constructed by allowing theorems to be created by *formal proof* only.

In general, a new theory is not constructed in a vacuum, but rather in a context of prior theories, which makes available the contents of the prior theories for use in the new theory. Thus theories are related one to another as parent to child, so that the parent is logically (but not physically) incorporated into the child.

Any new theory must be a child of an existing theory, and in fact may be a child of several different parent theories simultaneously. A collection of theories organised in a parent-child relationship is called a theory-hierarchy. The ProofPower system as issued contains a theory-hierarchy of approximately 20 theories. Of these, the theory called 'min' (for 'minimal') is the ultimate ancestor of all other theories, whether issued or user-defined. Each theory is devoted to a particular subject, so that there is, for example a theory of numbers in the issued database.

5.4.2 Theory Databases

A given theory is stored in what is called a theory database, which is a file in the filing system of the computer. Thus a theory database is what is stored between sessions of interaction with the ProofPower system.

In principle a whole hierarchy of theories can be stored in a single theory database. In practice however it may be more convenient to distribute a theory-hierarchy over several databases. For this purpose, databases may be organised in a parent-child relationship. Here each child database has exactly one parent.

Such an arrangement would allow a collection of theories in a common database to be read-only, and other theories under development to be in updatable child databases. Similar arrangements are possible within a single database: an individual theory may have a status of "locked" to prevent casual changes. Thus facilities for the management of theories are available both at the level of the individual theory and at the level of the database.

5.4.3 The Current Theory and Current Database

Any single ProofPower session works with a single database, the "current" database, which is that nominated in the UNIX command line which caused entry to the ProofPower system.

There is always a *current theory*: definitions and theorems are stored in the theory which is current at the time the definitions or theorems are generated. Each database has a theory which by default becomes the current theory immediately on entry to a session with that database.

Facilities for working with theories include the following:

print_status();	displays the name of the current theory and other information.
print_theory"*X*";	displays the contents of the theory named X. As a convenience, the current theory may be referred to by the name "-".
new_theory"*X*";	will create a new, empty theory, named X, which becomes current, being a child of the hitherto-current theory
open_theory"*X*";	will cause the existing theory X to become current.
new_parent"*X*";	will cause the current theory to acquire an additional parent, namely theory X.

5.4.4 Naming of Object

It has been explained that the state of a ProofPower session can be saved, and then retrieved on a later occasion. Within the state of the ProofPower session, there will be theorems and other objects: axioms, definitions, constants and so on. Now a theorem, for example, can be associated with a name, in the state of the ProofPower session, in either or both of two ways, which are distinct.

Firstly, a theorem is an ML value like any other, in that it can be associated with an ML name by the familiar process, seen many times above, of making a declaration:

```
SML                                                            15
│ val thm99 = refl_conv ⌜x⌝;

│ val thm99 = ⊢ x = x : THM
```

The value, and the association with the name, will survive the saving and retrieving of the state of the ProofPower session.

Secondly, the current theory is represented by a data structure within the state of the current session. This data structure has no ML name, but is instead provided with a number of access functions by which its contents may be inspected, extracted, and updated.

For example, the function **print_theory** enables the content of the theory to be inspected. There is a function **save_thm** which takes two arguments, a string and a theorem, and causes the theorem to be saved in the data structure which is the current theory under the name given by the string. A name given by such a string is called a key. The theorem can be recovered by another access function, **get_thm**, which takes as arguments a theory name (the current theory can be referred to by the name "−") and the key under which the theorem was stored. Note that there is no necessary connection between this string and the name of any ML variable used to hold a theorem. To emphasize the point, note that the key need not be a well-formed name.

```
SML                                                              16
│     save_thm ("theorem of 5 September 91", thm99);

│     val it = ⊢ x = x : THM

SML
│     get_thm "−" "theorem of 5 September 91";

│     val it = ⊢ x = x : THM
```

5.4.5 Example of Developing a New Theory

In this section an example is given of developing a new theory, which is chosen to be a treatment of Peano's postulates as axioms for the natural numbers. It is to be noted that there is already a theory built into ProofPower, called ℕ, which covers natural numbers and arithmetic, (in which Peano's postulates are in fact derived theorems rather than postulated as axioms). To emphasize that this example theory is just an example, and has no relation to ℕ except superficial resemblance, the example theory will be called *Peanissimo*.

Executing *new_theory* "*thy*" creates a new theory called *thy*; it fails if there already exists a theory so named in the current theory hierarchy.

```
SML                                                              17
│     new_theory "Peanissimo";

│     val it = () : unit
```

This starts a theory called *Peanissimo*, which is to be made into a theory containing Peano's postulates as axioms for the natural numbers. These postulates, stated informally, are:

P1 There is a number which we will call *zero*.

P2 There is a function which we will call *successor* such that if *n* is a number then the successor of *n* is a number.

P3 *zero* is not the *successor* of any number.

P4 If two numbers have the same *successor* then the numbers are equal.

P5 If a property holds of *zero*, and if whenever it holds of a number then it also holds of the *successor* of that number, then the property holds of all numbers. This postulate is called *Mathematical Induction*.

To formalize this in HOL a new type is introduced called *nat* (for natural number)

```
SML                                                                18
│     new_type ("nat", 0);

│     val it = ⌜:nat⌝ : TYPE
```

In general *new_type* ("*op*" *n*) makes *op* a new *n*-ary type operator in the current theory. Constant types (such as *BOOL* or ℕ) are regarded as degenerate type operators with no arguments, thus the new type *nat* is declared to be a *0*-ary type operator. An example of a *1*-ary type operator is *LIST*, occurring in for example ⌜[*a*;*b*;*c*] : ℕ *LIST*⌝; and an example of a 2-ary type operator is × occurring in for example ⌜(*x*,*y*) : *BOOL* × ℕ⌝;.

The axioms **P1** and **P2** can now be formalized by declaring two new constants to represent *zero* and *successor*.

Evaluating *new_const*("*c*", *σ*) makes *c* a new constant of type *σ* in the current theory. This fails if there already exists a constant named *c* in the current theory (or a parent of the current theory).

```
SML                                                                19
│     new_const ("zero", ⌜:nat⌝);

│     val it = ⌜zero⌝ : TERM

SML
│     new_const ("successor", ⌜:nat→nat⌝);

│     val it = ⌜successor⌝ : TERM
```

The HOL type checker ensures that **P1** and **P2** hold. **P3** is now asserted as an axiom:

```
SML                                                                20
│      new_axiom(["P3"], ⌜∀n• ¬(zero = successor n)⌝ );

│      val it = ⊢ ∀ n• ¬ zero = successor n : THM
```

This creates an axiom in the current theory (that is, in *Peanissimo*) called *P3*. Axiom *P4* can be declared similarly:

```
SML                                                                21
│      new_axiom(["P4"], ⌜∀m n •(successor m = successor n) ⇒ (m = n)⌝);

│      val it = ⊢ ∀ m n• successor m = successor n ⇒ m = n : THM
```

The final Peano axiom is Mathematical Induction:

SML 22

```
new_axiom(["P5"],⌜∀ P• P zero ∧ (∀ n • P n ⇒ P(successor n)) ⇒ (∀n• P n)⌝);

val it = ⊢ ∀ P• P zero ∧ (∀ n• P n ⇒ P (successor n)) ⇒ (∀ n• P n) : THM
```

To inspect the theory, the function *print_theory* can be used:

```
SML                                                              23
print_theory "−";


        === The theory Peanissimo ===

        −−− Parents −−−

                      demo

        −−− Constants −−−

        zero          nat
        successor     nat → nat

        −−− Types −−−

        nat

        −−− Axioms −−−

        P3             ⊢ ∀ n• ¬ zero = successor n
        P4             ⊢ ∀ m n
                  • successor m = successor n ⇒ m = n
        P5             ⊢ ∀ P
                         • P zero
                              ∧ (∀ n• P n ⇒ P (successor n))
                            ⇒ (∀ n• P n)


        === End of listing of theory Peanissimo ===
```

To end the session and make an update to the database in use, recording all the work of the session including the new theory, the current state of the session is saved to the database, by executing *save_and_quit*();.

```
SML                                                              24
    save_and_quit();


/par20/users/rda/tmp/sun4demo.db:131072 bytes
Closing /par20/users/rda/tmp/sun4demo.db now
Opening /par20/users/rda/tmp/sun4demo.db
```

The preceding session set up a first version of a theory, *Peanissimo*. It is usual to include in ‘Peano arithmetic’ axioms defining addition and multiplication. To do this a new session can be started and the theory further developed.

If you are using xpp and it is still running, you can start the new session by selecting the 'Restart' item from the Command Menu in xpp. Otherwise start a new session from UNIX as explained in section 1.3 (or 1.9, if you are not using xpp). You should now be in a position to continue developing the theory by issuing the ProofPower-ML command.

```
SML                                                              25
│     open_theory "Peanissimo";
```

The two new axioms can now be added, but first constants must be declared to represent addition and multiplication. Let us choose the names *pplus* and *ptimes* respectively for these. Since we wish to use these syntactically in the same way as + and ∗, that is, as infix operators with appropriate values for syntactic precedence, they are declared as such with *fixity* declarations **declare_infix** followed by **new_const**. Constants declared with *declare_infix* must have a type of the form $\sigma_1 \rightarrow \sigma_2 \rightarrow \sigma_3$.

```
SML                                                              26
│     declare_infix (300, "pplus");
│     declare_infix (310, "ptimes");

│     val it = () : unit
│     val it = () : unit

SML
│     new_const ("pplus", ⌜:nat→nat→nat⌝);
│     new_const ("ptimes", ⌜:nat→nat→nat⌝);
│

│     val it = ⌜$pplus⌝ : TERM
│     val it = ⌜$ptimes⌝ : TERM
```

Axioms defining *pplus* and *ptimes* can now be given.

```
SML                                                                      27
│    new_axiom(["pplus_def"],
│       ⌜(∀n• (zero pplus n) = n) ∧
│        (∀m n•((successor m) pplus n) = successor (m pplus n) )⌝ );

│    val it = ⊢ (∀ n• (zero pplus n) = n) ∧
│              (∀ m n• (successor m pplus n) = successor (m pplus n)) : THM

SML
│    new_axiom(["ptimes_def"],
│       ⌜(∀n• (zero ptimes n) = zero) ∧
│        (∀m n•((successor m) ptimes n) = ((m ptimes n) pplus n)  )⌝ );

│    val it = ⊢ (∀ n• (zero ptimes n) = zero) ∧
│              (∀ m n• (successor m ptimes n) = (m ptimes n pplus n)) : THM
```

The theory *Peanissimo* has now been extended to contain the new definitions.

This example shows how a theory is set up. How to prove consequences of axioms and definitions is described later. The ProofPower system contains a built-in theory of numbers called ℕ which contains Peano's postulates and the definitions of addition (+) and multiplication (∗) amongst others. In fact, Peano's postulates are theorems not axioms in the theory ℕ. The constants *0* and *Suc* (corresponding to *zero* and *successor* in *Peanissimo*) are *defined* in terms of purely logical notions. In HOL, *definitions* are a special kind of axiom that are guaranteed to be consistent. The commonest (but not only) form of a definition is:

$$f\ x_1\ \dots\ x_n = t$$

where f is declared to be a new constant satisfying this equation (and t is a term whose free variables are included in the set $\{x_1, \dots, x_n\}$). Such definitions cannot be recursive because, for example:

$$f\ x = (f\ x) + 1$$

would imply $0 = 1$ (subtract $f\ x$ from both sides) and is therefore inconsistent. An example of a definition is:

```
SML                                                                      28
│    simple_new_defn (["Double_def"], "Double", ⌜λx• (x pplus x)⌝);

│    val it = ⊢ Double = (λ x• x pplus x) : THM
```

This definition both declares *Double* as a new constant of the appropriate type and asserts the defining equation as a definitional axiom.

5.5 Constant Specification

There is an alternative form of introduction of constants, called specification, which involves predicates not restricted to the definitional form *name = value*, and therefore raising the question of possible inconsistency. Thus, in general, a specification will incur a proof obligation: a proof must be provided that there exists something which satisfies the predicate. A complete discussion of this topic is given in section 9, deferred until after the discussion of proof techniques.

However, in certain cases, the system is able to perform the existence-proof automatically. These cases include the definitional *name* = *value* form, and also simple predicates such as T. This means that the mechanism for the specification of constants can be used uniformly for both specification and definition.

Associated with the constant-specification mechanism is a facility for a graphic display. In the source-file of a document typeset with Latex, the characters which cause a display such as the following:

```
HOL Constant
│Square : nat → nat
├──────────────────────────────
│Square = λx• (x ptimes x)
```

can be pasted directly into the ProofPower window. The source-file characters are typed as:

```
│      ⓈHOLCONST
│      Square : nat → nat
│      ├
│      Square = λx• (x ptimes x)
│      ■
```

Entering these characters is equivalent making use of the function **const_spec** by entering:

```
│      const_spec (
│        ["Square"],
│        [⌜Square : nat → nat⌝],
│        ⌜Square = λx• (x ptimes x)⌝ );
```

In this tutorial, an occurrence of a display of this kind puts a strain on the convention we have followed, of showing system input and output in session-boxes, character by character. Such a display is meant to be understood as being in a small session-box of its own, which represents some input.

If the theory is now examined, the treatment of *Double* and *Square* can be compared:

```
SML                                                                 29
print_theory "-";

        === The theory Peanissimo ===

        --- Parents ---
                        demo

        --- Constants ---
        zero            nat
        successor       nat → nat
        $pplus          nat → nat → nat
        $ptimes         nat → nat → nat
        Double          nat → nat
        Square          nat → nat

        --- Types ---
        nat

        --- Fixity ---
        Infix 300:      pplus
        Infix 310:      ptimes

        --- Axioms ---
        P3              ⊢ ∀ n• ¬ zero = successor n
        P4              ⊢ ∀ m n
                          • successor m = successor n ⇒ m = n
        P5              ⊢ ∀ P
                          • P zero
                                ∧ (∀ n• P n ⇒ P (successor n))
                              ⇒ (∀ n• P n)
        pplus_def       ⊢ (∀ n• zero pplus n = n)
                            ∧ (∀ m n
                            • successor m pplus n
                                = successor (m pplus n))
        ptimes_def      ⊢ (∀ n• zero ptimes n = zero)
                            ∧ (∀ m n
                            • successor m ptimes n
                                = m ptimes n pplus n)

        --- Definitions ---
        Double_def      ⊢ Double = (λ x• x pplus x)
        Square          ⊢ Square = (λ x• x ptimes x)

        === End of listing of theory Peanissimo ===
```

To repeat the point made earlier, the theory *Peanissimo* is presented here solely as a small example of the development of a theory. In one important respect it is atypical, and that is in the introduction of axioms. The use of axioms, as illustrated here, carries considerable danger in general because it is very easy to assert inconsistent axioms. It is thus safer to use only definitions.

A theory containing only definitions is called a *definitional theory*. A number of useful definitional theories are built-in to the ProofPower system, and are shown in the ProofPower *Reference Manual* [12]. Examples include theories of numbers, sets, pairs and lists. Indeed it is particularly important to note that, with a single exception, *all* the built-in theories are purely definitional. The exception is the built-in theory *init* which contains the five primitive axioms of HOL. By inspecting the theories listed in the ProofPower *Reference Manual* [12], it may be seen that *init* is the only theory containing axioms, and all else is built up by a process of definition.

This topic is covered in section 9 below. It is noteworthy that if consistency is to be achieved by avoiding the use of axioms then a price must be paid which amounts to doing proofs. Further coverage of specification is thus deferred until after the coverage of proof.

INTRODUCTION TO PROOF WITH ProofPower

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

Theorems are represented in HOL by values of an abstract type called *THM*. The only way to create theorems is by proof. In ProofPower (following LCF, [2]), 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 can be retrieved from the theory *init* with the function **get_axiom**. For example, the Law of Excluded Middle can be retrieved with the key "*bool_cases_axiom*":

```
SML                                                          30
│    get_axiom "init" "bool_cases_axiom";

│    val it = ⊢ ∀ b• (b ⇔ T) ∨ (b ⇔ F) : THM
```

Theorems are printed with a turnstile ⊢ as illustrated above. 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 notation this might be:

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

This means that a theorem of the form below the line may be inferred from a theorem of the form above the line. Here Γ represents the assumptions, which must be the same in the inferred theorem as in the premise, and $t[t'/x]$ represents 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.

A rule very similar to this is represented in ML by a function ∀_*elim*[1] which, when given as arguments a term ⌜a⌝ and a theorem $\vdash \forall x \bullet t[x]$, returns the theorem $\vdash t[a]$, the result of substituting a for x in $t[x]$.

[1] This function is not a primitive rule of inference in the HOL logic, but is a derived rule. Derived rules are described in Section 6.1.

1

SML

```
val Th1 = get_axiom "init" "bool_cases_axiom";
```

```
val Th1 = ⊢ ∀ b• (b ⇔ T) ∨ (b ⇔ F) : THM
```

SML

```
val Th2 = ∀_elim ⌜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 ∀_*elim*; it interactively performs the following proof:

1. $\vdash\ \forall t.\ t = T \quad \vee \quad t = F$ [Axiom *bool_cases_axiom*]
2. $\vdash\ (1{=}2) = T \quad \vee \quad (1{=}2) = F$ [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.

A proof in the ProofPower 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 ProofPower system are described later.

The general form of a theorem is $t_1, \ldots, t_n \vdash t$, where $t_1, \ldots, t_n$ 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 $(t_1 \wedge \ldots \wedge t_n) \Rightarrow t$. Theorems with no assumptions are displayed in the form $\vdash t$.

Every value of type *THM* in the ProofPower system can be obtained by repeatedly applying **inference rules** to axioms.

Every **inference rule** is either a **derived rule** or else a **constructor of the abstract data type THM**.

Every **derived rule** is a procedure which invokes other rules each time the derived rule is invoked. Some derived rules are supplied as part of ProofPower and others may be user-defined.

Every rule which is a **constructor** is either a **primitive rule** or else a **built-in rule** or else a **definition schema**. The collection of constructor rules is fixed.

Every **built-in rule** can in principle be defined as a derived rule in terms of the primitive rules, but for efficiency reasons is not implemented in this way.

Every **definition schema** is justified, not in terms of the primitive rules, but rather in terms of a principle of definitional extension.

In the rest of this section, the process of *forward proof*, which has been sketched above, is decribed in more detail. In section 7 below, *goal directed proof* is described. Goal directed proof provides additional facilities for interactive proof development which makes it suitable as the most common mode of working with ProofPower.

6.1 Forward proof

Three of the primitive inference rules of the HOL logic are

- *asm_rule* (assumption introduction),
- *⇒_intro* (discharging, that is, eliminating, an assumption by introducing an implication) and
- *⇒_elim* (eliminating an implication, that is, Modus Ponens).

These rules will be used to illustrate forward proof and the writing of derived rules. The inference rule *asm_rule* generates theorems of the form $t \vdash t$. The function *dest_thm* decomposes a theorem into a pair consisting of list of assumptions and the conclusion. The ML type *SEQ*, or *GOAL*, abbreviates *TERM list ∗ TERM*; this is motivated in Section 7.

```
SML                                                          2
│   val Th3 = asm_rule ⌜t1⇒t2⌝;

│   val Th3 = t1 ⇒ t2 ⊢ t1 ⇒ t2 : THM

SML
│   dest_thm Th3;

│   val it = ([⌜t1 ⇒ t2⌝], ⌜t1 ⇒ t2⌝) : SEQ
```

The primitive inference rule *⇒_intro* (discharging, assumption elimination) infers from a theorem of the form $\cdots t_1 \cdots \vdash t_2$ the new theorem $\cdots \; \cdots \vdash t_1 \Rightarrow t_2$. *⇒_intro* 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:

```
SML                                                          3
│   val Th4 = ⇒_intro ⌜t1⇒t2⌝ Th3;

│   val Th4 = ⊢ (t1 ⇒ t2) ⇒ t1 ⇒ t2 : THM
```

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

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

Corresponding to Modus Ponens, the ML function *⇒_elim* takes argument theorems of the form $\cdots \vdash t_1 \Rightarrow t_2$ and $\cdots \vdash t_1$ and returns $\cdots \vdash t_2$. The next session illustrates the use of *⇒_elim* and illustrates also a common error, namely not supplying the HOL logic type checker with enough information.

```
SML                                                                4
│   val Th5 = asm_rule ⌜t1⌝;

│   Exception− Fail * ⌜t1⌝ is not of type ⌜:BOOL⌝ [asm_rule.3031] * raised

SML
│   val Th5 = asm_rule ⌜t1:BOOL⌝;

│   val Th5 = t1 ⊢ t1 : THM

SML
│   val Th6 = ⇒_elim Th3 Th5;

│   val Th6 = t1 ⇒ t2, t1 ⊢ t2 : THM
```

The assumptions of *Th6* can be extracted with the ML function *asms*, which returns the list of assumptions of a theorem. The conclusion of a theorem is returned by the function *concl*.

```
SML                                                                5
│   asms Th6;

│   val it = [⌜t1 ⇒ t2⌝, ⌜t1⌝] : TERM list

SML
│   concl Th6;

│   val it = ⌜t2⌝ : TERM
│
```

Discharging *Th6* twice establishes the theorem ⊢ *t1* ⇒ (*t1*⇒*t2*)⇒*t2*.

```
SML                                                                6
│   val  Th7 = ⇒_intro ⌜t1⇒t2⌝ Th6;

│   val Th7 = t1 ⊢ (t1 ⇒ t2) ⇒ t2 : THM

SML
│   val  Th8 = ⇒_intro ⌜t1:BOOL⌝ Th7;

│   val Th8 = ⊢ t1 ⇒ (t1 ⇒ t2) ⇒ t2 : THM
```

The sequence: *Th3*, *Th5*, *Th6*, *Th7*, *Th8* constitutes a proof in HOL of the theorem

⊢ *t1*⇒(*t1*⇒*t2*)⇒*t2*

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]

6.2 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]

Note that line 3 above mentions 'discharging' the assumption t', but t' is not actually amongst the assumptions. The rule $\Rightarrow$_*intro* does not in fact require its term argument (t') to be present in the assumptions of its theorem argument (line 2).

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 $\Gamma,\ t' \ \vdash \ t$).[2] For example, the next session uses this proof to add the hypothesis *t3* to *Th6*.

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

```
SML                                                              7
│   val  Th9 = asm_rule ⌜t3:BOOL⌝;

│   val Th9 = t3 ⊢ t3 : THM

SML
│   val  Th10 = ⇒_intro ⌜t3:BOOL⌝ Th6;

│   val Th10 = t1 ⇒ t2, t1 ⊢ t3 ⇒ t2 : THM

SML
│   val Th11 = ⇒_elim 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. An example of definition of a derived rule will now be given. A rule, called, say, *ADD_ASSUM*, will 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:

```
SML                                                              8
│   fun  ADD_ASSUM t th =
│    let val th9  = asm_rule t
│        val th10 = ⇒_intro t th
│    in
│    ⇒_elim th10 th9
│    end;

│   val ADD_ASSUM = fn : TERM −> THM −> THM

SML
│   ADD_ASSUM ⌜t3:BOOL⌝ Th6;

│   val it = t1 ⇒ t2, t1, t3 ⊢ t2 : THM
```

The body of *ADD_ASSUM* has been coded to mirror the proof done in session 9 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:

```
SML                                                                    9
│    fun  ADD_ASSUM t th = ⇒_elim (⇒_intro t th) (asm_rule t);

│    val ADD_ASSUM = fn : TERM -> THM -> THM

SML
│    ADD_ASSUM ⌜t3:BOOL⌝ Th6;

│    val it = t1 ⇒ t2, t1, t3 ⊢ t2 : THM
```

As another example of a derived inference rule, one which moves the antecedent of an implication to the assumptions, is shown below as *UNDISCH*.

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

An ML derived rule that implements this is:

```
SML                                                                    10
│    fun  UNDISCH th =  ⇒_elim th (asm_rule(fst(dest_⇒(concl th))));

│    val UNDISCH = fn : THM -> THM

SML
│    Th10;

│    val it = t1 ⇒ t2, t1 ⊢ t3 ⇒ t2 : THM

SML
│    UNDISCH Th10;

│    val it = t1 ⇒ t2, t1, 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]

Rules equivalent to *ADD_ASSUM* and *UNDISCH* (named respectively *asm_intro* and *undisch_rule*) are derived rules defined when the ProofPower system is built.

6.3 Rewriting

An important derived rule is *rewrite_rule*. This takes as arguments

- a collection of equations represented by a list of theorems, such that each theorem is an equation or a conjunction of equations, and
- a theorem $\Delta \vdash t$

and repeatedly replaces in t instances of the lefthand side of an equation by the corresponding instance of the righthand side 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, and Γ is the union of the assumptions in the equations.

The session below illustrates the use of *rewrite_rule*. In it the list of equations is a list *rewrite_list* containing the theorems of the theory ℕ defining addition and multiplication.

```
SML                                                              11
|     val rewrite_list = map (get_defn "ℕ") ["+", "∗"];


|     val rewrite_list = [
|        ⊢ ∀ m n• 0 + n = n ∧
|                 (m + 1) + n = (m + n) + 1 ∧
|                 Suc m = m + 1,
|        ⊢ ∀ m n• 0 ∗ n = 0 ∧
|                (m + 1) ∗ n = m ∗ n + n] : THM list
```

In the following example, the conclusion of a theorem (an arbitrary theorem just for this example) is rewritten using these definitions to produce a simpler theorem.

```
SML                                                              12
|     val th = asm_rule ⌜(0 + m) = ((0∗ n) +1)⌝;


|     val th = 0 + m = 0 ∗ n + 1 ⊢ 0 + m = 0 ∗ n + 1 : THM

SML
|     rewrite_rule rewrite_list th;


|     val it = 0 + m = 0 ∗ n + 1 ⊢ m = 1 : THM
```

rewrite_rule is not a primitive in HOL, but is a derived rule. In addition to the equations given explicitly as an argument, *rewrite_rule* makes use of equations in the supplied theories, as shown in the following example of rewriting with an empty list as argument:

13

SML

```
(asm_rule ⌜(T ∧ x) ∨ F ⇒ F⌝);
```

```
val it = T ∧ x ∨ F ⇒ F ⊢ T ∧ x ∨ F ⇒ F : THM
```

SML

```
rewrite_rule [ ] it;
```

```
val it = T ∧ x ∨ F ⇒ F ⊢ ¬ x : THM
```

There are powerful facilities in ProofPower for producing customized rewriting tools which scan through terms in user programmed orders; *rewrite_rule* is the tip of an iceberg.

GOAL ORIENTED PROOF

The style of forward proof described in the previous chapter is unnatural and too laborious for many applications. This chapter covers the topic of an alternative style, called ‘goal-oriented proof’, also known as ‘backward proof’ or ‘tactical proof’. In this style, interactive facilities are available to support the proof development process. These facilities are called ‘the subgoal package’. Before describing the subgoal package, the underlying concepts of goals and tactics are described.

7.1 Goals and Tactics

An important advance in proof generating methodology was made by Robin Milner in the early 1970s when he invented the notion of *tactics*. A conjecture, stated as a sequent, is called a ‘goal’ when it becomes a candidate for proving it to be a theorem. A tactic is a function which does two things:

- It decomposes a goal into one or more simpler goals, called subgoals.
- It keeps track of the reason why achieving the subgoal(s) will achieve the goal.

Consider, for example, the rule of ∧-introduction[1] shown below:

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

In HOL, ∧-introduction is represented by the ML function *∧_intro*, such that

$\land\_intro\ (\Gamma_1 \ \vdash \ t_1)\ (\Gamma_2 \ \vdash \ t_2)\ \ is\ (\Gamma_1 \cup \Gamma_2 \ \vdash \ t_1 \ \land \ t_2)$

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

```
SML                                                                    1
│     val  Th1 = asm_rule ⌜A:BOOL⌝ and Th2 = asm_rule ⌜B:BOOL⌝;

│     val Th1 = A ⊢ A : THM    val Th2 = B ⊢ B : THM

SML
│     val  Th3 = ∧_intro Th1 Th2;

│     val Th3 = A, B ⊢ A ∧ B : THM
```

Suppose the goal is to prove $A \ \land \ B$, 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 $\vdash \ A \ \land \ B$ can be deduced. Thus:

[1] In higher order logic this is a derived rule; in first order logic it is usually primitive. In HOL the rule is called *∧_intro*

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

(ii) The justification for the reduction of the goal $\vdash A \land B$ to the two subgoals $\vdash A$ and $\vdash B$ is the rule of $\land$-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 $t_1,\ldots,t_n \vdash t$. 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.

ML has a type abbreviating mechanism which is used to give mnemonic names to the various types associated with goal oriented proof. Some type abbreviations are as follows:

Abbreviation	**Type**
CONV	*TERM* −> *THM*
GOAL	(*TERM list*) ∗ *TERM*
PROOF	*THM list* −> *THM*
SEQ	(*TERM list*) ∗ *TERM*
TACTIC	*GOAL* −> (*GOAL list* ∗ *PROOF*)
THM_TACTIC	*THM* −> *TACTIC*
THM_TACTICAL	*THM_TACTIC* −> *THM_TACTIC*

The left hand side of these abbreviations can be used anywhere that the right hand side can.

If T is a tactic (i.e. an ML function of type *TACTIC*) and g is a goal (i.e. an ML value of type *GOAL*), then applying T to g (i.e. evaluating the ML expression $T\ g$) will result in an object of ML type *GOAL list* ∗ *PROOF*, that is, a pair whose first component is a list of goals and whose second component is a justification function, i.e. has ML type *PROOF*.

An example tactic is ∧_*tac*. For example, consider the trivial goal of showing $T \land T$, where T is a constant that stands for *true*:

```
SML                                                          2
│   val goal : GOAL =([ ], ⌜T ∧ T⌝);

│   val goal = ([ ], ⌜T ∧ T⌝) : GOAL
│

SML
│   ∧_tac goal;

│   val it = ([([ ], ⌜T⌝), ([ ], ⌜T⌝)], fn) : GOAL list ∗ PROOF
│

SML
│   val (goal_list,just_fn) = it;

│   val goal_list = [([ ], ⌜T⌝), ([ ], ⌜T⌝)] : GOAL list
│   val just_fn = fn : PROOF
```

Applying $\wedge$_*tac* has produced a goal list consisting of two identical subgoals, each of which is to show ([ ],⌜*T*⌝). Now, there is a preproved theorem in HOL, which is recorded in theory *misc* under the name of *t_thm*. It can be produced and bound to an ML name, say *TRUTH*, as follows:

```
SML                                                    3
|        val TRUTH = get_thm "misc" "t_thm";

|        val TRUTH = ⊢ 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:

```
SML                                                    4
|     just_fn [TRUTH, TRUTH];

|     val it = ⊢ T ∧ T : THM
```

Although this example is trivial, it does illustrate the essential idea of tactics.

7.1.1 Example of Defining a Tactic

Tactics are not special theorem-proving primitives. They are just ML functions. New tactics may be defined in terms of inference rules or (by means to be described below) by combining existing tactics. An example of the definition of a tactic equivalent to the built-in $\wedge$_*tac* would be:

```
|   fun ∧_tac_equivalent (asmlist, conjunct) =
|        let val (left, right) = dest_∧ conjunct
|        in
|        ([(asmlist,left), (asmlist,right)],
|         fn [th1, th2] => ∧_intro th1 th2)
|        end;
|
```

In this definition, the ML function *dest_*$\wedge$ splits a conjunctive term *conjunct* into its two conjuncts, *left* and *right*. If (*asmlist*, ⌜*left* $\wedge$ *right*⌝) is a goal, then $\wedge$_*tac_equivalent* splits it into the list of two subgoals (*asmlist*,⌜*left*⌝) and (*asmlist*,⌜*right*⌝).

The justification function, *fn* [*th1*, *th2*] => $\wedge$_*intro th1 th2*, takes a list [*th1*, *th2*] of theorems and applies the rule $\wedge$_*intro* to *th1* and *th2*.

It should be noted that there are facilities, described below, for defining new tactics by combining existing tactics

7.1.2 Effects of Tactics

To summarize: if *T* is a tactic and *g* is a goal, then applying *T* to *g* will result in an object of ML type *GOAL list* $*$ *PROOF*, i.e. a pair whose first component is a list of goals and whose second component is a justification function.

Suppose $T\ g = ([g_1, \ldots, g_n],\ p)$. The idea is that $g_1, \ldots, g_n$ are subgoals and p is a 'justification' of the reduction of goal g to subgoals $g_1, \ldots, g_n$. Suppose further that the subgoals $g_1, \ldots, g_n$ 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 $[th_1,\ \ldots,\ th_n]$ 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 tactic 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 $p\ [th_1, \ldots, th_n]$ 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 $t_1, \ldots, t_m \vdash t$ achieves a goal $([u_1, \ldots, u_n],\ u)$

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` being 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 generally fail (in the ML sense) if they are applied to inappropriate goals. For example, $\wedge$_*tac* will fail if it is applied to a goal whose conclusion is not a conjunction.

7.1.3 Notation for Specifying Tactics

Tactics are specified using the following notation:

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

For example, a tactic called $\wedge$_*tac* is described by

$$\frac{t_1 \wedge t_2}{\overline{t_1 \qquad t_2}}$$

Thus $\wedge$_*tac* reduces a goal of the form Γ, ⌜$t_1 \wedge t_2$⌝ 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 *induction_tac*, the tactic for doing mathematical induction on the natural numbers.

$$\frac{t[n]}{t[\mathtt{0}] \qquad \{t[n]\}\ t[\mathtt{Suc}\ n]}$$

Given the name of a variable, n say, which is to be the induction variable, *induction_tac* ⌜*n*:ℕ⌝ reduces a goal $(\Gamma,\ t[n])$ to

- a basis subgoal , $(\Gamma,\ t[0])$ and
- an induction step subgoal $(\Gamma \cup \{t[n]\}, t[n + 1])$. Here the set of assumptions are the original set Γ together with the extra assumption, written in the tactic-notation as a singleton set, $\{t[n]\}$

```
SML                                                              5
│      (induction_tac ⌜m:ℕ⌝)  ([ ], ⌜(m + n) = (n + m)⌝);

│      val it = ([      ([], ⌜0 + n = n + 0⌝),
│                       ([], ⌜(m + 1) + n = n + m + 1⌝)],
│                fn) : GOAL list * PROOF
```

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

7.2 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 *gl* to the empty list of goals, and
- bind *p* to a function which when applied to the empty list of theorems yields a theorem *th* achieving *g*.

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 built-in rewrites:

6

SML
```
val g = ([ ], ⌜T ∧ x ⇒ x ∨ (y ∧ F)⌝) : GOAL;
```
```
val g = ([ ], ⌜T ∧ x ⇒ x ∨ y ∧ F⌝) : GOAL
```

SML
```
val T = rewrite_tac [ ];
```
```
val T = fn : TACTIC
```

SML
```
val (gl, p) = T g;
```
```
val gl = [ ] : GOAL list val p = fn : PROOF
```

SML
```
val th = p[ ];
```
```
val th = ⊢ T ∧ x ⇒ x ∨ y ∧ F : THM
```

There is a useful built-in function **tac_proof** of ML type *GOAL ∗ TACTIC −> THM* such that *tac_proof* (*G*, *T*) proves the goal *G* using tactic *T* and returns the resulting theorem.

7.2.1 The Subgoal Package

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. ProofPower provides a package for building and traversing the tree of subgoals, stacking the justification functions and applying them properly; such a package was originally implemented for LCF by Larry Paulson.

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

The function **set_goal** of type *GOAL −> unit* initializes the subgoal package with a new **main goal** goal. It takes two arguments: a list of terms which are to be the assumptions and a term which is to be the conclusion. Usually main goals have no assumptions; the function *g* is useful in this case where g is defined by:

SML
```
fun g t = set_goal([ ],t);
```

To illustrate the facilities provided by the subgoal package the trivial theorem $m + 0 = m$ will be proved.

```
SML                                                   1
    g ⌜(m + 0) = m⌝;


    Now 1 goal on the main goal stack

    (* *** Goal "" *** *)

    (* ?⊢ *)  ⌜m + 0 = m⌝

    val it = (): unit
```

This sets up the goal. The system response consists of

- a display of the number of main goals now on the stack.
- A label for the goal. In this case the label is the empty string appearing between the “” marks.
- A display of the goal itself. The display consists of a list of assumptions, (there being none in this case), followed by the conclusion. The conclusion is marked by the symbols (∗ ?⊢ ∗)
- A display of the value returned by the *set_goal* (or g) function, which is always () : *unit*. Thus the preceding lines of the display produced are a side-effect of the function, not a returned value.

The next step is to choose a tactic and apply it to the goal. One of several possible approaches is to use induction to split the goal into a basis and step case. A suitable tactic is provided by **induction_tac**. Here we will induct on m so the tactic to be applied is *induction_tac* ⌜m:ℕ⌝.

To apply any tactic, use is made of the function **apply_tactic**. This frequently-used function is available under the alias **a**. It applies a tactic to the top goal on the stack, then pushes the resulting subgoals onto the goal stack, then prints the resulting subgoals. If there are no subgoals, the justification function is applied to the theorems solving the subgoals that have been proved and the resulting theorems are printed.

```
SML                                                        2
        a  (induction_tac ⌜m:ℕ⌝) ;


        Tactic produced 2 subgoals:

        (* *** Goal "2" *** *)

        (* ?⊢ *)  ⌜(m + 1) + 0 = m + 1⌝


        (* *** Goal "1" *** *)

        (* ?⊢ *)  ⌜0 + 0 = 0⌝

        val it = () : unit
```

The top of the goal stack is printed last. The basis case is an instance of the definition of addition, so is solved by rewriting with the equations for addition in the theory ℕ. These equations are amongst those used in rewriting by default, and so no explicit list of equations need be supplied:

```
SML                                                        3
        a (rewrite_tac [ ]);


        Tactic produced 0 subgoals:
        Current goal achieved, next goal is:

        (* *** Goal "2" *** *)

        (* ?⊢ *)  ⌜(m + 1) + 0 = m + 1⌝
```

The basis is solved and the goal stack popped so that its top is now the step case. This goal can be solved in the same way as the previous:

```
SML                                                        4
            a (rewrite_tac [ ]);



            Tactic produced 0 subgoals:
            Current and main goal achieved

```

The top goal (the step case) is solved , and since the basis is already solved, the main goal is solved. The theorem achieving the goal can be extracted from the subgoal package with **top_thm**, or with **pop_thm**: the former leaves the goal stack unchanged while the latter removes the goal from the stack.

```
SML                                                    5
      top_thm();


      val it = ⊢ m + 0 = m : THM
```

The order in which goals are worked on can be adjusted. Firstly the goal stack is backed up. The function *undo*takes an argument which is the number of steps by which to back up the goal-stack to a previous state: to go back to the point at which there were two subgoals will require undoing two steps:

```
SML                                                    6
          undo 2;


Current goal is:

(* *** Goal "1" *** *)

(* ?⊢ *)  ⌜0 + 0 = 0⌝


```

The system offers the basis case as the current subgoal. In order to survey all the possible subgoals, the command **print_goal_state** (*top_goal_state*()) is used:

```
SML                                                        7
        print_goal_state (top_goal_state());


        Main goal is:
        (* ?⊢ *) ⌜m + 0 = m⌝

        Goals to be proven are:

        (* *** Goal "1" *** *)

        (* ?⊢ *) ⌜0 + 0 = 0⌝


        (* *** Goal "2" *** *)

        (* ?⊢ *) ⌜(m + 1) + 0 = m + 1⌝


        Current goal is:

        (* *** Goal "1" *** *)

        (* ?⊢ *) ⌜0 + 0 = 0⌝

```

It can be seen that the current goal is labelled "1" and the other goal is labelled "2". To choose goal "2" to work on, it is made current with the command **set_labelled_goal**, providing an argument value of, in this case, "2".

```
SML                                                        8
    set_labelled_goal "2";


        Current goal is:

        (* *** Goal "2" *** *)

        (* ?⊢ *) ⌜(m + 1) + 0 = m + 1⌝

```

The top goal is now the step case not the basis case, so the tactic can be applied:

```
SML                                                          9
        a (rewrite_tac [ ]);


        Tactic produced 0 subgoals:
        Current goal achieved, next goal is:

        (* *** Goal "1" *** *)

        (* ?⊢ *) ⌜0 + 0 = 0⌝
```

These example have illustrated the working of the subgoal package, with multiple subgoals, using just two tactics, induction and rewriting. It may be noted that in fact rewriting alone is sufficient for this simple goal, to give a one-step proof:

```
SML                                                          10
        g ⌜(m + 0) = m⌝;


        Now 1 goal on the main goal stack

        (* *** Goal "" *** *)

        (* ?⊢ *) ⌜m + 0 = m⌝

SML
        a (rewrite_tac []);


        Tactic produced 0 subgoals:
        Current and main goal achieved
```

7.2.2 Multiple Main Goals

The subgoal package allows work on one main goal to be suspended(i.e. stacked) to work on another. The second goal can be quite independent of the first, although most use of this facility would be to prove a subsidiary theorem in the course of proving another.

To begin work on a second goal while suspending work on the first, the function **push_goal** is used rather than *set_goal* for stating the second goal.

It has already been mentioned that *pop_thm* can be used to retrieve a proved theorem from the topmost goal, and then discard that goal from the stack. Whatever the state of the proof , the topmost goal on the stack can be discarded by executing **drop_main_goal** ();

It may be noted that *set_goal* is equivalent to *drop_main_goal* followed by *push_goal*.

7.2.3 Working With Assumptions

The following example introduces two new tactics. The first of these is a general simplifying tactic called **strip_tac**. One of the effects of this tactic is to simplify the conclusion of the goal by replacing implications with assumptions. Other effects of **strip_tac** are described below, in section 8.1.

The second of the two new tactics is called **asm_rewrite_tac** which does everything that *rewrite_tac* does, but in addition uses the assumptions of the current goal as a source of rewriting equations, as well as any explicitly given as an argument, and the default equations of the built-in theories. Although *asm_rewrite_tac* does everything that *rewrite_tac* does, there is a purpose in retaining the two as separately available tactics, in that a greater degree of control is provided over which equations are used for rewriting on any occasion.

To illustrate:

```
SML                                                         11
        g ⌜P = Q ⇒ P x = Q x⌝;

        Now 1 goal on the main goal stack

        (* *** Goal "" *** *)

        (* ?⊢ *)  ⌜P = Q ⇒ P x = Q x⌝

SML
        a strip_tac;

        Tactic produced 1 subgoal:

        (* *** Goal "" *** *)

        (*  1 *)  ⌜P = Q⌝

        (* ?⊢ *)  ⌜P x = Q x⌝
```

Note that the goal is now displayed as a list of numbered assumptions followed by the conclusion. Here there is only one assumption, number 1. To continue:

```
SML                                                         12
        a (asm_rewrite_tac []);

        Tactic produced 0 subgoals:
        Current and main goal achieved
```

7.3 Tacticals

It is possible to do in one step the above proof by induction, by using a compound tactic built with the *tactical*[2] called **THEN**.

Tacticals are higher order operations for combining tactics. Thus a tactical is an ML function that returns a tactic (or tactics) as result. Tacticals may take various parameters; this is reflected in the various ML types that the built-in tacticals have. Some important tacticals in the ProofPower system are listed below.

7.3.1 The Tactical *THEN*

In the example above the tactic *induction_tac* was applied first. Then the tactic, *rewrite_tac* [] was applied to all (that is, both) the resulting subgoals.

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. The type of *THEN* is *TACTIC ∗ TACTIC −> TACTIC*.

To illustrate, the previous example will be done again with a one-step proof. (From now on the proof-sessions will be shown just in essentials, that is, omitting some of the annotations provided by the system.)

```
SML                                                        1
│      g ⌜(m + 0) = m⌝;

│      (∗ ?⊢ ∗)  ⌜m + 0 = m⌝

SML
│      a ((induction_tac ⌜m:ℕ⌝) THEN  (rewrite_tac [ ]));

│      Current and main goal achieved
```

This is typical: it is common to use a single tactic for several goals. A tactical similar to *THEN* is **THEN_LIST**. Whereas *THEN* applies the same tactic to all resulting subgoals, **THEN_LIST** applies the members of a list of tactics, taken in order, to corresponding subgoals.

7.3.2 The Tactical *REPEAT*

If T is a tactic then **REPEAT** T is a tactic which repeatedly applies T until it fails. The type of *REPEAT* is *TACTIC −> TACTIC*. This can be illustrated in conjunction with ∀_*tac*, which is specified by:

$$\frac{\forall x \bullet t[x]}{\overline{t[x']}}$$

[2]This usage was introduced by Robin Milner: 'tactical' is to 'tactic' as 'functional' is to 'function'.

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

∀_tac strips off one universal quantifier; *REPEAT ∀_tac* strips off all universal quantifiers:

```
SML                                                                 2
│   g ⌜ ∀x y z• (x + (y + z)) = ((x + y) + z ) ⌝;
│

│   (* ?⊢ *)  ⌜∀ x y z• x + y + z = (x + y) + z⌝
│

SML
│   a ∀_tac;

│
│   (* ?⊢ *)  ⌜∀ y z• x + y + z = (x + y) + z⌝
│

SML
│   a (REPEAT ∀_tac );

│   (* ?⊢ *)  ⌜x + y + z = (x + y) + z⌝
```

FURTHER TACTICS

This section describes some of the tactics built-in to the ProofPower system in addition to those described above. This section is not meant to provide complete coverage of the available tactics, but rather to acquaint the reader with more of the effects to be achieved in transforming goals,and some tactics to achieve those effects. There are many more available tactics, and variations of tactics, than are covered here. .

8.1 Simplifying the Goal

An important tactic is that which 'strips' or simplifies a goal. The tactic *strip_tac* which has already been mentioned, performs a variety of simplifications, and is often usefully applied at the outset of embarking on a proof. The simplifications achieved by *strip_tac* include the following:

- moving the antecedent of an implication from the conclusion to the assumptions of the goal:
- proving tautologies
- removing leading universal quantifiers
- using, where possible relevant, assumptions in the assumption-list

3

SML
```
g ⌜(P 3) ⇒ ∀x• x = 3 ⇒ P x⌝;
```

```
(* ?⊢ *) ⌜P 3 ⇒ (∀ x• x = 3 ⇒ P x)⌝
```

SML
```
a strip_tac;
```

```
(*  1 *) ⌜P 3⌝

(* ?⊢ *) ⌜∀ x• x = 3 ⇒ P x⌝
```

SML
```
a strip_tac;
```

```
(*  1 *) ⌜P 3⌝

(* ?⊢ *) ⌜x = 3 ⇒ P x⌝
```

SML
```
a strip_tac;
```

```
(*  2 *) ⌜P 3⌝
(*  1 *) ⌜x = 3⌝

(* ?⊢ *) ⌜P x⌝
```

SML
```
a strip_tac;
```

```
Exception−
   Fail
     * There is no stripping technique for ⌜P x⌝ in the current proof
       context [strip_tac.28003] * raised
```

SML
```
 a (asm_rewrite_tac [ ]);
```

```
Current and main goal achieved
```

The foregoing session showed 4 successive applications of *strip_tac* of which the first three each had an effect and the fourth failed, leaving a goal amenable to *asm_rewrite_tac*. With the knowledge provided by hindsight, we can see that a single compound tactic to achieve this goal would be to repeat *strip_tac* until failure, and then apply *asm_rewrite_tac*, thus:

```
SML                                                                    4
│   g ⌜(P 3) ⇒ ∀x•  x = 3 ⇒ P x⌝;


│   (* ?⊢ *)  ⌜P 3 ⇒ (∀ x• x = 3 ⇒ P x)⌝
│


SML
│   a ((REPEAT strip_tac)  THEN (asm_rewrite_tac [ ]));


│   Current and main goal achieved
```

Although this particular example is specific to the goal, nevertheless (*REPEAT strip_tac*) is often useful as an opening gambit.

The tactic *strip_tac* reduces the complexity in the conclusion of the goal, but does nothing to simplify the assumptions. In order to give *strip_tac* as much as possible to work on, it may be useful in the early stage of a proof to move complexity from the assumptions into the conclusion. A tactic, *all_asm_ante_tac*, is available to achieve this effect. In the following example, *strip_tac* is ineffective on a goal with such a simple conclusion($U = V$), but moving the assumptions into the conclusion with *all_asm_ante_tac* will make the conclusion amenable to (*REPEAT strip_tac*).

```
SML                                                                    5
│   set_goal([⌜P=Q⌝, ⌜¬ P = Q⌝], ⌜U = V⌝);


│   (*  2 *)  ⌜¬ P = Q⌝
│   (*  1 *)  ⌜P = Q⌝
│
│   (* ?⊢ *)  ⌜U = V⌝
│


SML
│   a all_asm_ante_tac;


│   (* ?⊢ *)  ⌜¬ P = Q ⇒ P = Q ⇒ U = V⌝
│


SML
│   a (REPEAT strip_tac);


│   Current and main goal achieved
```

In this example, the conclusion of the final goal is in fact a tautology, so it would be amenable to other tactics, notably **taut_tac**.

```
SML                                                          6
│   set_goal([⌜P=Q⌝, ⌜¬ P = Q⌝], ⌜U = V⌝);

│   (* 2 *) ⌜¬ P = Q⌝
│   (* 1 *) ⌜P = Q⌝
│
│   (* ?⊢ *) ⌜U = V⌝
│

SML
│   a (all_asm_ante_tac THEN taut_tac);

│   Current and main goal achieved
```

8.2 Specializing the Assumptions

Consider the following:

```
SML                                                          7
│   g ⌜(∀x•P x) ⇒ P y⌝;

│   (* ?⊢ *) ⌜(∀ x• P x) ⇒ P y⌝
│

SML
│   a (REPEAT strip_tac);

│   (* 1 *) ⌜∀ x• P x⌝
│
│   (* ?⊢ *) ⌜P y⌝
```

Here there is a universally-quantified assumption of which the conclusion is an instance. There is an applicable tactic, called **spec_nth_asm_tac** which takes two arguments:

- the assumption-number of the relevant universal assumption (in this case, 1)
- a term in which to instantiate the universal, so as to yield the conclusion. In this case the appropriate term would be ⌜y⌝.

```
SML                                                          8
│    a (spec_nth_asm_tac 1 ⌜y⌝);

│    Current and main goal achieved.
```

If specializing the universal is not sufficient to achieve the goal, the result is simply to strip the new specialized assumption into the list of assumptions. This in itself may be a useful step towards achieving the goal, as the following example is contrived to show.

```
SML                                                          9
│    g ⌜ (R = Q ∧  (P y) ∧ ∀x• P x  ⇒ Q x) ⇒ R y⌝;

│    (* ?⊢ *)  ⌜R = Q ∧ P y ∧ (∀ x• P x ⇒ Q x) ⇒ R y⌝
│

SML
│    a (REPEAT strip_tac);

│    (*  3 *)  ⌜R = Q⌝
│    (*  2 *)  ⌜P y⌝
│    (*  1 *)  ⌜∀ x• P x ⇒ Q x⌝
│
│    (* ?⊢ *)  ⌜R y⌝
│

SML
│    a (spec_nth_asm_tac 1 ⌜y⌝);

│    (*  4 *)  ⌜R = Q⌝
│    (*  3 *)  ⌜P y⌝
│    (*  2 *)  ⌜∀ x• P x ⇒ Q x⌝
│    (*  1 *)  ⌜Q y⌝
│
│    (* ?⊢ *)  ⌜R y⌝
│

SML
│    a (asm_rewrite_tac [ ]);

│    Current and main goal achieved
```

8.3 Existentially Quantified Goals

Consider the case when the conclusion of the goal is of the form $\exists x \bullet\ P\ x$. It will commonly be the case that propositions of this form are achievable by producing a witness w which has property P, so the goal becomes one of showing that $P\ w$ is true. The tactic **∃_tac** has the purpose of transforming the goal in this way, from $\exists x \bullet\ P\ x$ to $P\ w$. The following example takes the goal of proving that there is a number less than 1, and the required witness will be the number 0.

SML 10

```
g ⌜∃x•x < 1⌝;
```

```
(* ?⊢ *)  ⌜∃ x• x < 1⌝
```

SML

```
a (∃_tac ⌜0⌝);
```

```
(* ?⊢ *)  ⌜0 < 1⌝
```

This tactic has had the expected effect. By inspecting the listing of the theory ℕ we see that a relevant fact, that is, *0 < 1*, is obtainable from the theorem **less_clauses**. For the purpose of rewriting, the 'fact' *0 < 1* can be understood as the equation *0 < 1 = T*. Thus it will be sufficient to rewrite with : *less_clauses*. (Rewriting with an empty list of equations would also work, picking up =INLINEFT 0 ¡ 1 by default.)

SML 11

```
a (rewrite_tac [less_clauses]);
```

```
Current and main goal achieved.
```

8.4 Contradiction and Resolution

In this section some further tactics are introduced by showing some different approaches to the proof of $\exists x \bullet x < 1$

Firstly, we could try a proof by contradiction: if the conclusion is true then its negation should lead to a falsehood. A tactic to apply is **contr_tac**.

```
SML                                                    12
│      g ⌜∃x•x < 1⌝;

│      (∗ ?⊢ ∗)  ⌜∃ x• x < 1⌝
│

SML
│      a contr_tac;

│      (∗  1 ∗)  ⌜∀ x• ¬ x < 1⌝
│
│      (∗ ?⊢ ∗)  ⌜F⌝
```

Assumption 1 contradicts the fact that *0 < 1*, which we have seen already can be established from *less_clauses*, and this contradiction can be **resolved** to prove the goal with conclusion *F* by using a tactic called **Resolution.basic_res_tac1**. Since the resolution process may, in some circumstances continue indefinitely, the tactic takes an argument which is a number limiting the amount of processing. For this purpose a value of, say, 5, ought to be ample. The second argument of the tactic is a list of theorems to be resolved with the assumptions of the goal.

We use *less_clauses* as a suitable list.

```
SML                                                    13
│      a (Resolution.basic_res_tac1 5 [less_clauses]);

│      Current and main goal achieved
```

We saw above that *Resolution.basic_res_tac1* was appropriate with a goal of *F* and a contradiction exploitable. A variation of this tactic will in effect first apply *contr_tac*, so that the proof above can be performed in one step:

```
SML                                                    14
│      g ⌜∃x•x < 1⌝;

│      (∗ ?⊢ ∗)  ⌜∃ x• x < 1⌝
│

SML
│      a (Resolution.basic_res_tac 5 [less_clauses]);

│      Current and main goal achieved
```

Here is another example which illustrates the principle of resolution. It uses another resolution tactic called **asm_prove_tac**

SML 15
```
set_goal ([⌜P ∨ Q⌝ , ⌜R ∨ ¬ Q⌝] , ⌜P ∨ R⌝);
```

```
(* 2 *) ⌜R ∨ ¬ Q⌝
(* 1 *) ⌜P ∨ Q⌝

(* ?⊢ *) ⌜P ∨ R⌝
```

SML
```
a (asm_prove_tac [ ]);
```

```
Current and main goal achieved
```

8.5 Proof Contexts

It has been mentioned that rewriting automatically uses equations in the supplied theories as well as those equations supplied explicitly by the user. The choice of which equations are automatically used is in fact not fixed, but is an aspect of what is called the current **proof context**. Other features of the system are also influenced by the proof context, notably the stripping tactics, automatic existence proving in constant-specification and the behaviour of tactics such as *asm_prove_tac*.

All the examples of this tutorial have been presented in the proof-context which is provided by default.

In this default proof-context, context-sensitive features of the system have to some degree been optimized around the issued theories. There are facilities for users developing new theories to define proof-contexts specially tailored to those new theories. These facilities are covered in ProofPower *Reference Manual* [12] but a further description is beyond the scope of this tutorial.

SPECIFICATION WITHOUT AXIOMS

The section covers the topic of developing theories without the introduction of axioms. Firstly, note that the function *simple_new_defn* has already been mentioned (5.4.5 above) as providing one means of defining constants without axioms. However, the effects achievable by this function are limited to assigning names to terms, that is, to definitions of the form *name* = *value*. Means are now considered of specifying constants with predicates which are arbitrary, so long as consistency is maintained.

9.1 Specifying Constants

This section covers specifying new constants of existing types. The next section will cover specifying new types and constants of new types.

The following example shows specification of a function with a predicate consisting of two equations. (Recall that the means of entering specifications in this way was described in section 5.5 above.)

```
HOL Constant
│Factorial:ℕ→ℕ
├──────────────────────────────
│Factorial 0 = 1 ∧
│∀ x:ℕ • Factorial (x+1) = (x+1) * Factorial x
```

Executing *print_theory* "−"; at this point will show a new definition theorem for Factorial. This theorem can be retrieved by executing

- **get_spec** ⌜*Factorial*⌝
- **get_defn** "−" "*Factorial*"

Clearly, since there are two equations, it is conceivable that there is no function which satisfies them both. In the course of entering the definition of *Factorial*, the system was able to automatically prove a theorem to the effect that the definition of *Factorial* is consistent, that is, there exists a function with the same definition as *Factorial*. The automatic proving facilities are oriented towards defining functions with multiple equations, such as this, and may not necessarily be able to prove existence automatically for an arbitrary predicate. Here is an example session: a constant N is specified, very loosely, as any non-zero number:

```
HOL Constant                                                   16
│N:ℕ
├──────────────────────────────
│N > 0
```

and the system response is:

```
val it = ⊢ ConstSpec (λ N′• N′ > 0) N : THM
```

Observe that the form of the resulting theorem is different from that of the previous example: the presence of *ConstSpec* is a signal that more remains to be done. Examining the specification of N we see that it is qualified with an assumption about the consistency of the predicate:

```
SML                                                            17
    get_spec ⌜N⌝;

    val it = Consistent (λ N′• N′ > 0) ⊢ N > 0 : THM
```

Sooner or later this consistency-assumption should be discharged. This is achieved with the functions **push_consistency_goal** and **save_consistency_thm** as follows:

```
SML                                                            18
    push_consistency_goal ⌜N⌝;

    ...

    (* ?⊢ *)  ⌜∃ N′• N′ > 0⌝

SML
    a (∃_tac ⌜1⌝);
    a (rewrite_tac[ ]);

    Current and main goal achieved

SML
    save_consistency_thm ⌜N⌝ (pop_thm ());
```

Now the specification of N can be re-examined to see the change achieved by performing the proof: the consistency assumption has been discharged:

```
SML                                                            19
    get_spec ⌜N⌝;

    val it = ⊢ N > 0 : THM
```

9.2 Specifying Types

This section covers the specification of new types. A new type, as considered here, is defined in terms of a subset of an existing type with membership characterised by a predicate. A simple example is the ordinal numbers: the subset of the natural numbers which are non-zero.

The first step is to prove a theorem that such a subset is non-empty. The theorem must have the form $\exists x\bullet\ P\ x$ so the goal is taken as $\exists x{:}\mathbb{N} \bullet (\lambda x\bullet \neg\ x = 0)\ x$ rather than $\exists x{:}\mathbb{N} \bullet \neg\ x = 0$.

```
SML                                                              20
│      new_theory "ordinals";
│      set_goal ([ ], ⌜∃x:ℕ• (λx• ¬ x = 0) x⌝);
│      a  ((∃_tac ⌜1⌝) THEN (rewrite_tac [ ])) ;

│      Current and main goal achieved
```

The new type is now introduced. The function **new_type_defn** takes three arguments:

- one or more names (keys) under which a defining-theorem will be stored.
- a name for the type itself. In this example the name is 'Ordinal'.
- The existence theorem just proved, which is currently available on the top of goal-stack and so can be retrieved by *top_thm* or *pop_thm*.

```
SML                                                              21
│      new_type_defn (["ordinal_def"], "Ordinal", [ ], pop_thm());

│      val it = ⊢ ∃ f• TypeDefn (λ x• ¬ x = 0) f : THM
```

We have a new type, 'Ordinal', and can construct terms with variables of that type, but so far we have only very limited means of specifying constants.

```
HOL Constant
│VII:Ordinal
├──────────────────────────────
│T
```

Even though Ordinals were specified by a predicate which characterised a subset of the numbers, we cannot simply equate an Ordinal variable with a number because the equality will be ill-typed – left and right hand sides will be of different types.

```
HOL Const
│VIII:Ordinal
├──────────────────────────────
│VIII = 8
```

The system response is:

```
        Type error in ⌜VIII = 8⌝
        The operands of ⌜$=⌝ must have the same type
        The types inferred were:
        ⌜VIII:Ordinal⌝
        ⌜8:ℕ⌝
        Exception− Fail ∗ Type error [HOL−Parser.16000] ∗ raised
```

Thus it is not the case that the new type is a subset of the parent type, but rather that there is an isomorphism between the new type and the subset of the parent type. If we wish to specify the values of constants of the new type with predicates which include terms of existing types, we will need functions for, in this case, :

- mapping numbers to ordinals. This mapping is called 'abstraction' and a suitable function A will be developed in the example. With this function terms can be written such as *VII = A 7*.
- mapping ordinals to numbers. This mapping is called 'representation' and a suitable function R will be developed in the example. With this function terms can be written such as *7 = R VII*.

Defining the mappings A and R is the next task. A theorem must be proved which asserts that the intended isomorphism can exist, that is, that there are two mappings with suitable properties. Here we take a 'cookbook' approach, so that the following 'recipe' will serve, with slight adaptations, in most cases. The goal is in a 'standard ' form; note the occurrence in the goal of the characteristic predicate as *¬ n = 0*.

The reader is asked to accept without explanation that the four tactics shown below are generally effective for proving a goal such as this.

```
SML                                                                    22
        set_goal([ ],⌜∃ A R •
              (∀ a : Ordinal • A ( R a) = a)
              ∧
              (∀ n : ℕ • (¬ n = 0) ⇔ (R (A n) = n))⌝);

        a (strip_asm_tac (rewrite_rule[ ]
                (simple_⇒_match_mp_rule type_lemmas_thm
                (get_defn "−" "ordinal_def"))));
        a (∃_tac ⌜abs⌝);
        a (∃_tac ⌜rep⌝);
        a (asm_rewrite_tac[ ]);


        ...
        Current and main goal achieved
```

We now specify the constructor functions A and R as constants, by use of the function **new_spec**. This takes three arguments:

- A list of keys for the defining theorem which will be produced.
- a count of the number of constants – 2 in this case, (A and R)

- An existence theorem for A and R, which is the one just proved and available on the top of the goal-stack.

SML 23

```
    val ordinal_consts_def =
        new_spec(["R","A","ordinal_consts_def"],2,pop_thm());


    val ordinal_consts_def =
      ⊢ (∀ a• A (R a) = a) ∧ ∀ n• ¬ n = 0 ⇔ R (A n) = n) : THM
```

We are finally in a position to specify constants of the new type. Note that more than one constant can be specifiedat a time.

HOL Constant

```
IX X :Ordinal
─────────────────────────────
IX = A 9 ∧ X = A 10
```

The system response is:

```
    val it = ⊢ IX = A 9 ∧ X = A 10 : THM
```

What has been achieved is summarised in the current theory:

```
SML                                                                    24
print_theory "-";

        === The theory ordinals ===

        --- Parents ---

                demo

        --- Constants ---

        VII             Ordinal
        R               Ordinal → ℕ
        A               ℕ → Ordinal
        X               Ordinal
        IX              Ordinal

        --- Types ---

        Ordinal

        --- Definitions ---

        ordinal_def     ⊢ ∃ f• TypeDefn (λ x• ¬ x = 0) f
        VII             ⊢ T
        R
        A
        ordinal_consts_def
                        ⊢ (∀ a• A (R a) = a)
                            ∧ (∀ n
                            • ¬ n = 0 ⇔ R (A n) = n)
        IX
        X               ⊢ IX = A 9 ∧ X = A 10

        === End of listing of theory ordinals ===
```

MOVING ON

We hope that this tutorial provides a helpful introduction to ProofPower. Depending on your interests, we would recommend ProofPower *HOL Tutorial Notes* [10] or ProofPower *Z Tutorial* [9] as a next step.

A comprehensive reference manual to the facilities provided by ProofPower is supplied as the ProofPower *Reference Manual* [12]. Many of these facilities are intended for use by a programmer extending the system rather than by a user developing specifications or proofs. However, despite its length, many users find it useful to have the ProofPower *Reference Manual* [12] on the screen for interactive reference via its keyword-in-context index.

In response to popular demand, we conclude this document with a list of the names of some of the more commonly used tactics, rules, conversions, tacticals and conversionals. These have been extracted from some of the proof scripts provided with the system.

10.1 Tactics

∨_left_tac
∨_right_tac
⇒_tac
∀_tac
∃_tac
accept_tac
all_asm_ante_tac
ante_tac
asm_ante_tac
asm_fc_tac
asm_prove_tac
asm_rewrite_tac
asm_rewrite_thm_tac
asm_tac
bc_tac
bc_thm_tac
cases_tac
contr_tac
conv_tac
fc_tac
gen_induction_tac
id_tac
induction_tac
intro_∀_tac
lemma_tac
list_induction_tac
list_spec_asm_tac
list_spec_nth_asm_tac
once_rewrite_tac
once_rewrite_thm_tac
prove_∃_tac
prove_tac
pure_asm_rewrite_tac
pure_rewrite_tac
rename_tac
rewrite_tac
rewrite_thm_tac
spec_nth_asm_tac
step_strip_tac
strip_asm_tac
strip_tac
swap_asm_concl_tac
swap_nth_asm_concl_tac

10.2 Rules

⇒_elim
∀_elim
all_∀_elim
all_∀_intro
asm_rule
conv_rule
eq_sym_rule
list_∧_intro
list_∀_elim
list_∀_intro
pc_rule
prove_rule
rewrite_rule
strip_∧_rule
taut_rule

10.3 Conversions

eq_sym_conv
prove_∃_conv
rewrite_conv

10.4 Conversionals

ONCE_MAP_C
ORELSE_C
THEN_C
TOP_MAP_C

10.5 Tacticals

⇒_T
CASES_T
DROP_ASM_T
DROP_NTH_ASM_T
FC_T
GET_ASM_T
GET_NTH_ASM_T
INDUCTION_T
LEMMA_T
LIST_DROP_NTH_ASM_T
LIST_GET_NTH_ASM_T
LIST_SPEC_NTH_ASM_T
ORELSE
PC_T
POP_ASM_T
THEN
TOP_ASM_T
TRY_T

REFERENCES

[1] Michael J.C. Gordon and Tom F. Melham, editors. *Introduction to HOL.* Cambridge University Press, 1993.

[2] Michael J.C. Gordon, Arthur J. Milner, and Christopher P. Wadsworth. *Edinburgh LCF. Lecture Notes in Computer Science. Vol. 78.* Springer-Verlag, 1979.

[3] L.Paulson. *ML for the Working Programmer.* Cambridge University Press, 1991.

[4] R.Milner. A Theory of Type Polymorphism in Programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[5] A. Wikstrom. *Functional Programming Using Standard ML.* Prentice-Hall, 1987.

[6] DS/FMU/IED/USR001. *ProofPower Document Preparation.* Lemma 1 Ltd.

[7] DS/FMU/IED/USR005. *ProofPower Description Manual.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[8] DS/FMU/IED/USR007. *ProofPower Installation and Operation.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[9] DS/FMU/IED/USR011. *ProofPower Z Tutorial.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[10] DS/FMU/IED/USR013. *ProofPower HOL Tutorial Notes.* Lemma 1 Ltd., `http://www.lemma-one.com`.

[11] *Functional Programming in Standard ML.* R.Harper et al., LFCS, University of Edinburgh, 1988.

[12] LEMMA1/HOL/USR029. *ProofPower HOL Reference Manual.* Lemma 1 Ltd., `rda@lemma-one.com`.

[13] *The HOL System: Tutorial.* SRI International, 4 December 1989.

INDEX