ProofPower

HOL Tutorial Notes

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 on the use of ProofPower for specification and proof in Higher Order Logic (*HOL*).

The objectives of this tutorial are:

- to describe the basic principles and concepts underlying ProofPower
- to enable the student to write simple specifications and undertake elementary proofs in *HOL* using ProofPower
- to enable the student to make effective use of the reference documentation

0.2 Readership

This document is intended to be among the first to be read by new users of ProofPower, and is designed either for use with the ProofPower *HOL* course, or for independent self tuition.

0.3 Related Publications

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

1. ProofPower Tutorial *[14]*;
2. ProofPower Z Tutorial *[17]*;
3. ProofPower Description Manual *[15]*;
4. ProofPower Reference Manual *[20]*;
5. ProofPower Installation and Operation *[16]*;
6. ProofPower Document Preparation *[13]*.

0.4 Area Covered

This document consists of notes appropriate for the introductory ProofPower-HOL course, which gives an idea of the way ProofPower is used for checking specifications and conducting proofs in ProofPower-HOL.

After working through this tutorial, the reader should be capable of using ProofPower with ProofPower-HOL for simple tasks, and should be able to make effective use of the ProofPower documentation where necessary for approaching more difficult problems.

The tutorial should enable users of ProofPower to become familiar with the following subjects:

1. The dialect of *HOL* supported by the ProofPower system (which we call ProofPower-HOL) and its manipulation via the metalanguage.
2. Forward proof and derived rules of inference.
3. Goal directed proof, tactics and tacticals.

0.5 Prerequisites

Prior acquaintance with first order predicate logic and a functional programming language would be an advantage.

Some familiarity with:

- first order predicate calculus

  $\ulcorner(\forall x\bullet\ P\ x \Rightarrow R\ x) \Rightarrow ((\forall\ x\bullet\ P\ x) \Rightarrow (\forall x\bullet\ R\ x))\urcorner$;

- elementary set theory

  $\ulcorner\forall a\ b\ c\bullet\ a \cap (b \cap c) = (a \cap b) \cap c\urcorner$;

- functional programming

  ```
  SML
  fun     fact 0 = 1
  |       fact n = n * (fact (n - 1));
  ```

A suitable text for an introduction to the predicate calculus and elementary set theory is *Software Engineering Mathematics* [12]. A good account of Standard ML may be found in *ML for the Working Programmer* [8].

A gentler introduction to ProofPower may be found in ProofPower *Tutorial* [14], which though not strictly pre-requisite can beneficially be read before these tutorial notes. In particular, chapter 1 of ProofPower *Tutorial* [14], which describes basic interaction with ProofPower, is recommended, since this topic is not touched upon in any depth here.

0.6 How To Use This Tutorial

It is intended that this document will allow ProofPower users who have not attended the ProofPower-HOL course to work through the course material independently. In that case the material could be read in conjunction with the course OHP transparencies [18].

The best way to learn about ProofPower is by doing things with it.

The two kinds of things which you can do while working through these tutorial notes are:

- Do the set exercises.

  To make it easier to do the exercises the installation procedure for ProofPower results in the establishment of a ProofPower database called '`example_hol`', which contains the results of executing all of this tutorial document except the material in Chapter 14 where the solutions to the exercises may be found. To do the exercises the reader should attempt to set up his own version of the solutions document ('`usr013S.doc`') by working interactively in a ProofPower session using a copy of database `example_hol`.

  This is best done using a writeable copy of the database so that you can save the database after completing some of the exercises and then resume from that point later. This can be done as follows:

  ```
  cp $PPHOME/db/example_hol.polydb .
  chmod +w example_hol.polydb
  ```

  Here, $*PPHOME* is an environment variable which should be set up to be the pathname of the directory in which ProofPower has been installed.

  If you wish to use the X interface for ProofPower, `xpp`, you can now start your ProofPower session by starting X if necessary and then giving the UNIX command:

  ```
  xpp -d example_hol
  ```

  `xpp` will come up running ProofPower on your copy of the database and with its editor set up to work on a new, empty, script in which you can build up your solutions.

- Replay the illustrative material.

  This is best done using the source of the tutorial OHP transparencies, `usr022_slides.doc`. It can be done running on database `example_hol`, though you will find that some of the material will be rejected because definitions have already been made. Alternatively you can work from a clean database, but then you may find problems if you miss out any of the material. E.g., to work on the existing database using `xpp`, you might use the command:

  ```
  xpp -f $PPHOME/doc/usr022_slides.doc -d example_hol
  ```

  The illustrative material can be replayed in a batch mode, but this is not very instructive.

In both of these use of the source documents avoids unnecessary re-keying of material, and should be loaded into a text editor and used by copy-and-paste.

Two alternative approaches to working through the exercise material are:

- Follow the transparencies.

  Consulting the tutorial notes as necessary for further information, while using copy-and-paste on the OHP source file (`usr022_slides.doc`) to replay the illustrations and do the exercises.

- Follow the tutorial notes.

  Work through the tutorial notes (using a `.dvi` previewer or hard copy) and the exercises as presented in this document. Source documents are supplied for the exercises (Chapter 13, file `usr013X.doc`) and solutions (Chapter 14, file `usr013S.doc`).

In either case it is best to build up your own document containing your solutions to the exercises and any experiments you might wish to undertake. ProofPower does not keep any record of what you type into it, and so if you want to do it again you will need to keep a copy of your script.

0.7 Acknowledgements

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

We are particularly indebted to Mike Gordon of The University of Cambridge, for his leading role in some of the research on which the development of ProofPower has built, and for his positive attitude towards industrial exploitation of his work.

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 [4], 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 [5]. The functional language 'Standard ML' used both for the implementation and as an interactive metalanguage for proof development, originates in work at Edinburgh, and has been developed to its present state by an international group of academic and industrial researchers. The implementation of Standard ML on which ProofPower is based was itself originally implemented by David Matthews at the University of Cambridge, and is now commercially marketed by Abstract Hardware Limited.

The ProofPower system also supports specification and proof in the Z language, developed at the University of Oxford. We are therefore also indebted to the research at Oxford (and elsewhere) which has contributed to the development of the Z language.

INTRODUCTION

1.1 What is ProofPower?

ProofPower is a suite of tools providing support for the use of formal mathematical notations in the development of Information Systems.

The functionality supported is:

- document preparation (see ProofPower *Document Preparation* [13])
- syntax and type checking of specifications
- construction and checking of formal proofs
- theory management

These are described in greater detail below.

1.2 Attributes of ProofPower

ProofPower has been designed and implemented with the following attributes in mind:

- Pedigree
- Power
- Assurance
- Openness
- Extensibility

1.2.1 Pedigree

ProofPower is in the tradition of *Principia Mathematica* [1]. It is based on Church's Simple Theory of Types [2], augmented by Milner style polymorphism [9]. Its implementation builds on research at Universities of Edinburgh[5], Cambridge [3][4] and Oxford [11]. It follows the *LCF paradigm* [3], using standard ML as a 'meta-language' [19].

1.2.1.1 Logic

Following *Principia Mathematica*, ProofPower supports as its primary object language (in which specifications are written and proofs conducted) a logical type theory with a small number of primitive constructs. In the context of this logical system the main body of classical mathematics may be developed without further logical extensions other than conservative extensions which serve to define the concepts used to express the mathematics. The main merit of this approach is that once the primitive logical system has been de-bugged (i.e., shown to be consistent) the further development of mathematical theories can be undertaken without risk of compromising the consistency of the logic.

The logical system used in *Principia Mathematica* was Russell's *Theory of Types* [10], the first of many logical type theories to be developed. Probably the simplest type theory adequate for classical mathematics is the *Simple Theory of Types* published by Alonzo Church in 1940. Church's formulation formed the basis for the logical system implemented in the proof tool for Higher Order Logic (*HOL*) developed by Mike Gordon and others at the University of Cambridge [4]. This same formulation of *HOL* was adopted without material changes for use in ProofPower.

The transition from the simple Theory of Types to the logical system of Cambridge *HOL*, apart from some minor adjustments to the primitive constants and axioms, consists in adding those features which are desirable for the practical usability of the logic for the development of mathematics or for applications in Computer Science and Information Systems Engineering. The most significant of these are the introduction of *type variables* into the object language, giving an essential element of polymorphism, and the prescription of acceptable means of conservative extension, enabling new terminology to be safely introduced.

The polymorphism adopted is due to Robin Milner [9], and was originally used in the LCF system developed at the University of Edinburgh [5].

1.2.1.2 Implementation

The implementation method adopted in the LCF system (which we often refer to as the *LCF paradigm* was also used in ProofPower and is the source of many of the product characteristics. This involves the use of a strongly typed functional programming language as a meta-language both for the implementation of the proof tool and as the language in which the user interacts with the system. The meta-language used in ProofPower is 'standard ML' a more modern development than the ML available at the time the Cambridge *HOL* system was implemented.

1.2.2 Power

ProofPower supports expressive and convenient notations in strong logical systems providing a productive environment for specification and proof development.

ProofPower-HOL is:

- Logically as expressive as Zermelo set theory.
- Logically extendible in safe and well understood ways.
- Notationally concise.
- A very simple language.

ProofPower-HOL has a very high power to weight ratio, giving high levels of expressiveness and logical strength for very low levels of complexity.

ProofPower also supports the specification language Z, which provides more concise notations at the cost of some increase in complexity.

The functional programming language 'standard ML' is used as a meta-language in ProofPower, for user interaction, as the language for describing proofs, and as a productive vehicle for programming proof automation and for extending and adapting the capabilities of ProofPower to particular application domains.

These notations combine to give high levels of productivity in specification and proof, which are continually improved as the system is developed.

1.2.3 Assurance

ProofPower provides unparallelled levels of assurance in the correctness of propositions proven using ProofPower about specifications written in ProofPower-HOL or ProofPower-Z.

This assurance derives from:

- The simple uncontroversial classical logical system.

  Which reduces to a minimum the risk that the logic is inconsistent or unsound or its implementation flawed.

- The availability of mathematical and formal specifications of the syntax and semantics of the formal system.

  Which further increases confidence in the soundness of the logic and the correctness of its implementation.

- The provision of good support for specification by conservative extension.

  Which ensures that any errors arising while writing specifications do not compromise the consistency of the logical framework within which reasoning about these specification is conducted.

- The small logical kernel (<10% of the code in the system), implemented as an abstract datatype, which enforces the logical soundness of proofs.

  Minimisation of code in the system critical to the checking of proofs provides maximal confidence in correctness of the checking and enables further development of high level proof capabilities to be done without any risk of compromising the checking of proofs.

- The formal specification of the logical kernel.

  Which ensures not only that the logic itself is well understood but also that the mechanisms which enforce the checking of proofs against that logic have been thoroughly scrutinised.

1.2.4 Openness

ProofPower is 'open' in several different senses of that term:

- ProofPower provides support for standard well documented languages (Standard ML, *HOL*, Z, SAL/SPARK). Where standards are in place or under development ProofPower is intended to implement or intercept these standards.

- Most of the functions used to implement ProofPower are available for re-use by the user if he wishes to extend or customise the system.
- A comprehensive reference manual (600 + pages) is supplied documenting the functions supported by ProofPower (>1000 ML names).
- Extensive libraries of theories and 'proof contexts' are provided for re-use.

1.2.5 Extensibility

The open architecture of the *LCF paradigm* maximises the extent to which users can extend and customise the system to meet their special requirements.

- Users have access to the meta-language (Standard ML) for:
  - developing proofs
  - extending the system
  - implementing domain specific (or general purpose) proof automation
- The definitional forms acceptable to the system are extendible.

  Specifications are acceptable in any form provided that they can be shown to be consistent. Automatic consistency provers invoked by the system can be replaced or supplemented by the user extending the forms of specification which are accepted by the system as consistent.
- Many aspects of the behaviour of the automatic proof capabilities are context sensitive, enabling their effects to be continually augmented with knowledge of new problem domains. Customisable 'proof contexts' provide this information to the proof system and enable the proof developer to select a suitable context for conducting any particular proof, maximising the extent to which the proof support system eliminates or simplifies goals or subgoals automatically.
- The system is designed to support multiple object languages and permits mixed language working.
- Parser generators are available to simplify the task of providing support for additional notations.

1.3 Languages Supported

ProofPower is designed to support multiple object languages in a single coherent semantic framework. The approach of implementing secondary object languages by semantic embedding into the primary object language ProofPower-HOL not only provides for multiple notations, but enables specifications in distinct notations to be related to each other formally, and permits results to be transferred from one language to another.

The following languages are currently supported:

- Standard ML (as meta-language)
- Higher Order Logic
- Z

Support for SAL, the SPARK Annotation Language, has been prototyped, but is not yet generally available.

We hope to be able to provide support for ISO Standard Z when the standard has stabilised.

1.4 Functionality

ProofPower is a suite of programs and scripts intended to assist in the development and checking of formal specifications and proofs.

The functionality supported is:

- document preparation/printing (see ProofPower *Document Preparation* [13]):
  - using LaTeX 'literate scripts' with extended fonts for document sources
  - indexes, cross reference and theory listings
- syntax check/type check (interactive or batch)
- formal reasoning (interactive or batch)
- theory management:
  - specifications and theorems held in theory hierarchy
  - programmable access to theory hierarchy

1.4.1 Document Preparation

ProofPower provides facilities for producing documents containing specifications and proof scripts exploiting the LaTeX typesetting software. Following Knuth [6], these are known as 'literate scripts'.

The facilities include:

- Screen fonts for use with Sunview or X which enable source documents including formal specification material to be written using the appropriate special characters rather than ASCII encodings.
- Under X, a custom-built editor for developing scripts using the special characters.
- Tools for preprocessing source documents prior to their being processed by the ProofPower proof tool or LaTeX.
- Output from ProofPower suitable for inclusion in LaTeX documents.
- The production of indexes of defining occurrences of formal names in LaTeX documents.

1.4.2 Syntax and Type Checking

ProofPower processes a variety of formal notations including standard ML, Higher Order Logic, and Z. These are typically mixed together in a mixed language dialogue with the system. The system operates either interactively with a user at a console, usually making heavily use of cut and paste between ProofPower and external text editors, or in a batch mode of operation where input is read from a nominated file and output written to the standard output and/or other files.

Each of the notations used in interacting with ProofPower is *strongly typed.* This makes it possible to provide valuable diagnostic information at the time of processing of each definition or specification.

1.4.3 Proof Development

Following the *LCF paradigm* proofs are created and checked in ProofPower as computations which result in values of type *THM*. *THM* is an abstract type in the meta-language standard ML, and as such the means of computing values of this type are strictly limited to those provided at the time the type was defined. The constructors of the type *THM* have been engineered to correspond to inference rules in the supported logic, HOL, so that any value of type THM computed must follow by the rules of the logic from the declared axioms. Proof automation is provided by the implementation of high level proof development facilities which ultimately compute theorems only through the inference mechanisms built into the definition of the data type THM.

Though proofs are always ultimately comprised of elementary proof steps, the availability of a powerful modern functional programming language for generating such proofs permits continually growing sophistication in the automatic proof facilities provided. New subject domains can be incorporated in the scope of these automatic facilities, and users can customise and extend these capabilities to meet the special requirements arising in their applications.

1.4.4 Theory Management

In order that the system is able to give reliable indications of what theorems are derived from what premises, or in the context of which specifications, the system must manage in a secure way the axioms, definitions and theorems which are used in the development of specifications and proofs. These are organised in a theory hierarchy, permitting reusable theories to be exploited in different applications. An initial theory hierarchy is supplied with the system containing some commonly used theories, such as natural numbers, and lists.

1.5 Levels of Use of ProofPower

ProofPower may be used by several different types of user for different purposes. These different groups may require distinct levels of knowledge of ProofPower for their particular purposes.

1.5.1 Education

ProofPower has potential as an educational tool, even in areas not directly related to ProofPower or to the languages supported by ProofPower. For example, after development of suitable course material ProofPower might be used to support courses in logic and set theory. Course material of this nature is not yet available with the ProofPower system, but we hope that eventually it will be.

If training material was developed for teaching such topics using ProofPower, the level of knowledge of ProofPower required for using the material could be made very small.

For those learning the specification languages *HOL* or Z, the proof facilities provided by ProofPower are beneficial in developing an understanding of the semantics of these notations.

1.5.2 Specification

ProofPower is a useful and effective tool for those who wish to develop formal specifications of information systems, even if they have no requirement for conducting proofs about these specifications. Such users could use the document preparation facilities for writing their specifications, having these syntax and type checked interactively and incrementally by ProofPower as the specification is developed.

Those wishing to use ProofPower for developing and checking specifications in this way need have little more than an understanding of the specification language which they propose to use.

1.5.3 Proof Development

The group of users on whom ProofPower is most deliberately targetted are those who wish to develop formal specifications and then reason about or in the context of these specifications.

1.5.4 Research and Development

Because of its openness and extensibility ProofPower provides a good vehicle for many kinds of research, concerning the theory and practice of Formal Methods, or aspects of AI such as automated deduction.

It is also suitable as a platform for the development of tools with capabilities which include formal reasoning, or for interactive course support material for academic courses involving formal specification or discrete mathematics. Cutomised window based interfaces can be implemented using the Motif interfaces built into ProofPower.

It is this kind of use of ProofPower which is likely to require the most detailed and comprehensive knowledge of the facilities provided by ProofPower.

1.6 Using ProofPower

The usual way of using ProofPower to develop specifications and proofs involves two parallel interactive tasks:

- Using an editor to develop a literate script in which specifications and proofs are recorded.
- Executing ProofPower-ML commands, typically extracted from the script. This is the means by which specifications are checked and by which proof steps are taken.

Under the X Windows system, an integrated program xpp is supplied to support both of these tasks. For an introduction to the basic use of ProofPower the reader is referred to chapter 1 of ProofPower *Tutorial* [14].

1.7 Introduction to Proof

The proofs of many results are either automatic or straightforward using ProofPower.

1.7.1 Areas of Automation

The following are examples of areas where proof automation in ProofPower is particularly effective:

- propositional tautologies

  ProofPower proves these automatically, and uses propositional reasoning to simplify non-propositional goals automatically.

- first order predicate calculus

  Often these will also be automatically provable using a form of resolution. Where resolution fails, there is a simple systematic approach to proving these results using ProofPower.

- elementary set theory

  A useful collection of results from elementary set theory are automatically provable.

- other classes of results

  Whenever a new theory is introduced one or more proof contexts may be developed to solve automatically a range of results in that theory. 'Decision procedures' for such classes of results can be made available via 'prove_tac'.

1.7.2 A Simple Predicate Calculus Proof

The following example illustrated the style of a simple proof in ProofPower.

Most proofs are conducted using the subgoaling package (see Chapter 6). The following example shows how proofs of results in the first order predicate calculus can be conducted systematically using only two *TACTIC*s.

To conduct such a proof the first thing to do is set the goal to be proven:

SML
```
set_goal([],⌜(∀x y• P x ⇒ R y) ⇔ (∀v w• ¬ P w ∨ R v)⌝);
```

The 'two tactic method' uses proof by contradiction, which is initiated using *contr_tac.* Such a proof is reduced to derivation of ⌜F⌝ from the negation of the required result (the conclusion of the original goal). The negated assumption is pre-processed automatically by *contr_tac* and this results in two subgoals, the first of which is shown below.

SML
```
a contr_tac;
```

ProofPower output

```
Tactic produced 2 subgoals:
...
(* *** Goal "1" *** *)

(*  3 *)  ⌜∀ x y• P x ⇒ R y⌝
(*  2 *)  ⌜P w⌝
(*  1 *)  ⌜¬ R v⌝

(* ?⊢ *)  ⌜F⌝
```

The proof now proceeds by specialising the universals in the assumptions until the system is able to derive the required contradiction. In this case specialisation of assumption 3 with the values ⌜w⌝ and ⌜v⌝ will enable the contradiction to be derived:

SML

```
a (list_spec_asm_tac ⌜∀ x y• P x ⇒ R y⌝ [⌜w⌝,⌜v⌝]);
```

ProofPower output

```
Tactic produced 0 subgoals:
```

The first subgoal is discharged automatically once the necessary specialisation has been identified, and the subgoal package then present the second subgoal:

ProofPower output

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

(*  3 *)  ⌜∀ v w• ¬ P w ∨ R v⌝
(*  2 *)  ⌜P x⌝
(*  1 *)  ⌜¬ R y⌝

(* ?⊢ *)  ⌜F⌝
```

Specialisation of assumption 3 is again the way forward, in this case to the values ⌜y⌝ and ⌜x⌝:

SML

```
a (list_spec_asm_tac ⌜∀ v w• ¬ P w ∨ R v⌝ [⌜y⌝,⌜x⌝]);
```

ProofPower output

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

This enables discharge of the second subgoal and completes the proof. The theorem can then be obtained as an ML value:

SML

```
pop_thm();
```

ProofPower output

```
Now 0 goals on the main goal stack
val it = ⊢ (∀ x y• P x ⇒ R y) ⇔ (∀ v w• ¬ P w ∨ R v) : THM
```

The reader should be able to conduct proofs of many elementary results in ProofPower using these facilities and a collection of exercises which should now be achievable are given in section 13.1.

1.8 Notational Conventions

Formal text is included throughout this tutorial, mainly giving examples of input and output from ProofPower.

Formal material is almost always distinguished from informal text by the presence of either an enclosing box or a vertical bar on the left, usually with some kind of indicator at the top of what kind of formal material it is.

The most frequent formal inserts are in Standard ML and these are marked by a vertical bar headed by the acronym SML. These inserts represent samples of input to ProofPower, and are often followed by the resulting output from ProofPower.

For example, if the following is entered into a ProofPower session:

```
SML
│3+4+5;
```

the following output results:

```
ProofPower output
│val it = 12 : int
```

Within Standard ML in ProofPower it is possible to include quotations in other languages, of which the most important in this tutorial is *HOL*. This is done by enclosing the quotation in 'Quine corners'.

When quoted in this way an expression in Higher Order Logic evaluates to a value in Standard ML of type *TERM*:

```
SML
│⌜3+4+5⌝;
```

and when a value of type *TERM* is displayed the *HOL* pretty-printer is normally invoked automatically:

```
ProofPower output
│val it = ⌜3 + 4 + 5⌝ : TERM
```

Normally in text of the tutorial, where it is necessary to quote an expression in *HOL*, these same quotation marks will be used.

Many of the concepts which it is necessary to discuss in explaining Higher Order Logic as implemented in ProofPower correspond to a type or a value in standard ML as used in the implementation of ProofPower. Terms and types in Higher Order Logic are represented in standard ML using values whose ML types are *TERM* and *TYPE* respectively.

In interests of precision in the informal text, wherever an informal concept correponds precisely to a formally defined concept, the name of the formally defined concept may be used in the informal text as well as in formal texts. For example, *HOL* types are represented in ML by values of ML type *TYPE* and therefore whenever they are referred to in the informal text they are referred to by the name '*TYPE*'. In some places other kinds of type are referred to, e.g. ML types, and in these places the notation '*TYPE*' is not used.

For each technical term a 'defining occurrence' is displayed in bold type, and its page number appears in the index at the end of the document. Other occurrences appear in italics when it is intended to emphasise that the term has been defined elsewhere. It is intended that the appearance of a word or phrase in italics should indicate that this word or phrase has been defined elsewhere, and the location of this description may be found in the index.

1.9 Using the ProofPower Reference Manual

This tutorial is intended to provide a coverage similar to that of the short ProofPower-HOL course, and is not an exhaustive account of ProofPower. Many of the facilities are mentioned with very brief or no description.

To become a proficient user of ProofPower it is necessary to become familiar with use of the ProofPower *Reference Manual* [20]. All of the ML names mentioned in this tutorial are documented in the ProofPower *Reference Manual* [20], and the documentation may be found by reference to the one of the indexes to be found in that manual.

The **KWIC** index in the ProofPower *Reference Manual* [20] is also invaluable in identifying the full range of facilities available. Each ML name is composed of a number of atoms separated by underscore symbols, and the *KWIC* index groups together all the names containing any particular atom irrespective of where in the name the atom appears.

Knowledge of a small number of naming conventions enables the user to identify all the relevant facilities of a particular kind. For example, all tactics have names ending in the atom 'tac', and may therefore be found grouped together in the *KWIC* index under 'tac'. All the rewriting facilities have the atom 'rewrite' in their name, and will be found grouped together in the *KWIC* index.

Once the reader has grasped the main principles of ProofPower and ProofPower-HOL, it is a good idea to have a browse through the *KWIC* index to get an idea of the range of facilities available. Then when he is confronted with a problem in a proof he will know whether there is something already available which may help to solve the problem, and can then look up the documentation in detail to discover how the relevant facilities are used.

THE *HOL TYPE* SYSTEM

2.1 Introduction

The ProofPower-HOL language provides a notation for making assertions about values in some domain of discourse. The values in this domain of discourse are abstract entities suitable for use in mathematical models.

In assertions in *HOL*, *TERM*s are used to denote values in the domain of discourse. Each *TERM* used must be syntactically well formed, and must also be **well typed**; ProofPower will check this automatically and report any problems to the user. A *TERM* is *well typed* if there exists an assignment of *TYPE*s to the *TERM* and its sub-*TERM*s which is consistent with the *HOL TYPE* inference rules described below.

Each *TYPE* denotes a set of values. The semantics of *TYPE*s and *TERM*s in *HOL* are related so that the denotation of a *well typed TERM* is a value which is a member of the denotation of its *TYPE*.

In general, *TYPE*s in *HOL* may also contain *TYPE variables.* In this case they are known as *polymorphic TYPE*s and should be thought of as denoting a family of *monomorphic TYPE*s.

In this chapter a thorough and systematic (though informal) description of the *TYPE* system is given. This, on a smaller scale, provides a model for the structure of the *TERM* language treated in the next chapter.

To give a full account of the **TYPE** system we provide:

- An *abstract syntax* giving the logical structure of a notation for describing *TYPE*s. This corresponds to the primitive facilities available in the metalanguage for manipulating *TYPE*s.
- A *concrete syntax* providing a specific way of writing *TYPE*s for submission to ProofPower or for presentation in documents.
- A semantics indicating informally how the set denoted by any *TYPE* may be determined.

These suffice to define *TYPE*s as completely as they can be without discussing how they relate to *TERM*s. The chapter goes on to describe the principal non-primitive facilities available in ProofPower for performing computations with *TYPE*s.

2.2 The Abstract Syntax of *TYPE*s

There are just two primitive ways of constructing a *TYPE*. The first is the construction of a **TYPE variable**, which simply has a *name.* A *TYPE variable* denotes an arbitrary set. The second is the construction of a *TYPE* using a **TYPE constructor**. A *TYPE constructor* also has a name, normally one which has previously been declared for the purpose with an associated arity.

The construction is performed on the *TYPE constructor* (which may be thought of as a function from one or more *TYPE*s yielding another *TYPE*) together with a number of *TYPE*s to which the constructor is applied. A special case of a *TYPE constructor* is a 0-ary *TYPE constructor*. This special case encompasses some of the most familiar *TYPE*s such as the *TYPE* ⌜:*BOOL*⌝ which denotes the set of truth values (⌜{*T*,*F*}⌝), and the *TYPE* ⌜:ℕ⌝ which denotes the set of natural numbers (the positive whole numbers).

The abstract syntax may be described by giving the names and *TYPE*s in the metalanguage standard ML of the functions corresponding to the primitive *TYPE constructor*s:

```
SML
|mk_vartype : string               -> TYPE;
|mk_ctype   : string * TYPE list   -> TYPE;
```

2.3 The Concrete Syntax of *TYPE*s

ProofPower provides a parser for *HOL TYPE*s both as free standing standard ML values (of type *TYPE*) and as *TYPE casts* disambiguating the *TYPE* of a *TERM*.

The following consists of a simplified description of the concrete syntax of *TYPE*s, sufficient for the examples which follow, and sufficient to enter any well formed *TYPE*.

```
BNF
|        Type   =    Name
|               |    Typars, Name
|               |    Type, InfixName, Type
|               |    '(', Type, ')';
|        Typars =    Type
|               |    '(', Type, { ',', Type }, ')';
```

The first alternative for a *TYPE* covers both *TYPE* variables and 0-ary *TYPE constructor*s. The parser will disambiguate this by assuming that any name starting with a prime (') is intended as the name of a *TYPE* variable.

The second alternative is the default presentation of a *TYPE* construction, where the name of the *TYPE constructor* follows a comma separated list of *TYPE*s enclosed in round brackets.

Infix syntax is supported for binary *TYPE constructor*s. Such *TYPE constructor*s may be given infix status and a parsing precedence by an appropriate *fixity declaration*.

The last alternative allows brackets to be used to enclose a *TYPE* expression to override the precedence which the parser would otherwise use in parsing the *TYPE*.

2.4 The Semantics of *TYPE*s

*TYPE*s should be thought of as expressions which denote non-empty sets of values. To understand the meaning of the language of *TYPE*s it is therefore necessary to understand for any *TYPE* expression which set is denoted by that expression.

This is complicated slightly by two variable factors. Before the set denoted by any *TYPE* expression can be determined it is first necessary to know the denotations of the *TYPE* variables which occur in it (which will not usually be fixed), and secondly to know the denotations of the *TYPE constructor*s (which will usually have been partly determined by a declaration for the relevant *TYPE constructor*).

The meaning of a *TYPE* expression must therefore be understood to be given relative to an assignment of values to the *TYPE* variables and *TYPE constructor*s which appear in it. Such an assignment will assign to each *TYPE* variable a non-empty set, and to each n-ary *TYPE constructor* a function from n-tuples of non-empty sets (the denotations of the *TYPE* expressions to which the *TYPE constructor* is applied) to non-empty sets (the denotations of the *TYPE* expression formed by applying the *TYPE constructor* to the argument types).

Just describing the necessary context in which the value denoted by some *TYPE* expression must be determined conveys most of the content of the description of the semantics of the *TYPE* system. There are just two ways in which a *TYPE* expression may be formed and these determine the set denoted by the *TYPE* expression as follows.

If the expression is a *TYPE* variable then the set denoted is that assigned to the *TYPE* variable in the context. If the expression is a *TYPE constructor* then the value denoted by the *TYPE* expression is the value obtained by applying the function assigned in the context to the *TYPE constructor* to the tuple of sets which are the denotations of the *TYPE* expressions supplied as arguments to the *TYPE constructor*.

In summary:

- *TYPE*s denote non-empty sets of values.
- *TYPE* variables range over non-empty sets of values.
- *TYPE constructor*s denote functions from tuples of sets to sets.

2.5 Examples of *HOL TYPE*s

The following examples show a variety of *TYPE*s. They also show different ways in which a *TYPE* can be entered into the system. When a *TYPE* is entered as a top level expression to ProofPower it is evaluated to yield a standard ML value of type *TYPE* which is then displayed automatically invoking a pretty printer which will use the concrete syntax for *HOL TYPE*s. These outputs from ProofPower are shown together with the various ways of entering *TYPE*s.

A *TYPE* is a value (of type *TYPE*) in the metalanguage standard ML. The primitive method for entering such a *TYPE* is to enter the standard ML expression which computes the *TYPE* using the ML functions which are the constructors of the abstract data type.

Thus a *TYPE* which is a *TYPE variable* may be obtained by evaluating the ML function *mk_vartype* supplying it with the name of the variable:

SML
```
val t = mk_vartype "'a";
```

ProofPower output
```
val t = ⌜:'a⌝ : TYPE
```

Submitting the above command to the ProofPower system causes the *TYPE* variable with name ''a' to be constructed and bound to the ML name 't'.

A *TYPE* formed from a *TYPE constructor* may be obtained using *mk_ctype* as follows:

SML
```
│val u = mk_ctype ("BOOL",[]);
```

ProofPower output
```
│val u = ⌜:BOOL⌝ : TYPE
```

In the above the 0-ary *TYPE constructor* 'BOOL' is applied to an empty list of *TYPE*s. ⌜:*BOOL*⌝ is the *TYPE* denoting the set of truth values (true and false) and is a primitive *TYPE* of the *HOL* logic.

Type following example is another primitive *TYPE constructor*, the binary function space constructor:

SML
```
│mk_ctype ("→",[⌜:ℕ⌝,⌜:ℕ⌝]);
```

ProofPower output
```
│val it = ⌜:ℕ → ℕ⌝ : TYPE
```

This expression evaluates to the *TYPE* whose denotation in a standard model is the set of all total functions over the natural numbers.

In practice, for most purposes, entry of *TYPE*s as ML expressions is too cumbersome, and for this reason a parser is provided which enables the *TYPE*s to be entered in a convenient concrete syntax. This parser is invoked automatically when a quotation beginning with the symbol '⌜:' is encountered by the ProofPower system.

Thus:

SML
```
│⌜:'a⌝;
```

ProofPower output
```
│val it = ⌜:'a⌝ : TYPE
```

is an alternative way of entering the same *TYPE* variable as that previously bound to 't'.

The *TYPE* formed from the 0-ary constructor 'BOOL' is quoted thus:

SML
```
│⌜:BOOL⌝;
```

ProofPower output
```
│val it = ⌜:BOOL⌝ : TYPE
```

This is parsed as a *TYPE constructor* rather than a *TYPE* variable because its name does not begin with a prime.

The function space *TYPE* may be quoted using concrete syntax as follows:

SML
```
│⌜:ℕ → ℕ⌝;
```

ProofPower output
```
val it = ⌜:ℕ → ℕ⌝ : TYPE
```

This infix concrete representation is permitted for binary *TYPE constructor*s provided that they have been declared previously as infix identifiers in a *fixity declaration*.

Where a constructor has not been declared infix the concrete syntax requires postfix application of the constructor as in the *TYPE* of lists of natural numbers:

SML
```
⌜:(ℕ) LIST⌝;
```

ProofPower output
```
val it = ⌜:ℕ LIST⌝ : TYPE
```

For 1-ary *TYPE constructor*s the brackets surrounding the list of *TYPE*s preceding the constructor name are optional, so:

SML
```
⌜: ℕ LIST⌝;
```

ProofPower output
```
val it = ⌜:ℕ LIST⌝ : TYPE
```

is a quotation which yields exactly the same *TYPE* as the previous quotation.

The following example illustrates a typical use of *TYPE* variable, giving a polymorphic *TYPE* of lists. This enables the normal operations over lists to be defined in such a way that they apply to lists of any *TYPE* of element.

SML
```
⌜:'a LIST⌝;
```

ProofPower output
```
val it = ⌜:'a LIST⌝ : TYPE
```

The quotation facilities are multilingual and support what is sometimes referred to as *anti-quotation*. An **antiquotation** is an expression in ML, quoted inside an object language *HOL* quotation. The quoted ML expression must evaluate to an appropriate type of ML expression (in this case an expression of ML type *TYPE*), and the value of this expression is used at the point of quotation in the *TYPE* constructed by the *TYPE* parser. To supply an ML expression providing a constituent *TYPE*, the quotation symbol ('⌜') is subscripted with **SML** : as shown in this example:

SML
```
⌜: ⌜SML: t⌝ → ⌜SML: u⌝⌝;
```

which is typed into a source document as:

```
⌜: ⌜↘SML:↕ t⌝ → ⌜↘SML:↕ u⌝⌝;
```

and evaluates as:

ProofPower output
```
val it = ⌜:'a → BOOL⌝ : TYPE
```

The ML name 't' has previously been bound to the *TYPE* ⌜:'a⌝and 'u' to *TYPE* ⌜:*BOOL*⌝. These are is supplied as arguments to the function space binary infix *TYPE constructor* to yield the *TYPE* of boolean valued functions over the natural numbers (which may be thought of as properties of natural numbers).

This is the same *TYPE* as would have been obtained from the expression:

SML
```
mk_ctype("→",[t,u]);
```

ProofPower output
```
val it = ⌜:'a → BOOL⌝ : TYPE
```

Other examples of types formed from infix *TYPE constructor*s are:

SML
```
⌜:ℕ × ℕ⌝;              (* pairs of natural numbers *)
```

ProofPower output
```
val it = ⌜:ℕ × ℕ⌝ : TYPE
```

SML
```
⌜:ℕ + BOOL⌝;           (* disjoint union of ℕ and BOOL *)
```

ProofPower output
```
val it = ⌜:ℕ + BOOL⌝ : TYPE
```

2.6 Computation with *TYPE*s

A full set of constructors, recognisers and destructors for the two different kinds of *TYPE* are provided as the primitive operations of the abstract data type in standard ML. In addition to these primitive operators a range of higher level facilities are available. Those provided include all the major facilities required in implementing a proof system for a logic with this *TYPE* system, and provide a good basis for any further programming with *TYPE*s which the user may wish to do to extend or customise the capabilities of the system. Most users will find however, that programming with *TYPE*s is not necessary for their application.

2.6.1 Recognisers and Destructors

Corresponding to the two primitive constructors already introduced for building *TYPE*s by computation, there are recognisers, which may be used to discover whether a *TYPE* in hand was made by a particular constructor, and destructors, which may be used to take apart the *TYPE* again, yielding the values used originally as arguments to the constructor function.

These follow systematic naming conventions, the prefix **mk_** used in the constructor being replaced by **is_** in the name of the recogniser and **dest_** in the name of the destructor. This pattern of naming conventions is repeated many times in ProofPower, wherever a new language is considered. The pattern is repeated for the language of primitive *TERM*s in *HOL*, the language of *derived TERM*s, and for *TERM*s representing fragments of Z specifications.

It suffices here simply to record this set of ML procedure names and their ML types, leaving the reader to infer their functionality from the above informal description:

- constructors

```
SML
│mk_vartype   :string            -> TYPE;
│mk_ctype     :string*TYPE list  -> TYPE;
```

- recognisers

```
SML
│is_vartype   :TYPE -> bool;
│is_ctype     :TYPE -> bool;
```

- destructors

```
SML
│dest_vartype :TYPE -> string;
│dest_ctype   :TYPE -> string * TYPE list;
```

2.6.2 Basic Facilities

Using the primitive facilities, higher level facilities may be programmed, usually by recursion over the structure of *TYPE*s. The following are examples of such higher level facilities supplied with ProofPower.

- *TYPE* equality

  In order to retain flexibility over the internal representation of *TYPE*s they are not an 'equality type' in ML, and therefore cannot be compared using the ML equality relation. A function performing a comparison between two *TYPE*s is therefore provided, in the form of the infix ML boolean valued operation '=:'.

```
SML
│op =: : TYPE * TYPE -> bool;
```

- *TYPE* variables in a *TYPE*

  The following function will extract the names of all the *TYPE* variables used in a *TYPE*.

```
SML
│type_tyvars : TYPE -> string list;
```

- *TYPE constructor*s in a *TYPE*

  Similarly a function is available to extract the *TYPE constructor*s. In this case both the name and the arity of the *TYPE constructor* are extracted as a pair.

```
SML
│type_tycons : TYPE -> (string * int) list;
```

- *TYPE* instantiation

  When a polymorphic *TERM* is instantiated to a specific *TYPE* a substitution is made for *TYPE variables* occuring in the *TYPE*. This is known as *TYPE instantiation* and is performed by the following function:

```
SML
│inst_type : (TYPE * TYPE) list -> TYPE -> TYPE;
```

The first parameter is a list of pairs of *TYPE*s. Each pair consists of a *TYPE* to be substituted and a *TYPE* variable for which the substitution takes place.

2.6.3 Support for Pattern Matching

Standard ML provides support for pattern matching in function definitions. This support depends on the functions being defined over ML ‘datatype’s. Pattern matching is not available for value of ‘abstract types’. In general the ML types used for representing object language constructs have to be implemented as abstract types in order that necessary well-formedness conditions are enforced, including the prevention of unsound derivations.

To mitigate the inconvenience caused to programmers we have made available a number of ML datatypes suitable for pattern matching function definitions, together with transfer functions which convert values of these abstract types into values of the corresponding datatypes.

The first example of such a type is the type *DEST_SIMPLE_TYPE*, which is defined as:

```
datatype DEST_SIMPLE_TYPE =
        Vartype of string
|       Ctype of (string * TYPE list);
```

This type reflects the top level structure of a *TYPE*, the constructors of the datatype corresponding to the primitive constructors for *TYPE*s.

Using this datatype generalised constructors and destructors for *TYPE*s are provided:

- generalised constructor

  SML
  ```
  mk_simple_type     : DEST_SIMPLE_TYPE -> TYPE;
  ```

- generalised destructor

  SML
  ```
  dest_simple_type   : TYPE -> DEST_SIMPLE_TYPE;
  ```

The advantage of these facilities is:

- They enable functions over *TYPE*s to be defined using pattern matching.
- They provide a convenient interactive interface for investigating the underlying structure of a *TYPE*.

To illustrate the first advantage we provide an example of a function over *TYPE*s defined using pattern matching. The following is in effect a re-implementation of one of the functions provided with ProofPower. It traverses a *TYPE* forming a list of all the *TYPE variables* appearing in it.

SML
```
fun type_tyvars2 t =
 (fn    Vartype s       => [s]
  |     Ctype (s,tl)    => list_cup (map type_tyvars2 tl))
 (dest_simple_type t);
```

The argument to the function ‘t’ is a *TYPE*. This is transformed into a *DEST_SIMPLE_TYPE* by *dest_simple_type* and then passed to a function abstraction defined using pattern matching over the structure of the datatype.

The function *list_cup*, which forms the union of two lists regarded as sets (i.e. removing duplicates), is one of a number of general purpose functions supplied with ProofPower to supplement the standard ML library in areas relevant to the functionality of ProofPower.

HOL *TERM*s

3.1 Introduction

Following the structured presentation of *TYPE*s we now present the ProofPower-HOL **TERM** language.

The primitive *TERM* structure is first presented, covering:

- Abstract Syntax
- Concrete Syntax
- Typing Rules
- Semantics

Overlaid on this simple primitive structure there is a syntactically richer derived syntax, all of which can be explained as more convenient notation for writing *TERM*s which can also be written in the primitive language.

The derived syntax is described also, and the semantics of these constructs is illustrated.

When entering *TERM*s using the concrete syntax it is usually necessary to use 'Quine corners' ('⌜' and '⌝') as quotation marks. A pair of Quine corners enclosing a piece of concrete syntax is understood by ProofPower as an ML expression of type *TERM*. The effect of evaluating such an expression is achieved by parsing the concrete syntax, performing *TYPE* inference to determine whether the expression is *well typed*, and constructing the appropriate *TERM*. When an expression is submitted to ProofPower which results in the computation of a value of type *TERM*, the *TERM* pretty printer is automatically invoked to print the *TERM*, and will print the *TERM* in *HOL* concrete syntax enclosed in Quine corners. These features will be illustrated in the following material.

3.2 Abstract Syntax

The primitive abstract syntax of *TERM*s has just four constructors.

A *TERM* is either a *TERM variable*, a *constant*, an *application*, or a *lambda-abstraction*.

A *TERM variable* may be thought of as denoting unspecified value, a *constant* as denoting a previously defined value. An *application* denotes the value of some function when applied to an argument, and a *lambda-abstraction* describes a function by specifying the value of the function for an arbitrary value of its argument.

The standard ML datatype *DEST_SIMPLE_TERM* reflects the top-level structure of HOL *TERM*s and may be used in computations with *TERM*s.

```
datatype DEST_SIMPLE_TERM =
              Var        of string * TYPE
        |     Const      of string * TYPE
        |     App        of TERM * TERM
        |     Simpleλ    of TERM * TERM;
```

A *TERM* may be transformed into a *DEST_SIMPLE_TERM*, or vice-versa by the following functions.

```
dest_simple_term   : TERM -> DEST_SIMPLE_TERM;
mk_simple_term     : DEST_SIMPLE_TERM -> TERM;
```

Alternatively a full collection of constructors, recognisers and destructors which operate directly on *TERM*s are available. The names of these are formed by prefixing *mk_*, *is_* and *dest_* to the names of the constructors in the abstract data type above (without capitalisation).

3.3 Concrete Syntax

The following BNF describes a very limited subset of the concrete syntax of HOL *TERM*s which is sufficient for expressing *TERM*s according to their primitive structure. Later we we see a richer syntax which makes the notation more readable.

The clauses in this syntax describe:

1. **lambda abstraction**s
2. **function application**s
3. **infix function application**s
4. **type cast**s
5. **variable**s or **constant**s
6. **bracketted expression**s

in turn.

BNF
```
Term  =
              ‘λ‘, Name, [‘:‘, Type], ‘•‘, Term
        |     Term, Term
        |     Term, InfixName, Term
        |     Term, ‘:‘, Type
        |     Name
        |     ‘(‘, Term, ‘)‘;
```

Names are treated as variables unless they have been previously declared as constants. Infix status and priority are determined by fixity declarations.

3.4 *TYPE*s of *TERM*s

Terms must be **well typed**.

The *TYPE* of a *TERM* is determined by *TYPE* inference using the following rules:

3.4.1 Variables

A *TERM variable* in the abstract syntax (which may be thought of as its internal representation), consist simply of a name and a *TYPE*. The variable has the *TYPE* associated with it, and this is essentially an axiom schema in the *TYPE* inference system

In concrete syntax the *TYPE* of a variable may be explicitly cited using a *TYPE-cast*, in which case the variable constructed by the system in forming the *TERM* from the quotation will have that *TYPE* associated with it (provided a well-typing for the *TERM* as a whole can be discovered which is consistent with all the contraints imposed, otherwise an error report will be given and no *TERM* will be constructed.

This is reflected in the form of the axiom schema, which is expressed using the concrete syntax.

$$\frac{}{\ulcorner v{:}\alpha \urcorner : \alpha}$$

The schema states that the *TYPE* of any variable quoted with a cast is the same as the *TYPE* used in the cast.

Where a variable is not given a cast a most general (possibly polymorphic) *TYPE* will be inferred for it (if possible), and this will be used in the construction of the *TERM*. Nevertheless, the principle hold that in the asbtract representation it has the *TYPE* which was associated with it when it was formed.

3.4.2 Constants

A similar rule holds for *constants*. Each constant is also constructed as a constant name associate with a *TYPE*, and the *TYPE* supplied at the time of construction is the *TYPE* of the constant thus formed.

When constants are constructed by the system while evaluating an object language quotation, the *TYPE* inference rules adopted differ from those used for variables. In particular a name parsed will only be interpreted as a constant if a constant of that name has been declared in the current scope. The *TYPE* inferrer will also insist on the *TYPE* assigned to the constant being the *TYPE* associated with the constant when it was declared, or a *TYPE* instance of that *TYPE*.

$$\frac{}{\ulcorner c{:}\alpha \urcorner : \alpha}$$

3.4.3 Lambda Abstractions

A *lambda abstraction* denotes a function from values of the *TYPE* of its formal parameter to the values of the *TYPE* of the body of the lambda abstraction. The *TYPE* of the lambda abstraction is therefore a function space *TYPE* where the first argument to the function space constructor (the domain *TYPE*) is the *TYPE* of the formal parameter (or bound variable) and the second argument to the function space constructor (the co-domain *TYPE*) is the *TYPE* of the body of the lambda abstraction.

$$\frac{t\ :\ \alpha}{\ulcorner\lambda\ x{:}\beta\ \bullet\ t\urcorner\ :\ \beta\rightarrow\alpha}$$

The rule then states that if some *TERM* ‘t’ has *TYPE* α, then a lambda abstraction with *TYPE* cast β on its bound variable and body ‘t’ has *TYPE* $\ulcorner{:}\ \beta\rightarrow\alpha\ \urcorner$.

3.4.4 Applications

For a *function application* to be well-*TYPE*d the function must have a function space *TYPE*, and the argument must have the same *TYPE* as the domain *TYPE* of the function. In that case the application will have the same *TYPE* as the codomain of the function.

$$\frac{f\ :\ \alpha\rightarrow\beta;\ x\ :\ \alpha}{\ulcorner f\ \ x\urcorner\ :\ \beta}$$

3.4.5 Type Rules in the Meta-language

The same rules may be rendered in ML as follows:

The main difficulty in expressing rules of this kind in the metalanguage is that the metalanguage is a programming language, not a logic, and therefore it only contains constructs which are executable, which does not include universal quantifiers.

Nevertheless something similar to a free variable formulation of the rules can be given. This is an expression in ML containing variables which are to be understood as uninterpreted, together with the convention that to assert an ML expression containing such uninterpreted variables is to assert that the expression will evaluate to the boolean value ‘true’ whatever well-*TYPE*d value is substituted for the instances of uninterpreted variables, provided that distinct occurences of the same variable are assigned the same value throughout the expression.

Such a claim cannot be verified without a logic for the metalanguage, which we do not have, however it can be illustrated (or tested) by evaluating the expressions for one or more specific values.

Thus all the following ML encapsulations of the HOL *TYPE* inference rules have the following characteristic. If for a particular set of values of the free variables occuring in it, the premises of the rule (the assertions above the line) evaluate to the value ‘true’, then so will the conclusion. This can be illustrated by evaluating them after making the following specific bindings:

SML
```
val vname = "var";
val vtype = ⌜:BOOL⌝;
val cname = "0";
val ctype = ⌜:ℕ⌝;
val term = ⌜0⌝;
val ttype = ⌜:ℕ⌝;
val funterm = ⌜fun : 'a → 'b⌝;
val arg = ⌜arg : 'a⌝;
```

- variables

$$\frac{}{\mathit{type\_of}\ (\mathit{mk\_var}(\mathit{vname},\mathit{vtype})) =: \mathit{vtype};}$$

- constants

$$\frac{}{\mathit{type\_of}\ (\mathit{mk\_const}(\mathit{cname},\mathit{ctype})) =: \mathit{ctype};}$$

- lambda abstractions

$$\frac{\mathit{type\_of}\ \mathit{term} =: \mathit{ttype};}{\mathit{type\_of}\ \ulcorner\lambda\ x{:}'a \bullet\ {}_{\text{ML}}\ulcorner \mathit{term}\urcorner\urcorner =: \ulcorner{:}'a \to \ulcorner_{SML:}\ \mathit{ttype}\urcorner\urcorner;}$$

- applications

$$\frac{\mathit{type\_of}\ \mathit{funterm} =: \ulcorner{:}'a \to 'b\urcorner; \quad \mathit{type\_of}\ \mathit{arg} =: \ulcorner{:}'a\urcorner;}{\mathit{type\_of}\ \ulcorner{}_{\text{ML}}\ulcorner \mathit{funterm}\urcorner\ {}_{\text{ML}}\ulcorner \mathit{arg}\urcorner\urcorner =: \ulcorner{:}'b\urcorner;}$$

In fact the function *type_of* which is supplied in the ProofPower system can be defined by recursion in a manner very similar to the above statement of the *TYPE* inference rules.

3.5 Types of Terms - Examples

The following examples use the same method of expressing in the metalanguage claims about the *TYPE* of a *TERM* by citing ML expressions which evaluate to 'true'.

They also give a preview of how the typed lambda calculus is made into higher order logic, by showing the *TYPE*s of some of the logical connectives that are defined in the system.

SML
```
type_of    ⌜x:ℕ⌝              =:   ⌜:ℕ⌝;
type_of    ⌜x:'a⌝             =:   ⌜:'a⌝;
type_of    ⌜0⌝                =:   ⌜:ℕ⌝;
type_of    ⌜λx:ℕ • x + 1⌝     =:   ⌜:ℕ → ℕ⌝;
type_of    ⌜λx • x + 1⌝       =:   ⌜:ℕ → ℕ⌝;
type_of    ⌜(λx • x + 1) 3⌝   =:   ⌜:ℕ⌝;
type_of    ⌜$+ 1⌝                  =:   ⌜:ℕ → ℕ⌝;
type_of    ⌜$+⌝               =:   ⌜:ℕ → ℕ → ℕ⌝;
type_of    ⌜T⌝                =:   ⌜:BOOL⌝;
type_of    ⌜¬ T⌝              =:   ⌜:BOOL⌝;
type_of    ⌜$¬⌝               =:   ⌜:BOOL → BOOL⌝;
type_of    ⌜$∧⌝               =:   ⌜:BOOL → BOOL → BOOL⌝;
type_of    ⌜$∀⌝               =:   ⌜:('a → BOOL) → BOOL⌝;
```

3.6 The Semantics of *TERM*s

The semantics of *TERM*s is explained informally by giving rules for evaluation of a *TERM* to yield the value denoted by the *TERM*. This does not imply that the *TERM*s are executable, the evaluation rules will often involve application of non-computable functions.

As with *TYPE*s, *TERM*s can only be given a value in an appropriate context.

In the case of *TYPE*s the necessary context was:

- A **TYPE interpretation** identifying the *TYPE* universe (a collection of non-empty sets which represents the domain of discourse) and assigning values to the *TYPE constructor*s in use.
- An assignment of sets to the *TYPE variable*s occuring in the *TYPE*.

In the case of *TERM*s the same context is necessary to enable a denotation to be assigned to the *TYPE*s occurring in the *TERM*, and in addition the following are required:

- An interpretation which assigns to each constant a family of values, one for each monomorphic instance of the *TYPE* of the constant, each such value being a member of the denotation under the *TYPE* interpretation of the relevant monomorphic *TYPE*.
- For each assignment of *TYPE*s to the *TYPE* variables occuring in the *TERM*, an assignment of values for each distinct variable having free occurrences in the *TERM*. Two variables are considered the same only if their names and their *TYPE*s are the same. The values assigned to the variables must be members of the set denoted by their *TYPE* in the context of the *TYPE* variable assignment.

In such a context the denotation of a *TERM* can be established by rules depending on the top level abstract constructor as follows:

- **Variables**

  A variable denotes the value assigned to it under the variable assignment.
- **Constants**

  A constant denotes the value assigned to the constant in the interpretation for the specific values assigned to *TYPE* variables in the current *TYPE*-variable assignment.
- **Lambda Abstractions**

  A lambda abstraction denotes the function whose domain is the set assigned to the *TYPE* of its bound variable in the current context and whose value at any point 'p' is the value assigned to the *TERM* which is the body of the lambda expression in the context formed from the current context by replacing the value assigned to the free variable having the same name aned *TYPE* as the bound variable of the abstraction by the value 'p'.
- **Applications**

  Denotes the value of the function denoted by the first constituent *TERM* at the point which is the value denoted by the second constituent *TERM*.

3.7 Semantics of *TERM*s - Examples

In these examples we use proof in ProofPower to illustrate the semantics of HOL *TERM*s. The particular proof facilities used will be explained in greater detail later. For present purposes it is sufficient to know that an ML name ending in *_conv* is the name of a *conversion*, and that conversions are functions which take *TERM*s as arguments and return theorems after constructing a proof of the theorem behind the scenes. The theorems returned will normally be equations of which the left hand side is the *TERM* supplied as argument, and the right hand side is some transformation of that *TERM*.

The first example illustrates how the semantics of application and of lambda abstraction fit together to give $\beta -$ **reduction**. This is the technical term for the process of substituting an argument of a function into the body of the function definition.

SML

```
β_conv ⌜(λx • x + 1) 3⌝;
```

Hol Output

```
val it = ⊢ (λ x• x + 1) 3 = 3 + 1 : THM
```

β_conv is a very specific inference facility which does no more than the substitution. In general we will illustrate semantic features using the more powerful higher level rewriting facilities which will do *β-reduction*, and many other useful simplifications.

SML

```
rewrite_conv[] ⌜(λx • x + 1) 3⌝;
```

Hol Output

```
val it = ⊢ (λ x• x + 1) 3 = 4 : THM
```

In the above case not only was the *β-reduction* performed but also the evaluation of the resulting subexpression formed from numeric literals.

η_axiom is one of the primitive axioms of the HOL logic. Its purpose is to assert that functions in HOL are *extensional.*

SML

```
η_axiom;
```

Hol Output

```
val it = ⊢ ∀ f• (λ x• f x) = f : THM
```

This is more transparently stated by *ext_thm*, a theorem derived from *η_axiom. ext_thm* states that two functions are equal if and only if they have the same values at every point in their domains. (Quantification over the domain is ensured by the *TYPE* inferrer, given that all functions in HOL are total functions over their domain *TYPE*.)

SML

```
ext_thm;
```

Hol Output

```
val it = ⊢ ∀ f g• f = g ⇔ (∀ x• f x = g x) : THM
```

The final examples are intended to illustrate the facts that:

- ⌜43⌝ is a natural number.
- ⌜T⌝ is of *TYPE* ⌜:BOOL⌝.
- The values of *TYPE* ⌜:ℕ⌝ are all greater than or equal to zero.
- There are just two values of *TYPE* ⌜:BOOL⌝, ⌜T⌝ and ⌜F⌝ (true and false).

The facts are all provable using the automatic proof facilities provided in ProofPower, which the reader may verify by pasting in the standard ML expressions below into ProofPower and observing that the *TERM* supplied as argument is transformed into a theorem (by proof behind the scenes).

```
SML
│prove_rule [] ⌜∃ x:ℕ       •      43 = x⌝;
│prove_rule [] ⌜∃ b:BOOL    •      T = b⌝;
│prove_rule [] ⌜∀ x:ℕ       •      x ≥ 0⌝;
│prove_rule [] ⌜∀ b:BOOL    •      b = T ∨ b = F⌝;
```

3.8 Derived Syntax for *TERM*s

The primitive syntax described above for HOL *TERM*s is complete in the sense that every HOL *TERM* can be built up or taken apart using only the primitive constructors and destructors. In the same way the primitive concrete syntax is complete, permitting any *TERM* to be written down as applications or abstractions formed from variables or constants.

This Spartan concrete syntax is however not ideal for writing specifications in a concise and readable form. For this purpose it is desirable to be able to adopt a richer variety of syntactic forms corresponding more with normal mathematical usage.

Product HOL provides 'syntactic sugar' which permits specifications to be written in a more readable way. These forms are also known as 'derived syntax'. Coresponding to the derived syntax supported by the HOL parser there are computational facilities which permit *TERM*s to be processed in ways which correspond to the derived forms rather than the primitive forms.

To some extent it is arbitrary what features are treated in this way as part of the range of fully supported 'derived syntax'. A new datatype *DEST_TERM* is provided which gives a computational view of the derived syntax of HOL *TERM*s. A full range of constructors destructors and discriminators corresponding to the categories in this datatype are also available.

3.8.1 DEST_TERM

The full definition of the datatype *DEST_TERM* is as follows:

```
datatype DEST_TERM =
        DVar        of string * TYPE
|       DConst      of string * TYPE
|       DApp        of TERM * TERM
|       Dλ          of TERM * TERM
|       DEq         of TERM * TERM
|       D⇒          of TERM * TERM
|       DT
|       DF
|       D¬          of TERM
|       DPair       of TERM * TERM
|       D∧          of TERM * TERM
|       D∨          of TERM * TERM
|       D⇔          of TERM * TERM
|       DLet        of ((TERM * TERM)list * TERM)
|       DEnumSet    of TERM list
|       D∅          of TYPE
|       DSetComp    of TERM * TERM
|       DList       of TERM list
|       DEmptyList  of TYPE
|       D∀          of TERM * TERM
|       D∃          of TERM * TERM
|       D∃1         of TERM * TERM
|       Dϵ          of TERM * TERM
|       DIf         of (TERM * TERM * TERM)
|       Dℕ          of int
|       DChar       of string
|       DString     of string;
```

The generalised destructor and constuctor for this derived abstract syntax are *dest_term* and *mk_term* respectively, which are ML functions with the following types.

SML
```
dest_term    : TERM -> DEST_TERM;
mk_term      : DEST_TERM -> TERM;
```

Specific constructors, destructors and discriminators for each category in this may be obtained by the following algorithm:

1. Take the name of the corresponding constructor from the definition of DEST_TERM.
2. Drop the initial 'D'.
3. prefix with *mk_*, *dest_* or *is_* as appropriate.
4. Change to lower case every upper case letter in the name and insert an underscore character in front of it.

The signature of the constructors and destructors can be obtained directly from the type associated with the corresponding constructor in the datatype *DEST_TERM*.

Alternatively the *KWIC* index to the reference manual may be used to find a full listing of syntactic constructors and destructors.

For example, corresponding to the constuctor 'DSetComp' three functions are available as follows:

```
SML
|mk_set_comp       : TERM * TERM -> TERM;
|dest_set_comp     : TERM -> TERM * TERM;
|is_set_comp       : TERM -> bool;
```

It may be noted that in this structure the first four items correspond to the four constructors in the primitive abstract syntax, and therefore, in principle any *TERM* could be classified as one of these four. However, when a *TERM* is transformed into a *DEST_TERM* using the function *dest_term*, it will only be treated as a constant, a application or an abstraction if it cannot be interpreted as one of the following derived *TERM*s (all of which are in primitive *TERM*s, either constants, applications or abstractions).

The derived syntax available for TERMs may be classified and discussed in then following categories:

- prefix, infix and postfix operators
- binders
- pair matching lambda abstractions
- conditionals
- local definitions
- set displays and abstractions
- list displays
- literals (numeric, character, and string)

Of these the first two categories represent a general facility where any name can be given a special lexical status and will then be used in the concrete syntax in that special way. This does not affect the computational processing of *TERM*s.

Certain of the categories in the datatype DEST_TERM represent the treatment of important constants defined in the system as if they were fixtures of the language. This has been done primarily for those constructs which in first order logic are normally considered as built in features of the language rather than as defined primitive or defined constants, which is their true status in HOL. This includes the normal boolean operators, the quantifiers, and the constants 'T' and 'F' representing formulae which are true and false respectively in all standard models.

3.8.2 Binders

Constants having *TYPE*: $\ulcorner:('a \to 'b) \to 'c\urcorner$ (or any instance of this) may be declared as **binder**s.

In normal use a constant which has been declared as a *binder* must be applied to a lambda expression with the ' λ' symbol omitted. The lexical status can be suspended by prefixing the name by \$, and this must be done if it is required to use the name in any way other than by applying it to a lambda expression.

The following illustrates that an existentially quantified *TERM* is just the same as an application of the constant ⌜ $∃ ⌝ to a lambda expression (in this case 'just the same as' means 'is evaluated to yield exactly the same *TERM*').

SML
```
⌜∃ x• x = 4⌝ =$ ⌜$∃ λ x• x = 4⌝;
```

3.8.3 Nested Paired Abstractions

Nested lambda abstractions (often called *curried*) can be abbreviated as follows:

SML
```
⌜λx:ℕ•λy:ℕ• (x,y)⌝ =$ ⌜λ x y:ℕ• (x,y)⌝;
```

The second occurrence of λ being omitted, together with the preceding •. The function denoted by the nested abstraction takes two natural numbers and returns a pair. (“,” is the infix pairing operator.)

Functions taking pairs may be written as 'pattern matching lambda abstractions':

SML
```
rewrite_conv[] ⌜(λ(x,y):ℕ × ℕ• x)=Fst⌝;
```

ProofPower output
```
val it = ⊢ (λ (x, y)• x) = Fst ⇔ T : THM
```

This − *abstraction* takes an argument which is an ordered pair, and returns the first element of the pair.

Fst is the function defined in ProofPower to select the first element from an ordered pair. By use of extensionality of functions and the definition of *Fst* the rewriting facilities have reduced the given equation to ⌜T⌝.

These effects can be iterated or combined.

SML
```
rewrite_conv []
        ⌜(λ(x,y):ℕ × ℕ; ((v,w),z)• x + y + v + w + z) (1,2) ((3,4),5)⌝;
```

ProofPower output
```
val it =
        ⊢ (λ (x, y) ((v, w), z)• x + y + v + w + z) (1, 2) ((3, 4), 5) = 15 : THM
```

3.8.4 Conditionals

Conditionals may be written:

if t1 **then** t2 **else** t3

This *TERM* denotes the same value as the *TERM* ⌜ t2 ⌝ in contexts in which the *TERM* ⌜ t1 ⌝ denotes ⌜T⌝, otherwise it denotes the same value as the *TERM* ⌜t3⌝.

This is illustrated by the following examples:

SML
```
rewrite_conv[] ⌜if T then 0 else 1⌝;
```

ProofPower output
```
val it = ⊢ (if T then 0 else 1) = 0 : THM
```

SML
```
rewrite_conv[] ⌜if F then 0 else 1⌝;
```

ProofPower output
```
val it = ⊢ (if F then 0 else 1) = 1 : THM
```

SML
```
rewrite_conv[] ⌜if 3>6 then x else y⌝;
```

ProofPower output
```
val it = ⊢ (if 3 > 6 then x else y) = y : THM
```

3.8.5 Let Clauses

Local declarations may be made in the form:

let defs **in** term

This *TERM* denotes in any context the same value as the *TERM* ⌜term⌝ denotes in the context obtained by modifying the first context by subsituting the values denoted by the defining expressions in 'defs' to the variables to which they are bound in 'defs'. i.e. it denotes the value of *term* when evaluated in the context of the local definitions in *defs*. To eliminate a *let* clause, rewrite with **let_def**.

SML
```
rewrite_conv[let_def] ⌜let a = "Peter" in a,a⌝;
```

ProofPower output
```
val it = ⊢ (let a = "Peter" in (a, a)) = ("Peter", "Peter") : THM
```

The left hand side of a definition may be a variable structure (known as a **varstruct**) formed from simple variables using the infix pair constructor ',':

SML
```
rewrite_conv[let_def] ⌜let (x,y) = (1,T) in (y,x)⌝;
```

ProofPower output
```
val it = ⊢ (let (x, y) = (1, T) in (y, x)) = (T, 1) : THM
```

The left hand side of a definition may also be a (non-recursive) function definition:

SML

```
rewrite_conv[let_def] ⌜let f x = x*x in f 3⌝;
```

ProofPower output

```
val it = ⊢ (let f x = x * x in f 3) = 9 : THM
```

Multiple definitions may be given in a single let clause.

SML

```
rewrite_conv[let_def] ⌜let a = 1 and b = 2 in (a,b)⌝;
```

ProofPower output

```
val it = ⊢ (let a = 1 and b = 2 in (a, b)) = (1, 2) : THM
```

Multiple definitions are 'evaluated in parallel', i.e. the variable introduced by one definition is only in scope in the body of the let clause, not in the other definitions.

SML

```
rewrite_conv[let_def] ⌜let a = 2 and b = a*a in (a,b)⌝;
```

ProofPower output

```
val it = ⊢ (let a = 2 and b = a * a in (a, b)) = (2, a * a) : THM
```

If the first definition is required to be in scope for the second definition then nested let clauses should be used, e.g.:

SML

```
rewrite_conv[let_def] ⌜let a = 2 in let b = a*a in (a,b)⌝;
```

ProofPower output

```
val it = ⊢ (let a = 2 in let b = a * a in (a, b)) = (2, 4) : THM
```

3.8.6 Set Displays

Sets may be entered as *TERM*s by enumeration using the normal mathematical syntax, except that semi-colons are used instead of commas as separators in the list of elements:

SML

```
rewrite_conv[]⌜9 ∈ {1*1; 2*2; 3*3; 4*4}⌝;
```

ProofPower Output

```
val it = ⊢ 9 ∈ {1 * 1; 2 * 2; 3 * 3; 4 * 4} ⇔ T : THM
```

SML

```
rewrite_conv[]⌜10 ∈ {1*1; 2*2; 3*3; 4*4}⌝;
```

ProofPower Output

```
val it = ⊢ 10 ∈ {1 * 1; 2 * 2; 3 * 3; 4 * 4} ⇔ F : THM
```

Sets may also be entered as set abstractions:

SML
```
rewrite_conv[]⌜9 ∈ {x | x < 12}⌝;
```

ProofPower Output
```
val it = ⊢ 9 ∈ {x|x < 12} ⇔ T : THM
```

SML
```
rewrite_conv[]⌜z ∈ {(x, y) | x < y}⌝;
```

ProofPower Output
```
val it = ⊢ z ∈ {(x, y)|x < y} ⇔ Fst z < Snd z : THM
```

3.8.7 List Displays

A similar syntax is available for lists:

SML
```
rewrite_conv[append_def]
        ⌜[1*1; 2*2; 3*3; 4*4] @ [5*5]⌝;
```

ProofPower Output
```
val it = ⊢
        [1 * 1; 2 * 2; 3 * 3; 4 * 4] @ [5 * 5] = [1; 4; 9; 16; 25] : THM
```

SML
```
⌜Cons 1 [2;3;4;5]⌝;
```

ProofPower Output
```
val it = ⌜[1; 2; 3; 4; 5]⌝ : TERM
```

3.8.8 Literals

Numeric literals consist of a sequence of decimal digits (no sign):

SML
```
dest_simple_term ⌜123⌝;
```

ProofPower output
```
val it = Const ("123", ⌜:ℕ⌝) : DEST_SIMPLE_TERM
```

As shown this is interpreted as a constant of *TYPE* ⌜:ℕ⌝ whose name is the same as the literal itself (except that leading zeros will have been omitted.

Character literals consist of a single character in ‘ characters:

SML
```
dest_simple_term ⌜‘α‘⌝;
```

ProofPower output

```
val it = Const ("'α", ⌜:CHAR⌝) : DEST_SIMPLE_TERM
```

Again this is interpreted as a constant, in this case of *TYPE* ⌜:CHAR⌝. The name is prefixed by a ' character in order to prevent its name clashing with the name of any constant introduced by a user of ProofPower (but it is not terminated by ').

String literals consist of zero or more characters in "\"" characters:

SML

```
dest_simple_term ⌜"many characters αβγ"⌝;
```

ProofPower output

```
val it = Const ("\"many characters αβγ",
        ⌜:CHAR LIST⌝) : DEST_SIMPLE_TERM
```

In this case the name of the constant is prefixed with but not terminated by ".

A string literal denotes a LIST of characters:

SML

```
TOP_MAP_C string_conv ⌜"characters αβγ"⌝;
```

ProofPower output

```
val it = ⊢ "characters αβγ"
    = ['c'; 'h'; 'a'; 'r'; 'a'; 'c'; 't'; 'e'; 'r'; 's'; ' '; 'α'; 'β'; 'γ'] : THM
```

THEORIES

Having decribed the HOL *TYPE* and *TERM* languages, we now consider other aspects of ProofPower which enable these to be used in writing specifications and conducting formal proofs.

The term **specification** has a dual use in the following. It is used to describe a quantitity of formal material perhaps consisting of many documents and spread over several **THEORY**s. It is also used in the name of some of the procedures available for entering parts of a specification into the system (usually abbreviated to *spec*), or of *paragraphs* which provide an alternative concrete syntax for undertaking such specifications.

A specification (in the first sense), when processed by ProofPower extends the logical system which ProofPower provides with new *TYPE* constructors and constants which constitute mathematical models of the system specified. The means of extension may be classified into those which are *conservative* and those which are not. A conservative extension is one which is guaranteed safe, insofar as its introduction will not render the logical system inconsistent. For most purposes conservative extensions are sufficient, though it is sometimes desirable to make non-conservative extensions, either for reasons of cost and convenience, or (very rarely in applications) because there is a real need to strengthen the logic. Because of the risk associated with non-conservative extensions these are always recorded as new *axioms*. The use of an axiom is confined to the theory in which it is declared and its descendants, and it is always possible to discover the full set of axioms which are present in a theory and its ancestry.

- Information about specifications is held in the theory database.
- The theory database consists of a hierarchy of theories interconnected by the parent-child relationship.
- The details of a specification are put in the theories using various declarations, definitions and specifications which are calls to ML functions, or by the use of paragraphs which provide an alternative syntax for some of these procedure calls which avoids explicit use of the meta-language.

4.1 Theories

4.1.1 Introduction

A HOL theory contains the following information:

- The name of the theory and the names of its parents and children.
- The names and arities of *TYPE* constructors declared in the theory.
- The names and *TYPE*s of constants declared in the theory.
- Fixity and aliasing information.

- Possibly some axioms.
- Definitions of constants.
- A collection of saved theorems.

4.1.2 Access to Theories

To use a theory it must be “in context”, this can be achieved by opening the theory or one of its descendents:

SML
```
open_theory : string -> unit;
```

Open theory makes the named theory the **current theory**.

The theory ‘basic_hol’ must always be in context, since its ancestors contain definitions on which the soundness of the built-in inference rules depend. The ancestors of *basic_hol* may not themselves be opened (since this would permit theories to be created with incorrect variants of these critical definitions), but will always be in the ancestry of the currently opened theory.

To display the contents of a theory use:

SML
```
print_theory : string -> unit;
```

print_theory takes as parameter the name of the theory to be printed, and accepts the abbreviation `"-"` instead of the current theory name. The theory must be in context, i.e. the current theory or an ancestor of it.

To create a new theory which is a child of the current theory:

SML
```
new_theory : string -> unit;
```

new_theory will create a new theory whose parent is the current theory and whose name is the string supplied as parameter. This new theory will then become the current theory.

To add a new parent to the current theory:

SML
```
new_parent : string -> unit;
```

new_parent will add a new parent to the current theory. This enables the contents of the parent theory and any of its ancestors to be used in the current theory or its descendants.

To get things from the theory a range of functions are provided. These will normally be used only when the value retrieved is required for computations rather than for display, since *print_theory* will display the information suitably formatted, while these functions simply return the value rather than formatting it for display.

These functions, with the sole exception of *get_spec* take a string parameter which is the name of the theory to be accessed, and sometimes require a further string parameter which is a keyword under which the required value has been saved. The second parameter is omitted in those functions which retrieve all the values of a certain kind from a theory (usually names ending in ‘s’, e.g. *get_binders*).

SML
```
get_aliases        : string -> (string * TERM) list;
get_ancestors      : string -> string list;
get_axiom          : string -> string -> THM;
get_axioms         : string -> (string list * THM) list;
get_binders        : string -> string list;
get_children       : string -> string list;
get_consts         : string -> TERM list;
get_defn           : string -> string -> THM;
get_defns          : string -> (string list * THM) list;
get_descendants    : string -> string list;
get_parents        : string -> string list;
get_thm            : string -> string -> THM;
get_thms           : string -> (string list * THM) list;
get_spec           : TERM -> THM;
```

To save things in the theory use declarations, definitions, specifications or paragraphs (see below), or *save_thm*.

You should now be able to do the exercises in section 13.2.

4.2 Declarations, Definitions and Specifications

4.2.1 *TYPE* constructors

An uninterpreted new *TYPE* constructor may be introduced using *new_type*, which requires to know only the name and arity of the new *TYPE* constructor.

SML
```
new_type           : string * int -> TYPE;
```

A *TYPE constructor* may be *defined* using *new_type_definition* by identifying a non-empty subset of an existing *TYPE* with which the new *TYPE* is required to be in one-one correspondence.

SML
```
new_type_defn      : string list * string * string list * THM -> THM;
```

The parameters to *new_type_definition* are:

1. A list of keywords under which the *TYPE* definition will be stored (in the current theory).
2. the name of the new *TYPE constructor* to be introduced
3. A list of the names of the *TYPE* variables present in definition of the property which determines the subset of the representation *TYPE* to be in one-one correspondence with the new *TYPE*.

   These *TYPE* variables will correspond to *TYPE* parameters to the newly introduced *TYPE constructor*, and their order determines the order in which the *TYPE* parameters must be supplied to the new *TYPE constructor*.
4. the theorem stating that the set determined by the defining property is non-empty

TYPE abbreviations may also be introduced using:

```
SML
|declare_type_abbrev   : string * string list * TYPE -> unit;
```

Where the first name is the name of the *TYPE* abbreviation and the second is a list of *TYPE* variables occurring in the *TYPE* supplied as third parameter. These are effectively formal parameters to the definition. The actual parameters supplied when using the *TYPE* abbreviation will effectively be substituted for the formal parameters in the defining *TYPE*.

4.2.2 Constants

new_const enables a completely uninterpreted new constant to be introduced. This results in no definition of the constant.

The primitive way of introducing a constant is *simple_new_defn* which enables a constant to be introduced by identifying an existing *TERM* as the value for the new constant.

new_spec permits greater freedom in the form of the definition of a constant, but to ensure that the introduction is a conservative extension a prior proof is required that a value satisfying the definition already exists. A single call to *new_spec* can introduce several constants at once.

const_spec is a variant of *new_spec* which avoids the need to prove consistency prior to introducing the new constants. It will attempt a consistency proof itself using a consistency prover taken from the current proof context, and if it fails to complete the consistency the constants are nevertheless introduced, in a form which permits them to be used for their intended purpose only if the consistency proof completed later.

```
SML
|new_const         : string * TYPE -> TERM;
|simple_new_defn   : string list * string * TERM -> THM;
|new_spec          : string list * int * THM -> THM;
|const_spec        : string list * TERM list * TERM -> THM;
```

4.2.3 Product Specifications

Two special functions are provided for introducing new product *TYPE*s, both labelled and unlabelled. These introduce a new *TYPE* at the same time as a number of new constants.

```
SML
|unlabelled_product_spec
|      :      {tyi        : TYPE list,   tykey       : string,
|              tyname     : string,      tyvars      : TYPE list OPT,
|              conkeys    : string list, conname     : string}
|              -> THM;
|
|labelled_product_spec
|      :      {tykey      : string,      labels      : (string * TYPE) list,
|               tyname    : string,      tyvars      : TYPE list OPT,
|               conname   : string,      constkeys   : string list}
|               -> THM;
```

4.2.4 Lexical Declarations

Any identifier can be declared:

- **prefix**, **infix**, **postfix** (with a **priority**)
- a **binder** (like "∀" and "∃")

using the following **fixity declaration** procedures:

```
SML
│declare_prefix      : int * string -> unit;
│declare_infix       : int * string -> unit;
│declare_postfix     : int * string -> unit;
│declare_binder      : string -> unit;
```

Such a declaration affects all uses of the name including the use of the name as a variable, however the **lexical** status of names may vary from one theory to another, or the special fixity may be removed using:

```
SML
│declare_nonfix      : string -> unit;
```

The following procedure may be used to discover the lexical status of a name in the current theory.

```
SML
│get_fixity          : string -> Lex.FIXITY;
```

4.3 Paragraphs

Some declarations may be done without resort to the metalanguage. This facility enables specifications to be presented almost entirely in HOL without having to make use of standard ML as well throughout the specification documents.

The form of these **paragraphs** is similar to some of the paragraphs forms in the Z specification language [11], which is supported by ProofPower. These paragraphs, despite their similarity to Z are distinct from the similar Z paragraphs (which are also accepted by ProofPower). An introduction to ProofPower support for Z may be found in ProofPower *Z Tutorial* [17].

4.3.1 Constant Declaration Paragraphs

This is a special syntactic form used for invoking *const_spec*.

```
SML
│(open_theory "usr013" handle _ => (open_theory "hol"; new_theory "usr013"));
│set_pc "hol2";
│declare_postfix (200, "!");
```

A HOL constant specification (**HOLCONST**) paragraph is entered into a document or into ProofPower as follows:

```
│        ⓈHOLCONST
│        │ length : 'a LIST → ℕ
│        ├───────────────────────
│        │            length [] = 0
│        │∧ ∀ h t•    length (Cons h t) = length t + 1
│        │
│        ■
```

and this normally results in a printed form like this:

HOL Constant

```
│ length : 'a LIST → ℕ
├──────────────────────────────
│            length [] = 0
│∧ ∀ h t•    length (Cons h t) = length t + 1
│
```

This results in a new definition being entered into the *current theory* (see below).

Provided that the system succeeded in proving the consistency of the definition (which it did in this case) it can immediately be used as follows:

SML

```
│rewrite_conv[get_spec⌜length⌝] ⌜length [1;2;3;4;5]⌝;
```

ProofPower output

```
│val it = ⊢ length [1; 2; 3; 4; 5] = 5 : THM
```

4.3.2 Labelled Product Paragraphs

These provide an object language construct (**HOLLABPROD** paragraph) for introducing **labelled products** using *labelled_product_spec*.

The entry in the source document, which may be read or pasted into ProofPoweris:

```
│        ⓈHOLLABPROD Date────────
│        │      day month year:ℕ
│        ■───────────────────────
```

Which is printed as:

HOL Labelled Product

```
┌─Date──────────────────────────────────
│      day month year:ℕ
│
└───────────────────────────────────────
```

The following definitions result:

SML

```
│print_theory "usr013";
```

ProofPower output

```
 === The theory usr013 ===

--- Parents ---

                cache'hol       hol

--- Constants ---

length          'a LIST → ℕ
year            Date → ℕ
month           Date → ℕ
day             Date → ℕ
MkDate                  ℕ → ℕ → ℕ → Date

--- Types ---

Date

--- Fixity ---

Postfix 200:    !

--- Definitions ---

length          ⊢ length [] = 0
                    ∧ (∀ h t• length (Cons h t) = length t + 1)
Date            ⊢ ∃ f• TypeDefn (λ x• T) f
MkDate
day
month
year            ⊢ ∀ t x1 x2 x3
                  • day (MkDate x1 x2 x3) = x1
                      ∧ month (MkDate x1 x2 x3) = x2
                      ∧ year (MkDate x1 x2 x3) = x3
                      ∧ MkDate (day t) (month t) (year t) = t

 === End of listing of theory usr013 ===
```

You should now be able to attempt the exercises in section 13.3.

FORWARD PROOF

5.1 Introduction

ProofPower follows the **LCF paradigm** [5], in which an abstract data type implemented in a functional programming language (known as the **metalanguage**, in this case Standard ML) guarantees that a protected type (*THM* in this case) includes only values which have been obtained by computations which precisely correspond to proofs in the object language (ProofPower-HOL).

ProofPower keeps track of which theorems have been proven and in what context, so that the user can always establish whether a theorem is valid and if so, from which axioms and definitions it has been derived. ProofPower does not keep track of how a theorem has been derived, once it has fully checked the validity of the derivation. Users of ProofPower will usually keep a record of how they have derived their theorems in a document (often referred to as a 'proof script'). ProofPower does provide some support for the preparation of such proof scripts and for their printing and processing, interactively or in batch.

A **theorem** in ProofPower is a value of type **THM** computed from *axiom*s and *definition*s using *rule*s and *conversion*s. An **axiom** is a theorem introduced without proof, and recorded as such in the theory hierarchy. A **definition** is a special kind of axiom introduced by "conservative" mechanisms. A **rule** is a function which computes theorems. Rules may be logically **primitive rule**s, in which case the form part of the primitive abstract logic, **kernel rule**s in which case they are among the rules implemented directly in the logical kernel (which includes all the *primitive rules*), or they may be **derived rule**s, in which case they are implemented as standard ML functions which compute the required results using the *kernel rules*. A **conversion** is a special kind of rule which proves *THM*s which are equations. A conversion takes a *TERM* argument and (if successful) returns a *THM* whose conclusion is an equation with the same *TERM* on the left hand side of the equation. The idea of *derived rule*s as computations is due to Robin Milner. *conversion*s, a very important special case of *derived rule*s which form the basis for equational reasoning, were invented by Larry Paulson [7].

5.2 Theorems

The *HOL* logic is a **sequent calculus**. A **sequent** is a value in Standard ML having an ML type named either **SEQ** or **GOAL** defined as: (TERM list) * TERM. Each *TERM* in a *sequent* must have *HOL TYPE* ⌜:BOOL⌝. In such a *sequent* the list of TERMs on the left are known as **assumptions** or **asms** while the single *TERM* on the right is the known as the **conclusion** or **concl** of the *sequent*.

A *sequent* may be assigned a value in the context of a *HOL* **interpretation** and an assignment to the free variables occurring in the *sequent*. This value is defined using the rules for evaluation of *TERM*s (see section 3.6), which in the case of boolean *TERM*s will always assign either the value ⌜T⌝ or ⌜F⌝. The value assigned to the *sequent* will be ⌜T⌝ if the value assigned to the conclusion of the *sequent* is ⌜T⌝, or if the value assigned to any one of the assumptions is ⌜F⌝.

A *sequent* is **satisfied** by an *interpretation* if it evaluates to ⌜T⌝ in the context of that *interpretation* for every *well typed* assignment of values to the free variables in the *sequent*. A *sequent* is **entailed** by a set of axioms and definitions if it is satisfied by every model which satisfies all of the axioms and definitions.

The logic supported by ProofPower enables only those *sequents* to be proven in the context of any collection of axioms and definitions which are *entailed* by those axioms and definitions. In the case that the only axioms in context are the five primitive axioms of the HOL logic the sequent '([],⌜F⌝)' is not provable, no matter what collection of definitions are in context. (ProofPower will reject any attempt to enter a definition which would enable the sequent to be proven).

A theorem corresponds to a *sequent* which has been derived from axioms and definitions using the rules of the logic. Theorems are tagged with an indicator of the context in which they were derived (and because of this extra information are not identical with the corresponding sequent).

The *sequent* part of a theorem may be accessed using the following ML functions:

```
SML
│  dest_thm   : THM -> SEQ;
│  asms       : THM -> TERM list;
│  concl      : THM -> TERM;
```

dest_thm returns the complete *sequent* corresponding to a value of type *THM*, while *asms* and *concl* return the left hand part of the the *sequent* (the assumptions) and the right hand part (the conclusion) respectively.

No constructor is available which simply constructs a theorem from a *sequent*, since this may only be done by proof. The closest function to achieving this is:

```
SML
│  new_axiom  : (string list * TERM) -> THM;
```

which, though returning the required theorem, does so only after this has been recorded as an axiom in the current theory. The strings supplied are keywords against which the theorem is stored in the theory, and may be used subsequently to retrieve the theorem from the theory.

Theorems are displayed without 'Quine corners' ('⌜',' ⌝'). Unlike *TERM*s they cannot be parsed, they must be proven (or introduced as axioms).

5.3 The Primitive Logic

The primitive HOL logic can be described abstractly in very simple terms.

There are three primitive *TYPE* constructors:

- **BOOL**

  A 0-ary *TYPE* constructor which denotes a set containing the two truth values (⌜T⌝ and ⌜F⌝).

- **IND**

  A 0-ary *TYPE* constructor which denotes an infinite set of individuals.

- $\rightarrow$

  A 2-ary *TYPE* constructor known as the 'function space constructor' denoting the function which, given two sets (the *domain* and the *codomain*), returns the set of all functions which are total over the domain and have values in the codomain.

The primitive constants are:

- =

  the polymorphic curried infix equality function.

- ⇒

  The curried infix BOOLean operator which denotes material implication.

- ϵ

  Which denotes a polymorphic choice function

The primitive *TYPE* constructors and constants are introduced in theory **min**. A number of additional constants are defined in terms of these primitives in theory **log**, and then the five axioms of the HOL logic are introduced in theory **init**.

There are seven *primitive rules* which complete the primitive logic.

5.4 The Logical Kernel

The **Logical Kernel** of ProofPower is that part of ProofPower which is critical to the checking of proofs and the soundness of the implemented logical system.

The logical kernel implements the primitive logic and also a small number of non-primitive rules.

Features which are not in the primitive logic are included in the logical kernel for two main reasons.

Firstly, support for certain literals is in practice essential, but not directly addressed in the primitive logical basis. The literals supported at present are numeric literals, character literals, and string literals. Literals provide convenient concrete syntax for infinite families of constants. These constants in all cases could be introduced logically by the use of definitions were it not that an infinite number of definitions would be required. The literals are therefore treated as built in constants the definitions of which are obtained from conversions provided in the logical kernel. The *TYPE*s ⌜:ℕ⌝, ⌜:CHAR⌝ and ⌜:CHAR LIST⌝ of these literals are therefore in the pervasive theories.

Secondly, efficiency in computing and checking proofs is greatly improved if a small number of inference rules which could be implemented as derived rules are in fact implemented directly in the logical kernel.

In addition to these rules, the mechanisms for undertaking conservative extensions ought strictly also be considered to be part of the logic, since flaws in their definition or implemetation might result in the logical system being rendered inconsistent.

Closely associated with the logical kernel are the set of **pervasive theories**. These theories introduce the primitive *TYPE constructor*s and constants, and a number of definitions of other *TYPE constructor*s and constants. The theories contain all the definitions on which the soundness of the rules implemented in the logical kernel depend. The pervasive theories are *basic_hol* and its ancestors. No new theory may be introduced which does not have *basic_hol* as an ancestor.

5.5 Naming Conventions for Theorems and Rules

Certain naming conventions are useful in permitting theorems and rules to be located in the reference documentation (with the assistance of the KWIC index).

- **_axiom**

  Names ending with *_axiom* are used for axioms or for functions (e.g. *new_axiom*) which introduce or access axioms.

- **_def _spec**

  Name suffixes used for definitions or specifications.

- **_thm _clauses**

  Name suffixes for theorems. Many theorems are conjuncts of several useful results, and in these cases the suffix *_clauses* is used in the theorem name.

- **_rule _elim _intro**

  Names ending in *_elim* are usually inference rules which eliminate some HOL construct (e.g. a conjunction). Names ending in *_intro* are inference rules which *introduce* the relevant construct. Other rules will normally end with *_rule*.

- **_conv**

  Names ending in *_conv* are conversions, i.e. rules having type *TERM* -> *THM* where the *THM* is an equation (or bi-implication) with the *TERM* as its left hand operand.

5.6 A Selection of Useful Rules

5.6.1 Assume Rule

The ML function **asm_rule** implements the primitive rule corresponding in a *sequent* calculus to making an assumption in a natural deduction proof.

Given any BOOLean *TERM*, *asm_rule* will return the theorem which has that *TERM* both as its sole assumption and as its conclusion.

SML
```
val thm1 = asm_rule ⌜∀x y:ℕ• x*y > 0⌝;
```

ProofPower Output
```
val thm1 = ∀ x y• x * y > 0
                ⊢ ∀ x y• x * y > 0 : THM
```

5.6.2 Modus Ponens

The primitive rule often known as **modus ponens**, whose name in ProofPower is **⇒_elim**, enables a theorem to be deduced once it has been proven that it is implied by some other proven fact.

SML
```
val thm_a = asm_rule ⌜a:BOOL⌝;
val thm_b = asm_rule ⌜a⇒b⌝;
```

ProofPower Output
```
val thm_a = a ⊢ a : THM
val thm_b = a ⇒ b ⊢ a ⇒ b : THM
```

SML
```
val thm_c = ⇒_elim thm_b thm_a;
```

ProofPower Output
```
val thm_c = a ⇒ b, a ⊢ b : THM
```

5.6.3 Specialisation

The specialisation of a universal result to a particular case in performed by the rule **∀_elim**.

The value to be used for specialisation is supplied as a *TERM* in the first parameter, and must be consistent with the *TYPE* of the quantified variable.

SML
```
val thm2 = ∀_elim ⌜455⌝ thm1;
```

ProofPower Output
```
val thm2 = ∀ x y• x ∗ y > 0 ⊢ ∀ y• 455 ∗ y > 0 : THM
```

5.6.4 Multiple Specialisation

Several nested universal quantifications can be specialised at once using **list_∀_elim**. In this case the values to be used for specialisation are supplied as a list of *TERM*s.

SML
```
val thm3 = list_∀_elim [⌜455⌝,⌜0⌝] thm1;
```

ProofPower Output
```
val thm3 = ∀ x y• x ∗ y > 0 ⊢ 455 ∗ 0 > 0 : THM
```

5.6.5 Removing Outermost Universals

A special case of the above is the case where it is required to specialise all universals to the free variable having the same name as the variable used for quantification. **all_∀_elim** will accomplish this specialisation wihout need of any parameters.

SML
```
val thm4 = all_∀_elim thm1;
```

ProofPower Output
```
val thm4 = ∀ x y• x ∗ y > 0 ⊢ x ∗ y > 0 : THM
```

5.6.6 Splitting Conjunctions

From a theorem whose conclusion is a conjunction, a list of theorems can be derived. The conclusions of these theorems are the individual conjuncts of the original theorem, and the assumptions are the same as those in the original theorem.

In the context of the binding:

SML
```
plus_order_thm;
```

ProofPower output
```
val it = ⊢ ∀ i m n
  • m + i = i + m ∧ (i + m) + n = i + m + n ∧ m + i + n = i + m + n : THM
```

plus_order_thm can be broken apart by using **strip_∧_rule** as follows:

SML
```
val thm5 = all_∀_elim plus_order_thm;
```

ProofPower output
```
val thm5 = ⊢ m + i = i + m
         ∧ (i + m) + n = i + m + n
         ∧ m + i + n = i + m + n : THM
```

SML
```
val thms1 = strip_∧_rule thm5;
```

ProofPower output
```
val thms1 = [⊢ m + i = i + m,
         ⊢ (i + m) + n = i + m + n,
         ⊢ m + i + n = i + m + n] : THM list
```

5.6.7 Adding Universals

A theorem may be *closed* by universally quantifying over all variables which have free occurrences in the conclusion of the theorem but not in any of its assumptions. **all_∀_intro** accomplishes this task.

The ML function **nth** selects the nth element from a list.

SML
```
nth 2 thms1;
```

ProofPower output
```
val it = ⊢ m + i + n = i + m + n : THM
```

SML
```
val thm6 = all_∀_intro (nth 2 thms1);
```

ProofPower output
```
val thm6 = ⊢ ∀ m i n• m + i + n = i + m + n : THM
```

If the quantifiers are required in some specific order **list_∀_intro** should be used and supplied with the list of variables over which universal quantification is desired.

SML
```
val thm7 = list_∀_intro [⌜i⌝,⌜m⌝,⌜n⌝] (nth 2 thms1);
```

ProofPower output
```
val thm7 = ⊢ ∀ i m n• m + i + n = i + m + n : THM
```

You should now be able to attempt the exercises in section 13.4.

GOAL ORIENTED PROOF

6.1 Introduction

Direct forward proof of non-trivial results is usually a complex and difficult process.

In practice, finding such a proof will usually begin with a conjecture thought to be true of which a proof is sought, and will proceed by working backwards from this conjecture to more elementary results from which it can be derived until the conjecture is seen to be derivable from axioms or previously proven theorems.

A 'subgoal package' is available in ProofPower to assist in this process, and this is normally used in constructing all but the very simplest proofs.

The subgoal package is based on the notion of a *TACTIC*. A *TACTIC*, given a *GOAL* which the user wishes to prove, will determine one or more **subgoal**s from which the *GOAL* is derivable and return the list of *subgoal*s and a function, known as a *PROOF* which is able to prove the original *GOAL* from theorems corresponding to the *subgoal*s which the *TACTIC* has chosen. Normally a tactic would be expected to deliver a set of subgoals which are easier to prove than the original *GOAL*, and in this way the proof is progressed until *subgoal*s are reduced to the level at which *TACTIC*s can be found which are able to prove the *subgoal*s from the empty set of *subgoal*s.

The subgoal package helps the user to manage the development of a proof using *TACTIC*s by keeping track of the outstanding subgoals and composing together the fragments of proof delivered by the various *TACTIC*s applied during the development of the proof.

- a *GOAL*,

  is just a sequent, viz:

  - a list of assumptions (boolean *TERM*s)
  - a conclusion (also a boolean *TERM*)

  *GOAL = TERM list * TERM = SEQ*

- a **PROOF**,

  is a function which computes a theorem from a list of theorems.

  PROOF = THM list -> THM

- a *TACTIC*,

  is a function which:

  - takes a *GOAL*
  - returns
    - a list of *subgoal*s
    - a *PROOF* which will compute a theorem corresponding to ("achieving") the input goal from theorems corresponding to the *subgoal*s.

*TACTIC = GOAL -> (GOAL list * PROOF)*

6.2 Using the Subgoal Package

6.2.1 Getting Started

A proof using the subgoal package is initiated by supplying to the subgoal package the *GOAL* to be proven.

The following three functions are available for this purpose:

```
SML
|set_goal                 : GOAL -> unit;
|push_goal                : GOAL -> unit;
|push_consistency_goal    : TERM -> unit;
```

set_goal is the function most frequently used. If the subgoal package was already in use this will replace the current GOAL by the newly supplied goal and the proof in progress will be aborted.

In the case that a proof is in progress and the user wishes to return to that proof after completing the proof he is about to start, then he should use *push_goal.* In this case the old proof will be resurrected when the new proof has been completed.

push_consistency_goal is used when the consistency of a HOL constant specification has not been proven automatically when the constant was introduced and the user wishes to complete the proof himself. In this case a HOL *TERM* is supplied as parameter to the procedure. This TERM must consist of the constant the consistency of whose specification is to be proven, and the goal will then be set appropriately.

6.2.2 Doing the Proof

Only a small number of subgoal package procedures are needed to conduct the proof.

```
SML
|apply_tactic        : TACTIC -> unit;
|a                   : TACTIC -> unit;
|undo                : int -> unit;
|set_labelled_goal   : string -> unit;
|lemma_tac           : TERM -> TACTIC;
```

apply_tactic, which may be abbreviated simply as *a*, may be used to progress the proof by applying a tactic to the current goal. The user selects the tactic to be applied and supplies this as a parameter to *apply_tactic*. The tactic will then be applied to the current subgoal, and if it is successful the resulting subgoals will be displayed. If the application of the tactic is successful but no new subgoals result, then the tactic has completed the proof of the original subgoal. Another outstanding subgoal will be selected and displayed. If all subgoals have been proven the user will be informed that his proof is complete.

If, on seeing the effect of his selected tactic, the user decides that he would rather do something else, then he may use *undo* to reverse the effects of the tactic application. The user may step back through several steps, the number of steps remembered by the subgoal package for this purpose being determined by a system control parameter.

At each step in the development of the proof, after applying a tactic to the current subgoal, the subgoal package will select a subgoal as the *current* subgoal. If the user wishes to prove the outstanding subgoals in an order differing from that selected by the subgoal package then he may select a subgoal for processing by using *set_labelled_goal*. This function requires a string parameter which is the label assigned by the subgoal package to the goal when it was first introduced. This is displayed as a string suitable for input by cut and paste to *set_labelled_goal* when the goal is displayed.

In many cases the best thing to prove next is in fact not one of the outstanding subgoals but some other result from which the current subgoal could be proven. For this purpose no special functionality in the subgoal package is required, use of a *TACTIC* called *lemma_tac* will do the job. *lemma_tac* should be used to commence proof of a result which can be proven from the list of assumptions on the current subgoal, and which will facilitate the proof of the conclusion of the current subgoal. The tactic is used by supplying the new conclusion as a *TERM* parameter to *lemma_tac*. *lemma_tac* will then (usually) create two new subgoals. The first is the subgoal with the same assumptions as the current subgoal, but the new conclusion as supplied to *lemma_tac*. The second subgoal is the original subgoal with the new result stripped into the assumptions. The stripping process may cause further case splits, in which case more than two subgoals may arise, or it may even cause the second subgoal to be discharged, in which case only the one subgoal will result from the application of *lemma_tac*.

lemma_tac would not normally be used where the result to be proven is of general interest beyond the current proof. In such a case the general result would be set up as a new main goal by using *push_goal*. After completion of this subsidiary proof the result would be stored in a suitable theory or bound to an ML name. The original proof can then be resumed making use of the result thus obtained.

6.2.3 Finishing Off

When the subgoal package declares that the proof is complete the user has the following options:

```
SML
|top_thm          : unit -> THM;
|pop_thm          : unit -> THM;
|save_pop_thm     : string -> THM;
```

top_thm may be used to retrieve the theorem just proven, returning this as a value but making no other changes.

pop_thm returns the theorem just proven, without retaining the theorem in the subgoal package. If the proof was started with a *push* the previous proof will now be resumed by displaying the current subgoal in that proof.

Often the user will wish to save the theorem in his current theory, in which case *save_pop_thm* may be used. A single string is supplied as a keyword for retrieving the theorem from the theory, and the value of the theorem is returned as well as being stored (it is usually convenient to bind the theorem to an ML name at the same time).

also note:

```
SML
|drop_main_goal          : unit -> GOAL;
|save_thm                : string * THM -> THM;
|list_save_thm           : string list * THM -> THM;
|save_consistency_thm    : TERM -> THM -> THM;
```

drop_main_goal may be used to abandon a proof attempt (possibly because it has become clear that the original *GOAL* is unprovable). If there are other proofs stacked, the most recent will be resumed. *repeat drop_main_goal* may be used to discard all the goals on the main goal stack.

save_thm and *list_save_thm* may be used to save theorems which have been proven without using the subgoal package. *list_save_thm* allows the theorem to be saved under more than one key (and may be used for this, with *pop_thm* to save the result of a subgoal package proof under multiple keys).

save_consistency_thm should be used to save the result of a subgoal proof where the proof was initiated using *push_consistency_goal*. It does however require the theorem to be saved to be submitted as a parameter, and is therefore normally used with *pop_thm*. The *TERM* parameter must be one of the constants whose specification has been proven consistent.

PREDICATE CALCULUS

This chapter considers the three most important facilities in ProofPower for reasoning essentially within the first order predicate calculus with equality, and a number of miscellaneous tactics concerned with the predicate calculus.

The core functionality of each of these facilities is within the predicate calculus, but they all include context sensitive pre- and post-processing which may be set up to incorporate knowledge of broader areas.

The facilities are:

- *stripping*

  The facilities in this group are frequently used, being invoked continually during most proofs, and are almost entirely determined by context. Stripping is the principal technique for dealing with propositional connectives and for this reason is the source of most of the case splitting which occurs in ProofPower proofs.

- *forward chaining*

  Forward chaining is in most applications the most convenient way of instantiating generalised implications which are among the assumptions of the current goal, or among the theorems previous proven.

- *rewriting*

  The rewriting facilities provide support for using equations to transform *theorem*s or *GOAL*s.

7.1 Stripping

The **stripping** facilities, most frequently invoked using **strip_tac**, provide systematic ways of simplifying both the conclusion of a goal and assumptions introduced during a proof.

Stripping the conclusion is the primary purpose of *strip_tac*, one invocation of which will perform one step of transformation on the conclusion.

New assumptions are commonly created by the stripping of a conclusion which is an implication, but forward inference from the assumptions also results in the creation of new assumptions. The default action on creating a new assumption is to completely strip the assumption before adding it to the current assumptions.

Most of the effects created when stripping either conclusions or assumptions are achieved by the application of equational transformations to the existing conclusion or the new assumption. All such changes are configurable, and the set of such transformations in force at any time is determined by the current *proof context*. The small set of actions of the stripping facilities which are not configurable, are just those effects necessary to make stripping capable of solving propositional tautologies and to do the few completely uncontroversial quantifier rules (essentially ‘skolemisation’ of quantifiers).

To understand the behaviour of the stripping facilities (which is not necessary to use them successfully), you need to understand what the built in actions are, and what conversions are applied in any particular context. In this section we are concerned only with those configurable actions which are concerned with reasoning in the predicate calculus. Almost all proof contexts will include these actions as well as further actions relating to the non-predicate calculus reasoning which is to be exploited.

In the following sections we therefore describe the built in actions on stripping conclusions and assumptions, and then the conversions normally in place which relate to the predicate calculus.

- "stripping" facilities incorporate automatic propositional reasoning and enable domain specific knowledge to be invoked automatically during proof.
- *strip_tac* processes the conclusion of the current goal
- When new assumptions are created, by *strip_tac* or otherwise, they are normally stripped before being entered into the assumption list.
- Stripping of assumptions is different from stripping of conclusions.

7.1.1 Stripping Conclusions

There are three logical constructs whose behaviour is built in, because the required behaviour is more complex than simply transforming the conclusion itself.

- conjunctions

  are built in because the stripping of a conjunction is required to produce a split into two subgoals.
- implications

  are built in because stripping an implication causes the left hand side of the implication to be made into a new assumption.
- universals

  are built in because, though the processing of a universal results in its elimination by skolemisation, this effect is not obtainable by an equational transformation (because the new conclusion is not logically equivalent to the old).
- discharge

  finally, a number of checks are built in which may result in the discharge of the subgoal

The effect of stripping the conclusion of the current goal is therefore as follows:

1. apply conclusion stripping conversions from proof context
2. if no conversion applies then attempt one of the following:

   (a) :

   ```
   .. ?⊢ ∀x• P x ===> .. ?⊢ P x'
   ```

(b) :

| $.. ?\vdash P1 \wedge P2 ===>$
| $\quad .. ?\vdash P1$ *and* $.. ?\vdash P2$ (*two subgoals*)

(c) :

| $.. ?\vdash P1 \Rightarrow P2 ===>$
| $\quad strip\_asm\_tac(P1), .. ?\vdash P2$

3. then check if:
   (a) conclusion of the goal is ⌜T⌝
   (b) conclusion is in the assumptions

   if so, prove the result

The following transformations on conclusions concerning the predicate calculus, though not built in to the action of the stripping facilities, are present in almost all proof contexts:

- equivalence statements are transformed to conjunctions
- disjunctions are transformed into implications
- negations are pushed in over any logical construct using de-Morgans laws and double negations are cancelled

7.1.2 Stripping Assumptions

Like the stripping of conclusions, the stripping of assumptions has a number of actions built in, as well as applying transformations from the current proof context. Unlike the stripping of conclusions, the norm is to completely strip an assumption prior to adding it into the assumptions, rather than to transform assumptions one step at a time.

In summary the built in effects of assumption stripping are:

- conjunctions

  are broken in to two parts each of which is treated as a separate assumption
- disjunctions

  give rise to case splits, each resulting subgoal having just one side of the disjunction stripped into its assumptions
- existentials

  are eliminated by skolemisation
- checks

  a set of checks (not the same as the ones used when stripping conclusions) is made which may result in discharge of the subgoal

More precisely the algorithm is as follows:

1. Repeat the following transformations until no further changes occur:
   (a) : apply assumption stripping conversions from proof context
   (b) :

   | $\exists x\bullet\ P\ x \vdash ?\ ..\ ===>\ P\ x' \vdash ?\ ..$

   (c) :

   | $P1 \lor P2 \vdash ?\ ..\ ===>$
   | $\quad P1 \vdash ?\ ..\ \textit{and}\ P2 \vdash ?\ ..\ (\textit{two subgoals})$

   (d) :

   | $P1 \land P2 \vdash ?\ ..\ ===>$
   | $\quad P1,\ P2 \vdash ?\ ..\ (\textit{two assumptions})$

2. then for each resulting assumption, check if:
   (a) assumption = ⌜F⌝
   (b) assumption = concl
   (c) contradicts an existing assumption

   if so, prove the result.
3. also check if:
   (a) assumption = ⌜T⌝
   (b) is same as an existing assumption

   if so, discard the assumption.

Configured transformations for assumptions concerning the predicate caculus are:

- equivalence statements are transformed to conjunctions
- implications are transformed to disjunctions
- negations are pushed in over any logical construct using demorgans laws and double negations are cancelled

You should now be able to attempt the exercises in section 13.7.

7.2 Rewriting

A large amount of the power of ProofPower comes from its general purpose **rewriting** facilities. These are primarily used to transform the conclusion of the current goal using universal equations which are automatically instantiated to values which enable the conclusion to be rewritten (where possible). The facility is more general and powerful than can be achieved simply by application of a finite set of generalised equations. Arbitary algorithms which yield equational results can be invoked during the rewriting process, permitting simplifications which would not otherwise have been possible. For example, in most contexts rewriting peforms beta-reduction automatically whenever an application of a lambda-abstraction is found, even though beta-reduction is a rule which cannot be expressed as a theorem even in higher order logic (other than by stepping up into the metalanguage).

Because the reasoning involved in rewriting is equational it is applicable in several different ways, hence the rewriting facilities are available not only as *TACTIC*s but also as *conv*ersions and *rule*s. In addition, the facilities provide for two distinct strategies for traversing the *TERM* under transformation, (the occurrence of *once* in the name of the function indicating a strategy intended to prevent looping). Finally, the clauses which are used in the rewriting process will normally include a set of defaults determined by the current proof context to be applied, in addition to the clauses derived from theorems supplied as parameters to the function.

$$[\textbf{pure\_}][\textbf{once\_}][\textbf{asm\_}]\textbf{rewrite\_}\begin{cases} conv \\ rule \\ tac \\ thm\_tac \end{cases}$$

$$: \textbf{THM list}- > \begin{cases} conv(= TERM- > THM) \\ THM- > THM \\ TACTIC \end{cases}$$

$$: \textbf{THM}- > \textbf{TACTIC}$$

rewrites the term, theorem or goal using:

- conversions in *proof context* (unless *pure*)
- assumptions (if *asm* but not *conv*) (after context sensitive pre-processing)
- theorems in *THM list* (or *THM*) parameter (after context sensitive pre-processing)

Rewriting continues until no more rewrites are possible (unless *once*).

You should now be able to attempt the exercises in sections 13.5 and 13.6.

7.3 Forward Chaining

Facilities for **forward chaining** consist of a group of tactics for reasoning forward from the assumptions. These are based on a rule, **fc_rule**, which uses a list of implications to generate a list of new theorems from a list of "seed" theorems. The arguments to *fc_rule* are two lists:

- *Implications*

  a list of theorems which are implications (possibly universally quantified), e.g.:

  | $[\Gamma_1 \vdash \forall x1\ x2\ ...\bullet A_1 \Rightarrow B_1, ...]$

- *Seeds*

  a list of theorems to be matched against the antecedents of the implications, e.g:

  | $[\Gamma_1 \vdash c_1, ...]$

For each implication, $\vdash \forall x1\ x2\ ...\bullet A \Rightarrow B$ and for each seed $\vdash c$, *fc_rule* determines whether A can be specialised to give c and if so it includes the corresponding specialisation of B in its result.

For example:

SML
```
(fc_rule : THM list -> THM list -> THM list)

        [asm_rule     ⌜∀x•x > 10 ⇒ P x⌝,
         asm_rule     ⌜∀y•y < 10 ⇒ Q y⌝]

        [prove_rule [] ⌜101 > 10⌝,
         prove_rule [] ⌜4 < 10⌝];
```

ProofPower Output
```
val it = [∀ y• y < 10 ⇒ Q y ⊢ Q 4,
       ∀ x• x > 10 ⇒ P x ⊢ P 101] : THM list
```

In practice, rather than *fc_rule* one of the forward chaining tactics is more likely to be used.

The forward chaining *TACTIC*s are:

[all_][asm_]fc_tac

[all_][asm_]forward_chain_tac

All these functions have type:

THM list − > TACTIC

The *asm_* variants take the implications from their the argument together with the assumptions. Other variants just use list given as argument as implications. In all cases the seeds are the assumptions.

The variants without *all_* take one pass over the seeds for each implication. Variants with *all_* add any new implications to the list of implications and loop until no new results can be generated.

New theorems deduced by these tactics are stripped into the assumptions. The *all_* variants only strip in theorems which are not themselves implications.

For example:

SML
```
set_goal([], ⌜∀a b c d•a ≤ b ∧ b ≤ c ∧ c ≤ d ⇒ a ≤ d⌝);
a(REPEAT strip_tac);
```

ProofPower Output
```
(∗  3 ∗)  ⌜a ≤ b⌝
(∗  2 ∗)  ⌜b ≤ c⌝
(∗  1 ∗)  ⌜c ≤ d⌝

(∗ ?⊢ ∗)  ⌜a ≤ d⌝
```

SML
```
a(fc_tac[≤_trans_thm]);
```

ProofPower Output
```
(∗ 6 ∗) ⌜a ≤ b⌝
(∗ 5 ∗) ⌜b ≤ c⌝
(∗ 4 ∗) ⌜c ≤ d⌝
(∗ 3 ∗) ⌜∀ n• b ≤ n ⇒ a ≤ n⌝
(∗ 2 ∗) ⌜∀ n• c ≤ n ⇒ b ≤ n⌝
(∗ 1 ∗) ⌜∀ n• d ≤ n ⇒ c ≤ n⌝

(∗ ?⊢ ∗) ⌜a ≤ d⌝
```

SML
```
a(all_asm_fc_tac[] THEN all_asm_fc_tac[]);
```

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

Many useful properties are naturally formulated as universally quantified implications:

≤_trans_thm	⊢ ∀ m i n• m ≤ i ∧ i ≤ n ⇒ m ≤ n
less_trans_thm	⊢ ∀ m i n• m < i ∧ i < n ⇒ m < n
mod_less_thm	⊢ ∀ m n• 0 < n ⇒ m Mod n < n

Forward chaining saves having to specialise such facts explicitly.

A function, **fc_canon**, is used by the forward chaining *TACTIC*s to generate implications from the arguments to the forward chaining.

For example the theorems:

```
⊢ (A ∧ B) ∨ C
⊢ ∀ m i n• m ≤ i ∧ i ≤ n ⇒ m ≤ n
```

are treated as:

```
⊢ ¬ B ⇒ ¬ C ⇒ F
⊢ ¬ A ⇒ ¬ C ⇒ F
⊢ ∀ n i m• m ≤ i ⇒ i ≤ n ⇒ ¬ m ≤ n ⇒ F
```

The '⇒*F*' part produced by *fc_canon* is simplified away when the new theorem is stripped into the assumptions.

The new theorems stripped into the assumptions are made as general as possible by universally quantifying them over any free variables which do not appear in the goal.

7.4 More Predicate Calculus Tactics

7.4.1 strip_asm_tac

- **strip_asm_tac** strips a theorem into the assumptions in the same way that *strip_tac* adds new assumptions

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{strip\ \ c,\ \Gamma\ \}\ t} \qquad \begin{array}{l} strip\_asm\_tac \\ (\vdash c) \end{array}$$

- a *case split* results if the conclusion of the theorem is a disjunction
- names ending in *_cases_thm* indicate theorems designed for use with *strip_asm_tac* for case splits:

ℕ_cases_thm	$\vdash \forall\ m\bullet\ m = 0 \lor (\exists\ i\bullet\ m = i + 1)$
less_cases_thm	$\vdash \forall\ m\ n\bullet\ m < n \lor m = n \lor n < m$

- use [*list_*]∀*_elim* to specialise the *_cases_thm*

strip_asm_tac: example

SML
```
set_goal([], ⌜(if x = 0 then 1 else x) > 0⌝);
```

SML
```
∀_elim ⌜x⌝ ℕ_cases_thm;
```

ProofPower Output
```
val it = ⊢ x = 0 ∨ (∃ i• x = i + 1) : THM
```

SML
```
a(strip_asm_tac(∀_elim ⌜x⌝ ℕ_cases_thm));
```

ProofPower Output
```
(* *** Goal "2" *** *)

(*  1 *)  ⌜x = i + 1⌝

(* ?⊢ *)  ⌜(if x = 0 then 1 else x) > 0⌝

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

(*  1 *)  ⌜x = 0⌝

(* ?⊢ *) ⌜(if x = 0 then 1 else x) > 0⌝
```

7.4.2 cases_tac

- **cases_tac** ⌜c⌝ lets you reason by cases according to whether a chosen condition ⌜c⌝ is true or false:

Tactic

$$\frac{\{\ \Gamma\ \}\ t}{\{strip\ c,\ \Gamma\ \}\ t;\quad \{strip\ \neg c,\ \Gamma\ \}\ t}\qquad cases\_tac\ ⌜c{:}BOOL⌝$$

```
cases_tac ⌜c:BOOL⌝
```

- is effectively the same as:

```
strip_asm_tac(∀_elim ⌜c:BOOL⌝ (prove_rule [] ⌜∀b•b ∨ ¬b⌝));
```

  but it's less to type and quicker.

cases_tac: example

SML

```
set_goal([], ⌜(if x < y + 1 then x else y) < y + 1⌝);
```

SML

```
a(cases_tac ⌜x < y + 1⌝);
```

ProofPower Output

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

(*  1 *)  ⌜¬ x < y + 1⌝

(* ?⊢ *)  ⌜(if x < y + 1 then x else y) < y + 1⌝

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

(*  1 *)  ⌜x < y + 1⌝

(* ?⊢ *)  ⌜(if x < y + 1 then x else y) < y + 1⌝
```

7.4.3 swap_asm_concl_tac

- **swap_asm_concl_tac** lets you interchange (the negations) of an assumption and a conclusion

Tactic

$$\frac{\{\ \Gamma,\ t1\ \}\ t2}{\{strip\ \neg t2,\ \Gamma\ \}\ \neg t1}\qquad swap\_asm\_concl\_tac\ ⌜t1⌝$$

- Often used to rewrite one assumption with another
- Also useful when the conclusion is a negated equation

swap_asm_concl_tac: example

SML
```
set_goal([], ⌜(∀x y•x ≤ y ⇒ P(x,y)) ∧ x = y ⇒ P(x,y)⌝);
a(strip_tac);
```

ProofPower Output
```
(*  2 *)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(*  1 *)  ⌜x = y⌝

(* ?⊢ *)  ⌜P (x, y)⌝
```

ProofPower Output

SML
```
a(list_spec_nth_asm_tac 2 [⌜x⌝, ⌜y⌝]);
```

ProofPower Output
```
(*  3 *)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(*  2 *)  ⌜x = y⌝
(*  1 *)  ⌜¬ x ≤ y⌝

(* ?⊢ *)  ⌜P (x, y)⌝
```

SML
```
a(swap_asm_concl_tac ⌜¬ x ≤ y⌝);
```

ProofPower Output
```
(*  3 *)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(*  2 *)  ⌜x = y⌝
(*  1 *)  ⌜¬ P (x, y)⌝

(* ?⊢ *)  ⌜x ≤ y⌝
```

7.4.4 lemma_tac

- **lemma_tac** lets you state and prove an 'in-line' lemma

Tactic

$$\frac{\{\ \Gamma\ \}\ conc}{\{\ \Gamma\ \}\ lemma;\quad \{strip\ lemma,\ \Gamma\ \}\ conc} \qquad lemma\_tac\ \ulcorner lemma \urcorner$$

- Gives a more natural feel to many proofs
- If just one tactic will prove the lemma then **THEN1** is a convenient way of applying it
- *tac1 THEN1 tac2* first applies *tac1* and then applies *tac2* to the first resulting subgoal

lemma_tac: example

SML
```
set_goal([], ⌜(∀x y•x ≤ y ⇒ P(x,y)) ∧ x = y ⇒ P(x,y)⌝);
a(strip_tac);
```

ProofPower Output
```
(∗  2 ∗)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(∗  1 ∗)  ⌜x = y⌝

(∗ ?⊢ ∗)  ⌜P (x, y)⌝
```

SML
```
a(lemma_tac⌜x ≤ y⌝);
```

ProofPower Output
```
(∗ ∗∗∗ Goal "2" ∗∗∗ ∗)
(∗  3 ∗)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(∗  2 ∗)  ⌜x = y⌝
(∗  1 ∗)  ⌜x ≤ y⌝

(∗ ?⊢ ∗)  ⌜P (x, y)⌝

(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)
(∗  2 ∗)  ⌜∀ x y• x ≤ y ⇒ P (x, y)⌝
(∗  1 ∗)  ⌜x = y⌝

(∗ ?⊢ ∗)  ⌜x ≤ y⌝
```

You should now be able to attempt the exercises in section 13.10.

INDUCTION

Induction principles can be expressed and proven as *theorem*s in Higher Order Logic, e.g.:

- **induction_thm** is the usual principle of induction over the natural numbers

$$\begin{array}{l} \vdash \forall\ p\bullet\ p\ 0 \quad \wedge \\ \quad (\forall\ m\bullet\ p\ m \Rightarrow p\ (m + 1)) \\ \Rightarrow \quad (\forall\ n\bullet\ p\ n)\ :\ THM \end{array}$$

- **cov_induction_thm** expresses course of values induction over the natural numbers

$$\begin{array}{l} \vdash \forall\ p\bullet\ (\forall\ n\bullet\ (\forall\ m\bullet\ m < n \Rightarrow p\ m) \Rightarrow p\ n) \\ \Rightarrow \quad (\forall\ n\bullet\ p\ n)\ :\ THM \end{array}$$

- **list_induction_thm** is the principle of structural induction over lists

$$\begin{array}{l} \vdash \forall\ p\bullet\ p\ []\ \wedge \\ \quad (\forall\ list\bullet\ p\ list \Rightarrow (\forall\ x\bullet\ p\ (Cons\ x\ list))) \\ \Rightarrow \quad (\forall\ list\bullet\ p\ list)\ :\ THM \end{array}$$

Using $\forall$*_elim* and *all_β_rule* these can be specialised for use in forward proofs, however, in practice induction principles are normally used via induction tactics.

8.1 Induction Tactics

Special tactics are available to facilitate the use of induction principles:

In the following descriptions of tactics the notation expressing the effect of the tactic shows the goal at the point of applying the tactic above a double horizontal line. To the right of the line is the tactic employed together with any parameters, and below the line the resulting subgoals are described.

Goals are expressed as lists of assumptions enclosed in set brackets (suggesting that for many purposes the assumptions behave like a set) followed by the conclusion. Often these will involve meta-variables, though in these informal presentations there is no systematic way of distinguishing meta-variables from object language variables.

The notation 'strip{a, A}' means 'the set of assumptions arising from stripping the new assumption 'a' into the list of assumptions 'A'. Since this process may result in any number of subgoals the number of subgoals visible below the double line indicates only the number which would arise if no case splits or goal accomplishments occur during stripping.

If more than one goal is shown below the line they are separated by semicolons.

- induction over natural numbers using **induction_tac**

$$\frac{\{\ \Gamma\ \}\ t}{\{\ \Gamma\ \}\ t[0/x];\ strip\{t,\ \Gamma\}\ t[x{+}1/x]} \qquad induction\_tac\ \ulcorner x \urcorner$$

This tactic supports proof by induction over the natural numbers. It takes as a parameter a term which should be a variable of type ⌜:ℕ⌝ which occurs free in the conclusion of the current goal (but not free in the assumptions). Two subgoals would typically arise, corresponding to the base case and the induction step. The base case is arrived at by substituting ⌜0⌝ for the variable supplied as parameter, in the conclusion of the original goal, without changing the assumptions. The step case is obtained by stripping the original conclusion into the assumptions replacing the conclusion by the term resulting from substituting the induction variable +1 (⌜x + 1⌝) into the original conclusion.

- induction over natural numbers using **cov_induction_tac**

$$\frac{\{\Gamma\}\ t}{strip\{\ulcorner\forall m\bullet\ m < x \Rightarrow t[m/x]\urcorner,\ \Gamma\}\ t} \qquad cov\_induction\_tac\ \ulcorner x \urcorner$$

Course of values induction has a strong feel of getting something for nothing. Applied under similar circumstances as the standard natural number induction tactic, the resulting subgoal has the same conclusion as the original goal, but one extra premise is stripped into the assumptions. The extra premise is that for all values less than the value of the induction variable the original conclusion is true.

- for induction over lists **list_induction_tac** may be used

$$\frac{\{\Gamma\}\ t}{\{\Gamma\}\ t[[]/x];\ strip\{t,\ \Gamma\}\ t[Cons\ h\ x/x]} \qquad list\_induction\_tac\ \ulcorner x \urcorner$$

In the case of induction over lists the base case subgoal is obtained by substituting the empty list for the induction variable, while the step case involves substitution for the induction variable of the non-empty list formed by adding a value denoted by a new variable onto the front of the list denoted by the induction variable.

- for other kinds of induction **gen_induction_tac** may be useful

SML
```
gen_induction_tac      : THM -> TERM -> TACTIC;
```

This function is takes as its first parameter a *theorem* expressing an induction principle (e.g. *induction_thm*, *list_induction_thm*), and returns an induction tactic using that principle. This is a quick way of getting a *TACTIC* from a new induction principle which the user may have proved. For full details see theProofPower *Reference Manual* [20].

8.2 Induction Example

The theory of lists contains few *theorem*s, but most of the results needed are simple to prove by induction.

The following example is a proof of the associativity of append, further examples are found in the exercises.

First we set the goal, a universally quantified statement of the associativity of the append operator ⌜$@⌝, then immediately we strip the goal, discarding the universal quantifiers.

SML

```
set_goal([],⌜∀l1 l2 l3:'a LIST•
        (l1 @ l2) @ l3 = l1 @ (l2 @ l3)⌝);
(* remove universal quantifiers *)
a (REPEAT strip_tac);
```

ProofPower output

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

(* ?⊢ *)  ⌜(l1 @ l2) @ l3 = l1 @ l2 @ l3⌝
```

We now proceed by induction on ⌜l1⌝

SML

```
a (list_induction_tac ⌜l1⌝);
```

ProofPower output

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

(*  1 *)  ⌜(l1 @ l2) @ l3 = l1 @ l2 @ l3⌝

(* ?⊢ *)  ⌜∀ x• (Cons x l1 @ l2) @ l3
                 = Cons x l1 @ l2 @ l3⌝

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

(* ?⊢ *)  ⌜([] @ l2) @ l3 = [] @ l2 @ l3⌝
```

The base case is solved simply by rewriting with the definition of append:

SML

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

ProofPower output

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

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

(*  1 *)  ⌜(l1 @ l2) @ l3 = l1 @ l2 @ l3⌝

(* ?⊢ *)  ⌜∀ x• (Cons x l1 @ l2) @ l3
                 = Cons x l1 @ l2 @ l3⌝
```

In the step case the induction assumption has to be used, and we therefore rewrite with the assumptions as well as the definition of append.

SML

```
a (asm_rewrite_tac [append_def]);
val append_assoc_thm = pop_thm();
```

ProofPower output

```
Tactic produced 0 subgoals:
Current and main goal achieved
val append_assoc_thm =
 ⊢ ∀ l1 l2 l3• (l1 @ l2) @ l3 = l1 @ l2 @ l3 : THM
```

Finally the value of the *theorem* has been bound to the ML name *append_assoc_thm*.

You should now be able to attempt the exercises in section 13.8.

*TACTICAL*s AND OTHER 'ALS

A **TACTICAL** is a function which computes a *TACTIC* usually taking one or more *TACTIC*s as arguments. The suffix 'AL' follows mathematical usage in calling a function which operates over functions a functional.

There are many other kinds of higher order functions in ProofPower which greatly enhance productivity in developing new proof facilities.

At a very elementary level programming *TACTIC*s using *TACTICAL*s occurs througout proof work. At each step in a proof using the subgoal package an ML expression is presented for application to the current goal. *TACTICAL*s are very often used in a very simple way to compute the *TACTIC* to be applied.

More complex and sophisticated tactical programming may be used to make small or large extensions to the automatic proof capabilities of the system.

- *TACTICAL*s may be used to combine the available tactics.
- Expressions using *TACTICAL*s may be used directly in proofs, e.g.:

```
        a (REPEAT strip_tac);
```

- named tactics may be defined using *TACTICAL*s:

SML
```
        val repeat_strip_tac = REPEAT strip_tac;
```

- *TACTICAL*s may be used to define parameterised tactics:

SML
```
        fun list_induct_tac t = REPEAT strip_tac
                                THEN list_induction_tac t;
```

*TACTICAL*s usually have capitalised names ending in **_T**, though the most common (e.g. *REPEAT*, *THEN*) have aliases omitting the *_T*. This facilitates identification of the full range of *TACTICAL*s supplied with ProofPower since their names are all collected together under *_T* in the *KWIC index* to the reference manual.

As well as *TACTICAL*s many other higher order functions are available for programming extensions to the systems proof capability:

Among these are **conversional**s (ending with **_C**) which are used to compute *conversion*s, **THM_TACTICAL**s (ending with **_THEN**) which compute *THM_TACTIC*s from *THM_TACTIC*s, and **THM_TACTICAL combinator**s (ending with **_TTCL**) which compute *THM_TACTICAL*s from *THM_TACTICAL*s .

9.1 Commonly used *TACTICAL*s

- **REPEAT** - takes a tactic and returns a tactic which repeats that tactic until it fails.

  If goal splits occur the *REPEAT*ing continues on all subgoals.
- **THEN** - an infix tactical which composes two tactics together. The second tactic is applied to all subgoals arising from the first tactic. If any applications of the operand tactics fail then the resulting tactic fails.
- **ORELSE** - an infix tactical which attempts to apply its first argument, and if this fails applies its second argument. If both arguments fail then the resulting tactic fails.
- **TRY** - a tactical taking one argument which will attempt to do the *TACTIC* which is its argument and will do nothing (but succeed!) if its argument tactic fails.
- **THEN_TRY** - variant on *THEN* which does not fail even if the second tactic fails.

  t1 *THEN_TRY* t2 = t1 *THEN* (*TRY* t2)

9.2 Processing of "New" Assumptions

Tactics which add new assumptions normally do so using *strip_asm_tac*.

E.g., *strip_tac*, *cases_tac*, *lemma_tac* work like this.

However, there are many alternative things which you might want to do with the result instead of than stripping it into the assumptions. Sometimes, the stripping causes more case splitting than is desirable, you may wish to add the assumption without stripping it. Alternatively you may wish to perform other transformations before adding the result into the assumptions.

To give this kind of flexibility most *TACTIC*s which normally strip in new assumptions have corresponding *TACTICAL*s which will operate on a *THM_TACTIC* (which takes a *theorem* and yields a *TACTIC*) to give a new *TACTIC*. The new tactic performs the same function as the first tactic, except that instead of stripping in new assumptions it passes them to the *THM_TACTIC* for processing.

If *xxx_tac* is a *TACTIC* which creates new assumptions, then often *XXX_T* is a corresponding *TACTICAL* which allows the user to select how the results (which *xxx_tac* would have added to the assumptions) are processed.

For example, *cases_tac c* is the same as *CASES_T c strip_asm_tac*, where:

SML
```
|cases_tac     : TERM -> TACTIC;
|CASES_T       : TERM -> (THM -> TACTIC) -> TACTIC;
|strip_asm_tac : (THM -> TACTIC);
```

If the new assumptions arising from an application of *cases_tac* were needed to do a single rewrite of the conclusions then this could be done without adding them to the assumptions using: *CASES_T rewrite_thm_tac*. You need to be sure you will not need the assumptions again before using this technique. It's safe in examples like the following:

SML
```
|set_goal([], ⌜(if x < y + 1 then x else y) < y + 1⌝);
|a(CASES_T ⌜x < y + 1⌝ rewrite_thm_tac);
```

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

where the tactic finishes (a branch of) the proof.

Another occasionally useful *THM − > TACTIC* is *ante_tac*, which places its *THM* argument into the left of an implication in the new conclusion, of which the previous conclusion is the right hand side. This is like the inverse of stripping an implication and can be used to place an assumption in the conclusion for rewriting or automatic proof.

You should now be able to attempt the exercises in section 13.9.

PROOF CONTEXTS

10.1 Introduction

The behaviour of the proof facilities provided in ProofPower has been made 'context sensitive' in order that:

- When the user shifts from using one specification language to another the behaviour of the proof facilities changes in ways appropriate to the language in which reasoning is being conducted.
- Knowledge about various theories or problem domains can be applied automatically, in appropriate contexts, reducing the burden on the user to identify appropriate results from the relevant theories.

This is achieved by making many of the proof facilities (*TACTIC*s etc.) sensitive the values held in the current **proof context**, and by supplying a collection of *proof context*s in which the user can chose to conduct his proofs, as well as facilities for building new *proof context*s.

A *proof context* is therefore a named collection of settings of (implicit) parameters for many of the tactics, conversions, rules etc.

Proof contexts are used to customise many parts of the system including:

- stripping (*strip_tac*, *strip_asm_tac* etc.)
- rewriting (*rewrite_tac* etc.)
- forward chaining (*fc_tac*, *asm_fc_tac*, *all_fc_tac*)

Each *proof context* also contains some 'automatic' provers, which will attempt to solve problems in the domain for which they have been constructed.

- for general purpose automatic proof: *prove_tac* attempts to prove the conclusion of the current goal without using the assumptions, *asm_prove_tac* attempts such a proof making use of the assumptions of the current goal.
- for automatic existence proof *prove_∃_tac* is provided. This is used whenever a constant is specified using *const_spec* or a *HOLCONST* paragraph to attempt to prove the consistency of the specification, and is useful for use interactively in proving existentially quantified goals. Where a constant specification has not been proven automatically, manual application of *prove_∃_tac* often greatly simplifies the manual proof.

Each *proof context* is associated with some particular theory which must be in scope when the *proof context* is in use.

Some *proof context*s recommended for everyday use, together with the name of the relevant theory, are:

predicate calculus	*predicates*
sets	*sets_ext1*
above + lists etc.	*hol2*, *hol*

The function *get_pcs* may be used to list the names of all the *proof context*s currently available together with the name of theory each *proof context* is associated with.

Proof contexts with names beginning with ′ are *component proof context*s. These are intended for use in conjunction with others. Names without ′ are *complete proof context*s suitable for use on their own. The simplest way to create a new *proof context* is to create a *component* context containing only the new material and then merge that with other existing *proof context*s to give the complete range of coverage required.

10.2 Using Proof Contexts

The simplest way to use a *proof context* is to set the *proof context* using *set_pc* or *push_pc*:

```
SML
|set_pc        : string -> unit;
|push_pc       : string -> unit;
```

Each of these takes a single string parameter which is the name of the *proof context* which is to become the *current proof context*. Thenceforth all *TACTIC*s which make use of the *proof context* will refer to the identified *proof context*, until the *proof context* is changed again. The *proof context* identified must be associated with a theory which is currently in scope.

The *push* variant operates on a stack of *proof context*s, the current context being the one at the top of the stack, and is therefore useful if a temporary change of context is required, permitting the previous context to be restored using *pop_pc*.

```
SML
|pop_pc        : unit -> unit;
```

It is also possible to make local switches of *proof context*, perhaps for just one tactic, conversion or rule.

Among the facilities supporting this kind of context switching are:

```
SML
|PC_T          : string -> TACTIC -> TACTIC;
|PC_T1         : string -> ('a -> TACTIC) -> 'a -> TACTIC;
|
|PC_C          : string -> CONV -> CONV;
|PC_C1         : string -> ('a -> CONV) -> 'a -> CONV;
|
|pc_rule       : string -> ('a -> THM) -> 'a -> THM;
|pc_rule1      : string -> ('a -> 'b -> THM) -> 'a -> 'b -> THM;
```

Note here that the variants ending with '1' here are variants suitable for use with parameterised *TACTICS*, *conv*ersions or rules. These are needed where the parameterised function accesses the current *proof context* when it is supplied its first argument.

For example, to do a rewrite in a different *proof context* than the current context the following, will not suffice:

SML
```
|a (PC_T "lin_arith" (rewrite_tac[]));
```

even though this will be accepted and will do some rewriting. This is because the subexpression *(rewrite_tac[])* (intended to rewrite using only the default rewrite conversions in the current *proof context*) is evaluated first, in the current *proof context*, to yield a *TACTIC*. At this stage *rewrite_tac* has already made use of the current *proof context* in preprocessing the *theorems* to be used for rewriting, and has accessed the default rewriting conversions in the *proof context*. The resulting TACTIC is then applied in *proof context* "lin_arith" but is no longer context sensitive.

The correct way to rewrite in a specific *proof context* is:

SML
```
|a (PC_T1 "lin_arith" rewrite_tac []);
```

in which case *PC_T1* arranges for the *proof context* to be switched before *rewrite_tac* is applied to its first argument.

It is also possible to combine the contents of more than one of the available *proof contexts* using *proof context merge* facilities.

SML
```
|set_merge_pcs     : string list -> unit;
|MERGE_PCS_T       : string list -> TACTIC -> TACTIC;
```

which behave in a similar manner to their non-merge version, but accept a list of *proof context* names instead of a single name, and combine the contexts together to form the new current *proof context*.

print_status may be used to discover which *proof context* is current.

SML
```
|print_status      : unit -> unit;
```

10.2.1 What's in the *proof contexts*?

We now show one or two examples of the effects of some of the commonly used *proof contexts*.

Proof contexts which have the atom 'ext' in their name are usually contexts which apply the rule of extensionality of sets by default. Depending on what you are trying to prove this may either enable a rapid discharge of the goal (e.g. if the goal is a result of elementary set theory) or may cause confusion by expanding things which are not appropriate. It is therefore useful to get a feel for when extensional reasoning is likely to be helpful.

An example of the effect of using extensionality is:

SML
```
|PC_C1 "sets_ext1" rewrite_conv[]
|        ⌜{(1, 2)} ⊆ {(x, y) | x + 1 ≤ y} ∨ 4 > 5⌝;
```

ProofPower Output:
```
|val it = ⊢ {(1, 2)} ⊆ {(x, y)|x + 1 ≤ y} ∨ 4 > 5
|   ⇔ (∀ x1 x2• (x1, x2) = (1, 2) ⇒ x1 + 1 ≤ x2)
|        ∨ 4 > 5 : THM
```

Not all *proof context*s which contain these rules do have the 'ext' atom, the most frequently used extensional context without it is '**hol2**', which combines a knowledge of most of the theories which are ancestors of the theory 'hol' with a propensity for extensional reasoning. This means that there are a lot of elementary results which this context will solve automatically, but for less elementary results it may be too aggressive and cause the goal to become complicated and difficult to relate to the original problem.

SML
```
PC_C1 "hol2" rewrite_conv[] ⌜{(1, 2)} ⊆ {(x, y) | x + 1 ≤ y} ∨ 4 > 5⌝;
```

ProofPower Output:
```
val it = ⊢ {(1, 2)} ⊆ {(x, y)|x + 1 ≤ y} ∨ 4 > 5
                ⇔ (∀ x1 x2• x1 = 1 ∧ x2 = 2 ⇒ x1 + 1 ≤ x2) : THM
```

SML
```
PC_C1 "hol2" rewrite_conv[]⌜A ∩ A ⊆ B⌝;
```

ProofPower Output:
```
val it = ⊢ A ∩ A ⊆ B ⇔ (∀ x• x ∈ A ⇒ x ∈ B) : THM
```

SML
```
PC_C1 "hol" rewrite_conv[]⌜A ∩ A ⊆ B⌝;
```

ProofPower Output:
```
val it = ⊢ A ∩ A ⊆ B ⇔ A ⊆ B : THM
```

10.3 Automatic Proof Procedures

To encourage the development of domain specific automatic proof *TACTIC*s, which are capable of solving a useful collection of problems using domain specific techniques the *proof context*s have a place for such *TACTIC*s.

This provides a uniform interface to automatic proof *TACTIC*s reducing the tendency for such domain specific proof automation to cause every greater complexity in the user interface. A good example of the provision of this kind of facility is the provision of linear arithmetic. Though the linear arithmetic package is a complex package, a large part of its functionality is made available to the user through existing procedural interfaces. If the user selects the **lin_arith** *proof context* then the standard rewriting facilities will perform normalisation operations on the operands of arithmetic relations, and *prove_tac* will solve goals whose conclusions are terms in linear arithmetic.

The automatic proof facilities in the current *proof context* may be accessed by the use of the following three functions:

SML
```
prove_tac             : THM list -> TACTIC;
```

- when the conclusion of a goal is automatically provable on its own

SML
```
asm_prove_tac        : THM list -> TACTIC;
```

- when the goal is automatically provable using the assumptions

SML
```
prove_rule           : THM list -> TERM -> THM;
```

- to state and prove a conjecture automatically

If you merge several *proof contexts*, the "*prove_tac*" comes from the last one in the list.

Many *proof contexts* contain *basic_prove_tac*. It uses rewriting, a simple heuristic for eliminating equations involving variables, and a few steps of first-order resolution.

As seen with the *theorems* from PM and ZRM (in the 'easy proof' exercises, section 13.1), this is useful for simple predicate calculus *theorems* and for elementary facts about sets.

For example:

SML
```
prove_rule [] ⌜(∃x• φx) ∨ (∃y• ψy) ⇔ (∃z• φz ∨ ψz)⌝;
prove_rule [] ⌜∀a b•a ⊆ b ∧ b ⊆ a ⇔ a = b⌝;
```

ProofPower Output
```
val it = ⊢ (∃ x• φ x) ∨ (∃ y• ψ y) ⇔ (∃ z• φ z ∨ ψ z) : THM
val it = ⊢ ∀ a b• a ⊆ b ∧ b ⊆ a ⇔ a = b : THM
```

10.4 Linear Arithmetic

Proof context *lin_arith* contains an automatic proof procedure for **linear arithmetic**.

This is useful for many simple arithmetic problems.

For example:

SML
```
pc_rule1 "lin_arith" prove_rule[] ⌜a ≤ b ∧ a + b < c + a ⇒ a < c⌝;
```

ProofPower Output
```
val it = ⊢ a ≤ b ∧ a + b < c + a ⇒ a < c : THM
```

'linear arithmetic' means terms built up from:

- "Atoms" (numeric literals, variables of type ℕ, etc.)
- Multiplication by numeric literals
- Addition, =, ≤, ≥, <, >
- Logical operators

So all the following are terms of linear arithmetic:

$$\forall a\ c \bullet (\exists b \bullet a \geq b \land \neg\ b < c) \Rightarrow a \geq c$$
$$\forall a\ b\ c \bullet a + 2*b < 2*a \Rightarrow b + b < a$$
$$\forall\ x\ y \bullet \neg\ (2*x + y = 4 \land 4*x + 2*y = 7)$$

Rewriting/stripping in *lin_arith* processes numeric relations by "multiplying out and collecting like terms" as follows.

SML
```
pc_rule1 "lin_arith" rewrite_conv[]
        ⌜(i + j)*(j + i) ≤ j*j + j⌝;
```

ProofPower Output
```
val it = ⊢ (i + j) * (j + i) ≤ j * j + j
        ⇔ i * i + 2 * i * j ≤ j : THM
```

$i * i$, $i * j$ and j are then treated as atoms, so this *proof context* solves problems a little more general than "strict" linear arithmetic.

$$\neg(a < 1 + 2*b \land 4*b < 2*a)$$

is proved thus:

if	(1)	$a \leq 2 * b$
and	(2)	$4 * b + 1 \leq 2 * a$
then	2*(1) +(2)	$2 * a + 4 * b + 1 \leq 2 * a + 4 * b$
whence		$1 \leq 0$
whence		CONTRADICTION

You should now be able to attempt the exercises in section 13.12 (other exercises involving linear arithmetic may be found in sections 13.14 and 13.15).

CASE STUDY: VENDING MACHINE

11.1 Vending Machine Specification

The following paragraphs give a model of a simple vending machine:

SML

```
(open_theory "usr013" handle _ => (open_theory "hol"; new_theory "usr013"));
set_pc "hol2";
```

The state of the vending machine is defined as a labelled record type *VM_State* by the following paragraph:

HOL Labelled Product

```
VM_State
    takings          : ℕ;
    stock            : ℕ;
    price            : ℕ;
    cash_tendered    : ℕ
```

The labelled record type paragraph declares the following *projection functions*:

Projection Functions

```
takings       : VM_State → ℕ
stock         : VM_State → ℕ
price         : VM_State → ℕ
cash_tendered : VM_State → ℕ
```

If *st* is a state value, *takings st* is like *st.takings* in Z or Pascal or Ada.

This paragraph also introduces a *constructor* function:

Constructor Function

```
MkVM_State      : ℕ → ℕ → ℕ → ℕ → VM_State
```

If *t*, *s*, *p*, and *ct* are numbers, *MkVM_State t s p ct* is a state value with those numbers as its components.

The following paragraph introduces a new constant *vm* which is a functional model of the behaviour of the vending machine.

HOL Constant

```
│      vm : VM_State → VM_State
├──────────────────────────────────
│∀st•   vm st
│=      if      stock st = 0
│       then    MkVM_State
│               (takings st) (stock st) (price st) 0
│       else    if cash_tendered st < price st
│       then    st
│       else    if cash_tendered st = price st
│       then    MkVM_State
│               (takings st + cash_tendered st)
│               (stock st − 1) (price st) 0
│       else    MkVM_State
│               (takings st) (stock st) (price st) 0
```

We can use ProofPower to 'test' or 'animate' the behaviour of the vending machine model using the ProofPower rewriting facilities.

The following conversion may be used for animating *vm*:

SML

```
│val run_vm = rewrite_conv [get_spec ⌜vm⌝, get_spec ⌜MkVM_State⌝];
```

ProofPower Output

```
│val run_vm = fn : CONV
```

The behaviour of the vending machine can now be illustrated by executing *run_vm* on various vending machine states.

SML

```
│run_vm ⌜vm (MkVM_State 0 20 5 5)⌝;
│run_vm ⌜vm (MkVM_State t 20 5 5)⌝;
```

ProofPower Output

```
│val it = ⊢ vm (MkVM_State 0 20 5 5)
│              = MkVM_State 5 19 5 0 : THM
│
│val it = ⊢ vm (MkVM_State t 20 5 5)
│              = MkVM_State (t + 5) 19 5 0 : THM
```

Note that the second test case above is effectively doing *symbolic execution.*

11.2 Vending Machine Critical Requirements

The critical requirement might be informally stated as:

"No transaction of the vending machine causes the machine's owner to lose money".

We formalise this by specifying the set of transition functions which never reduce the value of the machine's contents.

The value of a state is computed by the following function.

HOL Constant

```
│        value : VM_State → ℕ
├──────────────────────────────────
│∀ st •value st = takings st + stock st ∗ price st
```

The set of machines satisfying the critical requirement is then:

HOL Constant

```
│        vm_ok : (VM_State → VM_State) SET
├──────────────────────────────────
│        vm_ok
│=       {       trf
│        |       ∀cb s p ct•
│                let      s1 = MkVM_State cb s p ct
│                in let   s2 = trf s1
│                in       value s2 ≥ value s1}
```

You should now be able to attempt the exercises in section 13.13.

PROOF STRATEGY

A large application proof may take several man years of effort to complete.

Top level proof strategy for large proofs must be carefully thought out. Major lemmas are best proven separately, stored in the theory, and combined in a top level proof delivering the required result. Exploration may be forwards or backwards.

Lemmas of moderate size may be proven using the goal package. Such a proof would consist of a combination of stripping, rewriting with definitions, assumptions and previously proven results, and other uses of previous results.

12.1 What to do when faced with a Goal

Sanity Checks:

- Decide whether the goal is true, if not, don't try to prove it!
- Decide whether the conclusion is relevant (are the assumptions inconsistent?).
- Do you see what the goal means? If not, can you simplify it.
- If all else fails, try retracing your steps.

Main Choices

- Decompose by stripping or contradiction (*strip_tac*, *contr_tac*)
- Work forwards from assumptions (e.g. *spec_asm_tac*, *fc_tac*)
- Do a case split (*strip_asm_tac*, *cases_tac*)
- Swap the conclusion with an assumption (*swap_asm_concl_tac*)
- Prove a lemma (*lemma_tac*)
- Prove automatically (e.g. *asm_prove_tac*, *prove_∃_tac*)
- Transform the conclusion by rewriting (e.g. with a definition)
- Induction (*..._induction_tac*)

You should now be able to attempt the exercises in sections 13.14 and 13.15.

EXERCISES

This Chapter contains the exercises set for students on the last short course on ProofPower-HOL, with the exception of the exercise in Section 13.15.

Solutions to most of these exercises may be found in Chapter 14.

Unless otherwise stated the exercises should be conducted in proof context 'hol2'.

The source of the exercises for use with copy-and-paste may be found in file `usr022_slides.doc` (the tutorial overheads) or `usr013X.doc`.

13.1 Easy Proofs

Set the theory and the proof context:

```
SML
|open_theory"hol";
|set_pc "hol2";
```

Set the goal (from the examples supplied):

```
|set_goal([],⌜conjecture⌝);
```

Then try the following methods of proof:

- Two tactic method using:

```
|a contr_tac; (* once *)
|a (list_spec_asm_tac ⌜asm⌝ [⌜t1⌝, ⌜t2⌝]);
|  (* as many as necessary *)
```

- or

```
|a (prove_tac[]); (* once *)
```

- or

```
|a step_strip_tac; (* many times *)
```

in case of difficulty, revert to the two tactic method.

SML

```
(* Results from Principia Mathematica *2 *)
val PM2 =[
⌜(* *2.02 *) q ⇒ ( p ⇒ q)⌝,
⌜(* *2.03 *) (p ⇒ ¬ q) ⇒ (q ⇒ ¬ p)⌝,
⌜(* *2.15 *) (¬ p ⇒ q) ⇒ (¬ q ⇒ p)⌝,
⌜(* *2.16 *) (p ⇒ q) ⇒ (¬ q ⇒ ¬ p)⌝,
⌜(* *2.17 *) (¬ q ⇒ ¬ p) ⇒ (p ⇒ q)⌝,
⌜(* *2.04 *) (p ⇒ q ⇒ r) ⇒ (q ⇒ p ⇒ r)⌝,
⌜(* *2.05 *) (q ⇒ r) ⇒ (p ⇒ q) ⇒ (p ⇒ r)⌝,
⌜(* *2.06 *) (p ⇒ q) ⇒ (q ⇒ r) ⇒ (p ⇒ r)⌝,
⌜(* *2.08 *) p ⇒ p⌝,
⌜(* *2.21 *) ¬ p ⇒ (p ⇒ q)⌝];
```

SML

```
(* Results from Principia Mathematica *3 *)
val PM3 =[
(* *3.01 *) ⌜p ∧ q ⇔ ¬(¬ p ∨ ¬ q)⌝,
(* *3.2  *) ⌜p ⇒ q ⇒ p ∧ q⌝,
(* *3.26 *) ⌜p ∧ q ⇒ p⌝,
(* *3.27 *) ⌜p ∧ q ⇒ q⌝,
(* *3.3  *) ⌜(p ∧ q ⇒ r) ⇒ (p ⇒ q ⇒ r)⌝,
(* *3.31 *) ⌜(p ⇒ q ⇒ r) ⇒ (p ∧ q ⇒ r)⌝,
(* *3.35 *) ⌜(p ∧ (p ⇒ q)) ⇒ q⌝,
(* *3.43 *) ⌜(p ⇒ q) ∧ (p ⇒ r) ⇒ (p ⇒ q ∧ r)⌝,
(* *3.45 *) ⌜(p ⇒ q) ⇒ (p ∧ r ⇒ q ∧ r)⌝,
(* *3.47 *) ⌜(p ⇒ r) ∧ (q ⇒ s) ⇒ (p ∧ q ⇒ r ∧ s)⌝];
```

SML

```
(* Results from Principia Mathematica *4 *)
val PM4 =[
(* *4.1  *) ⌜p ⇒ q ⇔ ¬ q ⇒ ¬ p⌝,
(* *4.11 *) ⌜(p ⇔ q) ⇔ (¬ p ⇔ ¬ q)⌝,
(* *4.13 *) ⌜p ⇔ ¬¬ p⌝,
(* *4.2  *) ⌜p ⇔ p⌝,
(* *4.21 *) ⌜(p ⇔ q) ⇔ (q ⇔ p)⌝,
(* *4.22 *) ⌜(p ⇔ q) ∧ (q ⇔ r) ⇒ (p ⇔ r)⌝,
(* *4.24 *) ⌜p ⇔ p ∧ p⌝,
(* *4.25 *) ⌜p ⇔ p ∨ p⌝,
(* *4.3  *) ⌜p ∧ q ⇔ q ∧ p⌝,
(* *4.31 *) ⌜p ∨ q ⇔ q ∨ p⌝,
(* *4.33 *) ⌜(p ∧ q) ∧ r ⇔ p ∧ (q ∧ r)⌝,
(* *4.4  *) ⌜p ∧ (q ∨ r) ⇔ (p ∧ q) ∨ (p ∧ r)⌝,
(* *4.41 *) ⌜p ∨ (q ∧ r) ⇔ (p ∨ q) ∧ (p ∨ r)⌝,
(* *4.71 *) ⌜(p ⇒ q) ⇔ (p ⇔ (p ∧ q))⌝,
(* *4.73 *) ⌜q ⇒ (p ⇔ (p ∧ q))⌝];
```

SML

```
(∗ Results from Principia Mathematica ∗5 ∗)
val PM5 =[
(∗ ∗5.1  ∗) ⌜p ∧ q ⇒ (p ⇔ q)⌝,
(∗ ∗5.32 ∗) ⌜(p ⇒ (q ⇔ r)) ⇒ ((p ∧ q) ⇔ (p ∧ r))⌝,
(∗ ∗5.6  ∗) ⌜(p ∧ ¬ q ⇒ r) ⇒ (p ⇒ (q ∨ r))⌝];
```

SML

```
(∗ Definitions from Principia Mathematica ∗9 ∗)
val PM9 =[
(∗ ∗9.01 ∗) ⌜¬ (∀x• φx) ⇔ (∃x• ¬ φx)⌝,
(∗ ∗9.02 ∗) ⌜¬ (∃x• φx) ⇔ (∀x• ¬ φx)⌝,
(∗ ∗9.03 ∗) ⌜(∀x• φx ∨ p) ⇔ (∀x• φx) ∨ p⌝,
(∗ ∗9.04 ∗) ⌜p ∨ (∀x• φx) ⇔ (∀x• p ∨ φx)⌝,
(∗ ∗9.05 ∗) ⌜(∃x• φx ∨ p) ⇔ (∃x• φx) ∨ p⌝,
(∗ ∗9.06 ∗) ⌜p ∨ (∃x• φx) ⇔ p ∨ (∃x• φx)⌝];
val PM9b =[
(∗ ∗9.07 ∗) ⌜(∀x• φx) ∨ (∃y• ψy) ⇔ (∀x•∃y• φx ∨ ψy)⌝,
(∗ ∗9.08 ∗) ⌜(∃y• ψy) ∨ (∀x• φx) ⇔ (∀x•∃y• ψy ∨ φx)⌝];
```

SML

```
(∗ Results from Principia Mathematica ∗10 ∗)
val PM10 =[
(∗ ∗10.01  ∗) ⌜(∃x• φx) ⇔ ¬ (∀y• ¬ φy)⌝,
(∗ ∗10.1   ∗) ⌜(∀x• φx) ⇒ φy⌝,
(∗ ∗10.21  ∗) ⌜(∀x• p ⇒ φx) ⇔ p ⇒ (∀y• φy)⌝,
(∗ ∗10.22  ∗) ⌜(∀x• φx ∧ ψx) ⇔ (∀y• φy) ∧ (∀z• ψz)⌝,
(∗ ∗10.24  ∗) ⌜(∀x• φx ⇒ p) ⇔ (∃y• φy) ⇒ p⌝,
(∗ ∗10.27  ∗) ⌜(∀x• φx ⇒ ψx) ⇒ ((∀y• φy) ⇒ (∀z• ψz))⌝,
(∗ ∗10.271 ∗) ⌜(∀x• φx ⇔ ψx) ⇒ ((∀y• φy) ⇔ (∀z• ψz))⌝,
(∗ ∗10.28  ∗) ⌜(∀x• φx ⇒ ψx) ⇒ ((∃y• φy) ⇒ (∃z• ψz))⌝,
(∗ ∗10.281 ∗) ⌜(∀x• φx ⇔ ψx) ⇒ ((∃y• φy) ⇔ (∃z• ψz))⌝,
(∗ ∗10.35  ∗) ⌜(∃x• p ∧ φx) ⇔ p ∧ (∃y• φy)⌝,
(∗ ∗10.42  ∗) ⌜(∃x• φx) ∨ (∃y• ψy) ⇔ (∃z• φz ∨ ψz)⌝,
(∗ ∗10.5   ∗) ⌜(∃x• φx ∧ ψx) ⇒ (∃y• φy) ∧ (∃z• ψz)⌝,
(∗ ∗10.51  ∗) ⌜¬(∃x• φx ∧ ψx) ⇒ (∀y• φy ⇒ ¬ ψy)⌝];
```

SML

```
(∗ Results from Principia Mathematica ∗11 ∗)
val PM11 =[
(∗ ∗11.1  ∗) ⌜(∀x y• φ(x,y)) ⇒ φ(z,w)⌝,
(∗ ∗11.2  ∗) ⌜(∀x y• φ(x,y)) ⇔ ∀y x• φ(x,y)⌝,
(∗ ∗11.3  ∗) ⌜(p ⇒ (∀x y• φ(x,y)))
                 ⇔ (∀x y• p ⇒ φ(x,y))⌝,
(∗ ∗11.32 ∗) ⌜(∀x y• φ(x,y) ⇒ ψ(x,y))
│               ⇒ (∀x y• φ(x,y)) ⇒ (∀x y• ψ(x,y))⌝,
│(∗ ∗11.35 ∗) ⌜(∀x y• φ(x,y) ⇒ p) ⇔ (∃x y• φ(x,y)) ⇒ p⌝,
│(∗ ∗11.45 ∗) ⌜(∃x y• p ⇒ φ(x,y))
│               ⇔ (p ⇒ (∃x y• φ(x,y)))⌝,
│(∗ ∗11.54 ∗) ⌜(∃x y• φx ∧ ψy) ⇔ (∃x• φx) ∧ (∃y• ψy)⌝,
│(∗ ∗11.55 ∗) ⌜(∃x y• φx ∧ ψ(x,y))
│               ⇔ (∃x• φx ∧ (∃y• ψ(x,y)))⌝,
│(∗ ∗11.6  ∗) ⌜(∃x• (∃y• φ(x,y) ∧ ψy) ∧ χx)
│               ⇔ (∃y• (∃x• φ(x,y) ∧ χx) ∧ ψy)⌝,
│(∗ ∗11.62 ∗) ⌜(∀x y• φx ∧ ψ(x,y) ⇒ χ(x,y))
│               ⇔ (∀x• φx ⇒ (∀y• ψ(x,y) ⇒ χ(x,y)))⌝
│];
```

SML

```
│(∗ results from ZRM provable by stripping ∗)
│val ZRM1 = [
│⌜a ∪ a = a ∪ {}⌝,
│⌜a ∪ {} = a ∩ a⌝,
│⌜a ∩ a = a \ {}⌝,
│⌜a \ {} = a⌝,
│⌜a ∩ {} = a \ a⌝,
│⌜a \ a = {} \ a⌝,
│⌜{} \ a = {}⌝,
│⌜a ∪ b = b ∪ a⌝,
│⌜a ∩ b = b ∩ a⌝,
│⌜a ∪ (b ∪ c) = (a ∪ b) ∪ c⌝,
│⌜a ∩ (b ∩ c) = (a ∩ b) ∩ c⌝,
│⌜a ∪ (b ∩ c) = (a ∪ b) ∩ (a ∪ c)⌝,
│⌜a ∩ (b ∪ c) = (a ∩ b) ∪ (a ∩ c)⌝,
│⌜(a ∩ b) ∪ (a \ b) = a⌝,
│⌜(a \ b) ∩ b = {}⌝,
│⌜a \ (b \ c) = (a \ b) ∪ (a ∩ c)⌝,
│⌜(a \ b) \ c = (a \ (b ∪ c))⌝,
│⌜a ∪ (b \ c) = (a ∪ b) \ (c \ a)⌝,
│⌜a ∩ (b \ c) = (a ∩ b) \ c⌝,
│⌜(a ∪ b) \ c = (a \ c) ∪ (b \ c)⌝];
```

SML

```
│val ZRM2 = [
│⌜a \ (b ∩ c) = (a \ b) ∪ (a \ c)⌝,
│⌜¬ x ∈ {}⌝,
│⌜a ⊆ a⌝,
│⌜¬ a ⊂ a⌝,
│⌜{} ⊆ a⌝,
│⌜⋃ {} = {}⌝,
│⌜⋂ {} = Universe⌝];
```

SML
```
(∗ results from ZRM ∗)
val ZRM3 = [
⌜a ⊆ b ⇔ a ∈ ℙ b⌝,
⌜a ⊆ b ∧ b ⊆ a ⇔ a = b⌝,
⌜¬ (a ⊂ b ∧ b ⊂ a)⌝,
⌜a ⊆ b ∧ b ⊆ c ⇒ a ⊆ c⌝,
⌜a ⊂ b ∧ b ⊂ c ⇒ a ⊂ c⌝,
⌜{} ⊂ a ⇔ ¬ a = {}⌝,
⌜⋃ (a ∪ b) = (⋃ a) ∪ (⋃ b)⌝,
⌜⋂ (a ∪ b) = (⋂ a) ∩ (⋂ b)⌝,
⌜ a ⊆ b ⇒ ⋃ a ⊆ ⋃ b ⌝,
⌜ a ⊆ b ⇒ ⋂ b ⊆ ⋂ a ⌝];
```

13.2 HOL Theory Explorations

- Find the names of all the theories:

  SML
  ```
  get_theory_names();
  ```

- Print selected theories, e.g.:

  SML
  ```
  open_theory"sets";
  print_theory"sets";
  ```

- Get the terms from the definitions in a theory, e.g.:

  SML
  ```
  open_theory "bin_rel";
  (map concl o map snd o get_defns) "bin_rel";
  ```

- Now select interesting terms and take them apart using, e.g.:

  SML
  ```
  dest_simple_term ⌜∀ r s• r ⊕ s = (Dom s ⩤ r) ∪ s⌝;
  ```

  Hol Output
  ```
  val it = App (⌜$∀⌝, ⌜λ r• ∀ s• r ⊕ s = (Dom s ⩤ r) ∪ s⌝) : DEST_SIMPLE_TERM
  ```

  SML
  ```
  dest_simple_term ⌜{1;2;3}⌝;
  ```

  Hol Output
  ```
  val it = App (⌜Insert 1⌝, ⌜{2; 3}⌝) : DEST_SIMPLE_TERM
  ```

  SML
  ```
  get_spec ⌜Insert⌝;
  ```

  Hol Output
  ```
  val it = ⊢ ∀ x y a
   • ¬ x ∈ {} ∧ x ∈ Universe ∧ (x ∈ Insert y a ⇔ x = y ∨ x ∈ a) : THM
  ```

13.3 Specification

- Create a new theory, e.g. :

  SML
  ```
  open_theory "usr013";
  new_theory "usr013X";
  ```

- Write a specification in HOL of a function to add the elements of a list of numbers.

  HINT: if your specification goes in as a "Constspec" then the system could not prove it consistent, and its probably either wrong or poorly structured. Try to make it clearly 'primitive recursive'.

- Use it to "evaluate" the term ⌜*list_sum*[*1*; *2*; *3*; *4*; *5*]⌝.

  ```
  rewrite_conv[get_spec⌜list_sum⌝]
          ⌜list_sum [1;2;3;4;5]⌝;
  ```

13.4 Forward Proof

1. Using ⇒_*elim* and *asm_rule* prove:

   (a) $b \Rightarrow c, a \Rightarrow b, a \vdash c$

   (b) $a \Rightarrow b \Rightarrow c, a, b \vdash c$

2. Using ∀_*elim* with ¬_*plus1_thm* prove:

   (a) $\vdash \neg 0 + 1 = 0$

   (b) $\vdash \neg x * x + 1 = 0$

3. Using *all*_∀_*elim* with ≤_*trans_thm* prove:

   (a) $\vdash m \le i \land i \le n \Rightarrow m \le n$

4. Using *list*_∀_*elim* prove:

   (a) (with ¬_*less_thm*)$\vdash \neg 0 < 1 \Leftrightarrow 1 \le 0$

   (b) (with ≤_*trans_thm*)⊢ ∀ n• 3 ≤ x * x ∧ x * x ≤ n ⇒ 3 ≤ n

5. Using *all*_∀_*elim*, *strip*_∧_*rule*, *nth*, *all*_∀_*intro*:

   (a) (with ≤_*clauses*)⊢ ∀ i m n• i + m ≤ i + n ⇔ m ≤ n

   (b) (using *list*_∀_*intro*)⊢ ∀ m i n• i + m ≤ i + n ⇔ m ≤ n

13.5 Rewriting with the Subgoal Package

1. set a goal from the examples on set theory, e.g.:

   SML
   ```
   set_goal([],⌜a \ (b ∩ c) = (a \ b) ∪ (a \ c)⌝);
   ```

2. rewrite the goal using the current proof context:

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

3. step back using undo:

   SML
   ```
   undo 1;
   ```

4. now try rewriting without using the proof context:

   ```
   a (pure_rewrite_tac[]);
   ```

   (this should fail)

5. try rewriting one layer at a time:

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

   repeat until it fails.

6. now try rewriting with specific theorems:

   SML
   ```
   set_goal([],⌜a \ (b ∩ c) = (a \ b) ∪ (a \ c)⌝);
   a (pure_rewrite_tac[sets_ext_clauses]);
   a (pure_rewrite_tac[set_dif_def]);
   a (pure_rewrite_tac[∩_def, ∪_def]);
   a (pure_rewrite_tac[set_dif_def]);
   ```

7. finish the proof by stripping:

   SML
   ```
   a (REPEAT strip_tac);
   ```

8. extract the theorem

   SML
   ```
   top_thm();
   ```

9. repeat the above then try repeating:

   SML
   ```
   pop_thm();
   ```

13.6 Combining Forward and Backward Proof

Prove the following results by rewriting using the goal package:

(for each example, try the methods which worked on the previous examples first to see how they fail before following the hint)

Prove:

SML
```
set_goal([],⌜x + y = y + x⌝);
```

1.

SML
```
set_goal([],⌜x + y + z = (x + y) + z⌝);
```

2.

SML
```
set_goal([],⌜z + y + x = y + z + x⌝);
```

3.

SML
```
set_goal([],⌜x + y + z = y + z + x⌝);
```

4.

SML
```
set_goal([],⌜x + y + z + v = y + v + z + x⌝);
```

5.

Hints:

1. try rewriting (with nothing but the default rewrites)
2. try using plus_assoc_thm
3. try using plus_assoc_thm1
4. try using ∀_elim with plus_assoc_thm1
5. try using ∀_elim with plus_order_thm

13.7 Stripping

Use the examples from Principia Mathematica & ZRM given earlier, e.g.:

```
SML
│        set_goal([],⌜p ∧ q ⇒ (p ⇔ q)⌝);
```

with

```
SML
│        a strip_tac;
```

and/or

```
SML
│        a step_strip_tac;
```

Observe the steps taken and try to identify the reasons for discharge of subgoals.

Now select the weakest "proof context":

```
SML
│        push_pc "initial";
```

and retry the examples (or previous exercises).

When you have finished restore the original proof context by:

```
SML
│        pop_pc();
```

13.8 Induction

Prove the following results using the subgoal package.

1. Appending the empty list has no effect

   ```
   SML
   │set_goal([], ⌜∀l1 • l1 @ [] = l1⌝);
   ```

2. "Reverse" distributes over "@" (sort of)

   ```
   SML
   │set_goal([], ⌜∀l1 l2 •
   │  Rev (l1 @ l2) = (Rev l2) @ (Rev l1)⌝);
   ```

3. "Map" distributes over "@"

   ```
   SML
   │set_goal([], ⌜∀f l1 l2 •
   │  Map f (l1 @ l2) = (Map f l1) @ (Map f l2)⌝);
   ```

4. "Length" distributes over "@"

   ```
   SML
   │set_goal([], ⌜∀l1 l2• Length (l1 @ l2)
   │        = Length l1 + Length l2⌝);
   ```

Hints: You will need the result proven in the tutorial text concerning associativity of append (*append_assoc_thm*). The result of the first exercise is also required in the remaining problems.

13.9 TACTICALs

1. Write a tactic which does *strip_tac* three times.

   test it on:

   SML
   ```
   |set_goal([],⌜(a ⇒ b ⇒ c) ⇒ a ⇒ b ⇒ c⌝);
   |set_goal([],⌜(a ⇒ b) ⇒ a ⇒ c⌝);
   ```

2. Write a tactic which does *strip_tac* up to 3 times.

   Try it on the same examples.

3. Write a tactic which takes two arguments:

   - a term which is a variable
   - a list of theorems

   and performs an inductive proof of a theorem concerning lists by:

   - stripping the goal
   - inducting on the variable
   - rewriting with the assumptions and the list of theorems

   Use it to shorten the earlier proofs about lists.

13.10 *strip_asm_tac* etc.

1. Use *strip_asm_tac* (with ∀*_elim* and ℕ*_cases_thm*) or *cases_tac* to prove

   (a) $\forall x \bullet (if\ x = 0\ then\ 1\ else\ x) > 0$

   (b) $\forall x\ y \bullet (if\ x < y + 1\ then\ x\ else\ y) < y + 1$

   (c) $\forall a\ b \bullet a \le (if\ a \le b\ then\ b\ else\ a)$

   (d) $\forall a \bullet a = 0 \lor 1 \le a$

2. Using *(i) swap_asm_concl_tac* and *(ii) lemma_tac* give two different proofs of each of:

   (a) $(\forall x\ y \bullet x \le y \Rightarrow P(x, y)) \Rightarrow (\forall x\ y \bullet x = y \Rightarrow P(x, y))$

   (b) $(\forall x\ y \bullet f\ x \le f\ y \Rightarrow x \le y) \Rightarrow (\forall x\ y \bullet f\ x = f\ y \Rightarrow x \le y)$

13.11 Forward Chaining

1. Experiment with the various *all_* and *asm_* variants of *fc_tac* to prove the following goals:

   (a) (using ≤*_trans_thm*)

   ```
   |        ∀a b c d•a ≤ b ∧ b ≤ c ∧ c ≤ d ⇒ a ≤ d
   ```

(b) (no theorem required)

$$\forall X\ Y\ Z \bullet X \subseteq Y \land Y \subseteq Z \Rightarrow X \subseteq Z$$

In each case, what is the minimum number of applications of a forward chaining tactic required and why?

2. Can you use forward chaining to simplify the proof of the following example from exercises 10: $(\forall x\ y \bullet f\ x \leq f\ y \Rightarrow\ x \leq y) \Rightarrow (\forall x\ y \bullet f\ x = f\ y \Rightarrow x \leq y)$

13.12 Proof Contexts

1. Using *REPEAT strip_tac* and *asm_rewrite_tac* prove

   $(\forall x\ y \bullet f(x,\ y) = (y,\ x)) \Rightarrow \forall x\ y \bullet f(f\ (x,\ y)) = (x,\ y)$

   Apply the tactics one at a time rather than using *THEN*. Now set the proof context to "*predicates*" using *set_pc* and prove it again. What differences do you observe?

   Set the proof context back to "*hol2*" when you've finished.

2. Prove the following

   (a) $\{(x,\ y) \mid \neg x = 0 \land y = 2*x\} \subseteq \{(x,\ y) \mid x < y\}$

   (b) $\{(x,\ y) \mid x \geq 2 \land y = 2*x\} \subseteq \{(x,\ y) \mid x + 1 < y\}$

   (c) $A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$

   (d) $\forall m \bullet \{i \mid m \leq i \land i < m + 3\} = \{m;\ m+1;\ m+2\}$

   (e) $\{i \mid 5*i = 6*i\} = \{0\}$

13.13 Case Study

First of all execute the *new_theory* command and the 4 paragraphs of the vending machine specification.

1. Execute the definition of *run_vm*:

   SML
   ```
   val run_vm = rewrite_conv[get_spec⌜vm⌝, get_spec⌜MkVM_State⌝];
   ```

   Experiment with the model by using *run_vm* to see what it does on various test data. What does the vending machine do if the price is set to *0*?

2. Prove that the model of the vending machine satisfies its critical requirements. I.e., prove:

   vm ∈ vm_ok

   Hints:

   (a) Try *REPEAT strip_tac*

   (b) Try rewriting with the definitions of any of *MkVM_State*, *vm*, *vm_ok* or *worth* which appear in the goal.

(c) *let*-expressions may be eliminated by rewriting with *let_def*.

(d) Is there an *if*-term in the goal? Can you use ℕ*_cases_thm* or *less_cases_thm* (together with *strip_asm_tac* and ∀*_elim* or *list_*∀*_elim*) to perform the relevant case analysis?

(e) If you believe the goal is true by dint of arithmetic facts alone try:

```
            PC_T1 "lin_arith" asm_prove_tac[]
```

(f) If none of the above hints apply, do you have an *if*-term which could be simplified using an "obvious" arithmetic consequence of your assumptions. If so set the "obvious" consequence up as a lemma with *lemma_tac*.

13.14 Advanced Techniques

1. Use *contr_tac*, and *spec_asm_tac* and rewriting prove that there is no greatest natural number:

SML
```
set_goal([], ⌜∀m•∃n• m < n⌝);
```

(Hint: $m < m + 1$).

2. Rather than using *contr_tac*, it is often more natural to prove goals with existentially quantified conclusions directly. ∃*_tac* lets you do this by supplying a term to act as a "witness". Use ∃*_tac* to give a more natural solution to the previous exercise:

SML
```
set_goal([], ⌜∀m•∃n• m < n⌝);
```

3. Prove that there is no onto function from the natural numbers to the set of all numeric functions on the natural numbers:

SML
```
set_goal([], ⌜∀f : ℕ → (ℕ → ℕ)•∃g•∀i•¬f i = g⌝);
```

(Hints: Note that for f of the above type, $\lambda j \bullet (f\ j\ j) + 1$ cannot be in the range of f. Rewriting with *ext_thm* is useful for reasoning about equations between functions.)

4. It can happen that an equation is the wrong way round for use as a rewrite rule. The usual means for dealing with this type of problem is the conversion *eq_sym_conv*. Like other conversions this may be propagated over a term using the conversionals *MAP_C* and *ONCE_MAP_C*. Execute the following lines one at a time to see what happens:

```
eq_sym_conv ⌜1 + 1 + 1 = 3⌝;
eq_sym_conv ⌜∀x•x + x + x = 3*x⌝;
ONCE_MAP_C eq_sym_conv ⌜∀x•x + x + x = 3*x⌝;
```

A conversion may be converted into a tactic using *conv_tac*. Use this and the conversion and conversional you have just experimented with together with the tactics *swap_asm_concl_tac* and the theorems *ext_thm* and *comb_k_def* to prove the following:

SML
```
set_goal([], ⌜∀f:'a→'b→'a•(∀x y•x = f x y) ⇒ f = CombK⌝);
```

(Hint: take care to avoid looping rewrites by using the "once" rewriting tactics while you look for the proof.)

5. A common way of using a theorem is to to strip it into the assumptions. This is done with *strip_asm_tac*. Very often one specialises the theorem with *∀_elim* or *list_∀_elim* before stripping it in and sometimes one may wish to use *rewrite_rule* to rewrite it too. Use the theorem *div_mod_unique_thm* in this way to prove:

SML
```
set_goal([], ⌜∀i j•0 < i ⇒ (i * j) Div i = j⌝);
```

(Hints: rewrite the theorem with *times_comm_thm* suitably specialised to identify subterms of the form $i * j$ and $j * i$ into the same form; use the technique of the previous exercise to avoid a looping rewrite with the assumption added by *strip_asm_tac*).

6. Execute the following paragraph to define a function σ which maps i to the sum of the first i positive integers:

HOL Constant
```
        σ : ℕ → ℕ
─────────────────────────────
        σ 0 = 0
∧       ∀i• σ(i+1) = σ i + (i + 1)
```

The consistency of this paragraph should be proved automatically. Check this by using *get_spec* to get the defining axiom for σ, which should have no assumptions. Prove the following theorem:

SML
```
set_goal([], ⌜∀i•σ i = (i*(i + 1)) Div 2⌝);
```

(Hint: use induction to prove a lemma that $i * (i + 1) = 2 * \sigma i$ and then use the result of the previous exercise; the lemma may be proved by rewriting with assumptions and the definition of σ and then using the proof context *lin_arith*.)

7. Construct a paragraph defining a function ϕ such that for positive i, ϕi is the i^{th} element of the Fibonacci sequence, $1, 1, 2, 3, 5, \ldots$, where each number is the sum of the previous two. Does the system automatically prove the consistency of your definition?

8. If you did the previous exercise, delete the function ϕ you defined (using *delete_const*). Enter the following paragraphs which define ϕ using an auxiliary function γ:

HOL Constant
```
        γ : ℕ → (ℕ × ℕ)
─────────────────────────────
        γ 0 = (0, 1)
∧       ∀i•γ(i+1) = let (a, b) = γ i in (b, a + b)
```

HOL Constant
```
        φ : ℕ → ℕ
─────────────────────────────
        ∀i•φ i = Fst (γ i)
```

These definitions are proved consistent automatically. Prove that ϕ does indeed compute the Fibonacci numbers:

```
│set_goal([], ⌜
│        φ 0 = 0
│∧       φ 1 = 1
│∧       ∀i•φ(i+2) = φ(i+1) + φ i
│⌝);
```

(Hints: first rewrite with the definition of ϕ; then prove a lemma or lemmas showing how $\gamma\ 1$ and $\gamma(i + 2)$ may be rewritten so that the definition of γ may be used to rewrite them.)

9. The approach of the previous exercise has the disadvantage that the specification was not as abstract as one might like. A cleaner approach is to use the obvious definition of ϕ, and then prove that it is consistent using a function γ which is only introduced as a variable during the course of the proof. The tactic *prove_∃_tac* gives access to the mechanisms that the system uses in its attempt to prove that paragraphs are consistent.

We demonstrate the above technique in this exercise.

First of all, delete the function γ that you defined in the previous exercise (using *delete_const*, which will also cause ϕ to be deleted).

```
SML
│delete_const⌜γ⌝;
```

Enter the following paragraph which gives the natural definition of ϕ:

```
HOL Constant
│       φ : ℕ → ℕ
├──────────────────────────────
│       φ 0 = 0
│ ∧     φ 1 = 1
│ ∧     ∀i•φ(i+2) = φ(i+1) + φ i
```

Examine the theorem that *get_spec* returns for ϕ, it has a consistency caveat as an assumption. Discharge this consistency caveat as follows:

First of all go into the subgoaling package using the following command:

```
│push_consistency_goal⌜φ⌝;
```

Now set as a lemma the existence of a γ as in the previous exercise; the lemma is proved immediately by *prove_∃_tac* and you can then use *∃_tac*⌜$\lambda i \bullet Fst(\gamma\ i)$⌝ followed a proof almost identical with the previous exercise (hint: *rewrite_tac* will eliminate the β-redexes introduced when you apply *∃_tac*). Save the consistency theorem using the following command:

```
│save_consistency_thm ⌜φ⌝ (pop_thm());
```

If you now examine the theorem that *get_spec* returns for ϕ, you should see that it no longer has an assumption.

(Note: the variable name 'ϕ'', created by decorating 'ϕ' is displayed by the pretty printer as $ "ϕ'" since it violates the HOL lexical rules for identifiers. The parser will accept identifiers violating the normal lexical rules if they are presented in this way.)

13.15 Supplementary Exercises

13.15.1 Linear Arithmetic

Prove that any amount of money greater than seven cents can be made up from three and five cent coins.

This example came to us from SRI.

Hint: Conduct the proof in proof context 'hol2', invoking 'lin_arith' only when needed. Use proof by induction. In the step case use a case split on whether there are any five cent coins in the solution assumed for the induction variable ⌜i⌝ and construct your witness for ⌜i+1⌝ accordingly.

SOLUTIONS TO EXERCISES

No solutions are provided for those exercises whose nature is exploratory rather than problem solving.

Nevertheless, section headings are included for those exercises, so that the correspondence between sections in this Chapter and Chapter 13 is maintained.

14.1 Easy Proofs

14.2 HOL Theory Explorations

14.3 Specification

14.4 Forward Proof

SML
```
(* 1(a) *)
val ext1_thm1 = asm_rule ⌜a⇒b⌝;
val ext1_thm2 = asm_rule ⌜b⇒c⌝;
val ext1_thm3 = asm_rule ⌜a:BOOL⌝;
val ext1_thm4 = ⇒_elim ext1_thm1 ext1_thm3;
val ext1_thm5 = ⇒_elim ext1_thm2 ext1_thm4;
(* 1(b) *)
val ext2_thm1 =
 ⇒_elim (asm_rule ⌜a⇒b⇒c⌝)(asm_rule ⌜a:BOOL⌝);
```

SML
```
(* 2(a) *)
val ext3_thm1 = ∀_elim ⌜0⌝ ¬_plus1_thm;
(* 2(b) *)
val ext4_thm1 = ∀_elim ⌜x*x⌝ ¬_plus1_thm;
```

SML
```
(* 3(a) *)
val ext5_thm1 = all_∀_elim ≤_trans_thm;
```

SML
```
(* 4(a) *)
val ext6_thm1 = list_∀_elim [⌜0⌝,⌜1⌝] ¬_less_thm;
(* 4(b) *)
val ext7_thm1 = list_∀_elim [⌜3⌝,⌜x*x⌝] ≤_trans_thm;
```

SML
```
(* 5(a) *)
val ext8_thm1 = strip_∧_rule (all_∀_elim ≤_clauses);
val ext8_thm2 = all_∀_intro (nth 3 ext8_thm1);
(* 5(b) *)
val ext8_thm2 = list_∀_intro [⌜m⌝,⌜i⌝,⌜n⌝](nth 3 ext8_thm1);
```

14.5 Rewriting with the Subgoal Package

14.6 Combining Forward and Backward Proof

1. :

SML
```
set_goal([],⌜x + y = y + x⌝);
a (rewrite_tac[]);
```

2. :

SML
```
set_goal([],⌜x + y + z = (x + y) + z⌝);
a (rewrite_tac[plus_assoc_thm]);
```

3. :

SML
```
set_goal([],⌜z + y + x = y + z + x⌝);
a (rewrite_tac[plus_assoc_thm1]);
```

4. :

SML
```
set_goal([],⌜x + y + z = y + z + x⌝);
a (rewrite_tac[∀_elim ⌜y⌝ plus_assoc_thm1]);
```

5. :

SML
```
set_goal([],⌜x + y + z + v = y + v + z + x⌝);
a (rewrite_tac[∀_elim ⌜x⌝ plus_order_thm]);
```

14.7 Stripping

14.8 Induction

SML
```
set_goal([],⌜∀l1 • l1 @ [] = l1⌝);            (* no. 1 *)
a strip_tac;
a (list_induction_tac ⌜l1⌝
  THEN asm_rewrite_tac [append_def]);
val empty_append_thm = pop_thm();
```

SML
```
set_goal([],⌜∀l1 l2 • Rev (l1 @ l2) =
        (Rev l2) @ (Rev l1)⌝);        (* no. 2 *)
a (REPEAT strip_tac);
a (list_induction_tac ⌜l1⌝ THEN asm_rewrite_tac
  [append_assoc_thm, empty_append_thm,
        append_def, rev_def]);
val rev_distrib_thm = pop_thm();
```

SML
```
set_goal([],⌜∀f l1 l2 • Map f (l1 @ l2) =
        (Map f l1) @ (Map f l2)⌝); (* no. 3 *)
a (REPEAT strip_tac);
a (list_induction_tac ⌜l1⌝ THEN asm_rewrite_tac
  [map_def, empty_append_thm, append_def]);
val map_distrib_thm = pop_thm();
```

SML
```
set_goal([],⌜∀l1 l2• Length (l1 @ l2) =
        Length l1 + Length l2⌝);      (* no. 4 *)
a (REPEAT strip_tac);
a (list_induction_tac ⌜l1⌝ THEN asm_rewrite_tac
  [append_def, length_def, plus_assoc_thm]);
val length_distrib_thm = pop_thm();
```

14.9 TACTICALs

SML
```
(* no. 1 *)
val strip3_tac = TRY_T strip_tac THEN strip_tac THEN strip_tac;
set_goal([],⌜(a ⇒ b ⇒ c) ⇒ a ⇒ b ⇒ c⌝);
a strip3_tac;
```

SML
```
(* no. 2 *)
val stripto3_tac = strip_tac THEN_TRY strip_tac
                THEN_TRY strip_tac;
set_goal([],⌜(a ⇒ b) ⇒ a ⇒ c⌝);
a stripto3_tac;
```

SML
```
(* no. 3 *)
fun list_induct_tac var thl =
        REPEAT strip_tac
        THEN list_induction_tac var
        THEN_TRY asm_rewrite_tac thl;

set_goal([],⌜∀l1 l2 l3 •
  (l1 @ l2) @ l3 = l1 @ (l2 @ l3)⌝);
a (list_induct_tac ⌜l1:'a LIST⌝ [append_def]);
val append_assoc_thm = pop_thm ();

set_goal([], ⌜∀l1:'a LIST • l1 @ [] = l1⌝);
a (list_induct_tac ⌜l1:'a LIST⌝ [append_def]);
val empty_append_thm = pop_thm();
```

14.10 *strip_asm_tac* etc.

SML
```
                                    (* (a) *)
set_goal([], ⌜∀x•(if x = 0 then 1 else x) > 0⌝);
a(REPEAT strip_tac);
a(strip_asm_tac(∀_elim⌜x⌝ℕ_cases_thm) THEN asm_rewrite_tac[]);
pop_thm();
```

SML
```
                                    (* (b) *)
set_goal([], ⌜∀x y•(if x < y + 1 then x else y) < y + 1⌝);
a(REPEAT strip_tac);
a(CASES_T ⌜x < y + 1⌝ rewrite_thm_tac);
pop_thm();
```

SML
```
                                    (* (c) *)
set_goal([], ⌜∀a b•a ≤ (if a ≤ b then b else a)⌝);
a(REPEAT strip_tac);
a(CASES_T ⌜a ≤ b⌝ rewrite_thm_tac);
pop_thm();
```

SML
```
                                    (* (d) *)
set_goal([], ⌜∀a•a = 0 ∨ 1 ≤ a⌝);
a(strip_tac);
a(strip_asm_tac(∀_elim⌜a⌝ℕ_cases_thm) THEN asm_rewrite_tac[]);
pop_thm();
```

With *swap_asm_concl_tac*:

SML
```
set_goal([],                              (* (i)(a) *)
 ⌜(∀x y•x ≤ y ⇒  P(x, y)) ⇒ (∀x y•x = y ⇒  P(x, y))⌝);
a(REPEAT strip_tac);
a(list_spec_nth_asm_tac 2[⌜x⌝, ⌜y⌝]);
a(swap_asm_concl_tac ⌜¬ x ≤ y⌝ THEN asm_rewrite_tac[]);
pop_thm();
```

SML
```
set_goal([],                              (* (i)(b) *)
 ⌜(∀x y•f x ≤ f y ⇒  x ≤ y) ⇒ (∀x y•f x = f y ⇒ x ≤ y)⌝);
a(REPEAT strip_tac);
a(list_spec_nth_asm_tac 2[⌜x⌝, ⌜y⌝]);
a(swap_asm_concl_tac ⌜¬ f x ≤ f y⌝ THEN asm_rewrite_tac[]);
pop_thm();
```

With *lemma_tac*:

SML
```
set_goal([],                              (* (ii)(a) *)
 ⌜(∀x y•x ≤ y ⇒  P(x, y)) ⇒ (∀x y•x = y ⇒  P(x, y))⌝);
a(REPEAT strip_tac);
a(lemma_tac⌜x ≤ y⌝ THEN1 asm_rewrite_tac[]);
a(list_spec_nth_asm_tac 3 [⌜x⌝, ⌜y⌝]);
pop_thm();
```

SML
```
set_goal([],                              (* (ii)(b) *)
 ⌜(∀x y•f x ≤ f y ⇒  x ≤ y) ⇒ (∀x y•f x = f y ⇒ x ≤ y)⌝);
a(REPEAT strip_tac);
a(lemma_tac⌜f x ≤ f y⌝ THEN1 asm_rewrite_tac[]);
a(list_spec_nth_asm_tac 3 [⌜x⌝, ⌜y⌝]);
pop_thm();
```

14.11 Forward Chaining

SML
```
set_goal([], ⌜∀a b c d•a ≤ b ∧ b ≤ c ∧ c ≤ d ⇒ a ≤ d⌝);
                                          (* 1(a) *)
a(REPEAT strip_tac);
a(all_fc_tac[≤_trans_thm] THEN all_fc_tac[≤_trans_thm]);
pop_thm();
```

SML
```
set_goal([], ⌜∀X Y Z•X ⊆ Y ∧ Y ⊆ Z ⇒ X ⊆ Z⌝);      (* 1(b) *)
a(REPEAT strip_tac);
a(all_asm_fc_tac[] THEN all_asm_fc_tac[]);
pop_thm();
```

In both cases, at least 2 applications of forward chaining are needed since a result from one forward chaining pass must be added to the assumptions to "seed" the second pass.

SML
```
set_goal([],                              (* 2 *)
 ⌜(∀x y•f x ≤ f y ⇒  x ≤ y) ⇒ (∀x y•f x = f y ⇒ x ≤ y)⌝);
a(REPEAT strip_tac);
a(lemma_tac ⌜f x ≤ f y⌝ THEN1 asm_rewrite_tac[]);
a(all_asm_fc_tac[]);
pop_thm();
```

14.12 Proof Contexts

SML
```
                                          (* 1 *)
set_goal([], ⌜(∀x y•f(x, y) = (y, x)) ⇒ ∀x y•f(f (x, y)) = (x, y)⌝);
a(REPEAT strip_tac);
(* *** Goal "1" *** *)
a(asm_rewrite_tac[]);
(* *** Goal "2" *** *)
a(asm_rewrite_tac[]);
pop_thm();
```

SML
```
set_pc"predicates";
set_goal([], ⌜(∀x y•f(x, y) = (y, x)) ⇒ ∀x y•f(f (x, y)) = (x, y)⌝);
a(REPEAT strip_tac);
a(asm_rewrite_tac[]);
pop_thm();
set_pc"hol2";
```

The second proof is shorter because the proof context *predicates* does not cause equations between pairs to be split into pairs of equations.

SML
```
                                        (∗ 2 ∗)
map (merge_pcs_rule1["hol2", "lin_arith"] prove_rule[]) [
(∗ (a) ∗)      ⌜{(x, y) | ¬x = 0 ∧ y = 2∗x} ⊆ {(x, y) | x < y}⌝,
(∗ (b) ∗)      ⌜{(x, y) | x ≥ 2 ∧ y = 2∗x} ⊆ {(x, y) | x + 1 < y}⌝,
(∗ (d) ∗)      ⌜∀m•{i | m ≤ i ∧ i < m + 3} = {m; m+1; m+2}⌝,
(∗ (e) ∗)      ⌜{i | 5∗i = 6∗i} = {0}⌝];
(∗ (c) ∗)      pc_rule1 "sets_ext1" prove_rule[]
               ⌜A ∪ (B ∩ C) = (A ∪ B) ∩ (A ∪ C)⌝;
```

(Alternatively, use the subgoal package and *PC_T1*.).

14.13 Case Study

The following test cases check out each branch of the *if*-terms in the definition of *vm*:

Branch 1: out of stock: the machine refunds any cash tendered.

SML
```
run_vm ⌜vm (MkVM_State t 0 p ct)⌝;
```

Branch 2: in stock; cash tendered is less than the price: the machine waits for more cash to be tendered:

SML
```
run_vm ⌜vm (MkVM_State t 20 5 2)⌝;
```

Branch 3: in stock; cash tendered is equal to the price: the machine dispenses a chocolate bar and adds the cash tendered to its takings:

SML
```
run_vm ⌜vm (MkVM_State t 20 5 5)⌝;
```

Branch 4: in stock; cash tendered exceeds the price: the machine refunds the cash tendered:

SML
```
run_vm ⌜vm (MkVM_State t 20 5 6)⌝;
```

If the price is set to *0*, the machine first refunds any cash tendered and then gives away all the stock!

SML
```
run_vm ⌜vm (MkVM_State t 4 0 6)⌝;
run_vm ⌜vm (MkVM_State t 4 0 0)⌝;
run_vm ⌜vm (MkVM_State t 3 0 0)⌝;
run_vm ⌜vm (MkVM_State t 2 0 0)⌝;
run_vm ⌜vm (MkVM_State t 1 0 0)⌝;
run_vm ⌜vm (MkVM_State t 0 0 0)⌝;
```

Now we show a proof against the ‘critical requirements’.

SML
```
set_goal([], ⌜vm ∈ vm_ok⌝);
(* Goal "": Expand definitions and let−terms: *)
a(rewrite_tac [get_spec ⌜vm_ok⌝, get_spec⌜vm⌝,
              get_spec⌜MkVM_State⌝, let_def]);

(* Goal "": remove outer universal quantifiers *)
a(REPEAT strip_tac);

(* Goal "": case split on the amount of stock:
                        s = 0 ∨ s = i + 1 for some i *)
a(strip_asm_tac(∀_elim⌜s⌝ ℕ_cases_thm) THEN asm_rewrite_tac[]);

(* Goal "1": s = 0 *)
a(asm_rewrite_tac[get_spec⌜value⌝, get_spec⌜MkVM_State⌝]);

(* Goal "2": case split on ct < p: ct < p ∨ ct = p ∨ p < ct *)
a(strip_asm_tac(list_∀_elim[⌜ct⌝, ⌜p⌝] less_cases_thm));

(* Goal "2.1": ct < p: *)
a(asm_rewrite_tac[get_spec⌜MkVM_State⌝]);

(* Goal "2.2": ct = p: *)
a(asm_rewrite_tac[get_spec⌜value⌝, get_spec⌜MkVM_State⌝]);
a(PC_T1 "lin_arith" asm_prove_tac[]);

(* Goal "2.3": ct > p: need ¬ct < p ∧ ¬ ct = p to evaluate if *)
a(lemma_tac ⌜¬ct < p ∧ ¬ ct = p⌝ THEN1
      PC_T1 "lin_arith" asm_prove_tac[]);
a(asm_rewrite_tac[get_spec⌜value⌝, get_spec⌜MkVM_State⌝]);

val vm_ok_thm = pop_thm();
```

14.14 Advanced Techniques

For running these solutions in batch we need to ask for the warning arising from deletion of constants to be ignored:

SML
```
set_flag("ignore_warnings", true);
```

SML

```
(* no. 1 *)
set_goal([], ⌜∀m•∃n• m < n⌝);
a(contr_tac);
a(spec_asm_tac⌜∀ n• ¬ m < n⌝⌜m+1⌝);
val thm1 = pop_thm();
```

SML

```
(* no. 2 *)
set_goal([], ⌜∀m•∃n• m < n⌝);
a(REPEAT strip_tac);
a(∃_tac⌜m+1⌝);
a(rewrite_tac[]);
val thm2 = pop_thm();
```

SML

```
(* no. 3 *)
set_goal([], ⌜∀f : ℕ → (ℕ → ℕ)•∃g•∀i•¬f i = g⌝);
a(REPEAT strip_tac);
a(∃_tac⌜λj•(f j j) + 1⌝);
a(rewrite_tac[ext_thm]);
a(REPEAT strip_tac);
a(∃_tac⌜i⌝ THEN REPEAT strip_tac);
val thm3 = pop_thm();
```

SML

```
(* no. 4 *)
set_goal([], ⌜∀f:'a→'b→'a•(∀x y•x = f x y) ⇒ f = CombK⌝);
a (REPEAT strip_tac);
a (rewrite_tac[ext_thm, comb_k_def]);
a (swap_asm_concl_tac⌜∀ x y• x = f x y⌝);
a (conv_tac(ONCE_MAP_C eq_sym_conv));
a (swap_asm_concl_tac⌜¬ f x x' = x⌝ THEN asm_rewrite_tac[]);
val thm4 = pop_thm();
```

SML

```
(* no. 5 *)
set_goal([], ⌜∀i j•0 < i ⇒ (i * j) Div i = j⌝);
a (REPEAT strip_tac);
a (strip_asm_tac(
        rewrite_rule[∀_elim⌜j⌝times_comm_thm]
        (list_∀_elim[⌜i*j⌝, ⌜i⌝, ⌜j⌝, ⌜0⌝] div_mod_unique_thm)));
a (swap_asm_concl_tac⌜j = (i * j) Div i⌝ THEN
                (conv_tac(ONCE_MAP_C eq_sym_conv)));
a (strip_tac);
val thm5 = pop_thm();
```

SML
```
(* no. 6 *)
set_goal([], ⌜∀i•σ i = (i*(i + 1)) Div 2⌝);
a (REPEAT strip_tac);
a (lemma_tac⌜i * (i + 1) = 2 * σ i⌝);
(* *** Goal "1" *** *)
a (induction_tac⌜i⌝ THEN asm_rewrite_tac[get_spec⌜σ⌝]);
a(PC_T1 "lin_arith" asm_prove_tac[]);
(* *** Goal "2" *** *)
a (asm_rewrite_tac[rewrite_rule[](list_∀_elim[⌜2⌝, ⌜σ i⌝]thm5)]);
val thm6 = pop_thm();
```

SML
```
(* no. 7 *)
```

The obvious way of defining the Fibonacci function is not automatically proved consistent:

SML
```
delete_const⌜φ⌝;
```

HOL Constant
```
        φ : ℕ → ℕ
─────────────────────────────
        φ 0 = 0
∧       φ 1 = 1
∧       ∀i•φ(i+2) = φ(i+1) + φ i
```

SML
```
get_spec⌜φ⌝;
```

SML
```
delete_const⌜φ⌝;
(* no. 8 *)
```

HOL Constant
```
        γ : ℕ → (ℕ × ℕ)
─────────────────────────────
        γ 0 = (0, 1)
∧       ∀i•γ(i+1) = let (a, b) = γ i in (b, a + b)
```

HOL Constant
```
        φ : ℕ → ℕ
─────────────────────────────
        ∀i•φ i = Fst (γ i)
```

SML
```
set_goal([], ⌜
        ϕ 0 = 0
∧       ϕ 1 = 1
∧       ∀i•ϕ(i+2) = ϕ(i+1) + ϕ i
⌝);
a (rewrite_tac[get_spec⌜ϕ⌝]);
a (lemma_tac⌜γ 1 = γ(0 + 1) ∧ ∀i• γ(i + 2) = γ((i+1)+1)⌝);
(* *** Goal "1" *** *)
a (rewrite_tac[plus_assoc_thm]);
(* *** Goal "2" *** *)
a (pure_asm_rewrite_tac[get_spec⌜γ⌝, let_def] THEN rewrite_tac[]);
val thm8 = pop_thm();
```

SML
```
(* no. 9 *)
delete_const⌜γ⌝;
```

HOL Constant
```
        ϕ : ℕ → ℕ
──────────────────────────────────
        ϕ 0 = 0
 ∧      ϕ 1 = 1
 ∧      ∀i•ϕ(i+2) = ϕ(i+1) + ϕ i
```

SML
```
get_spec⌜ϕ⌝;
push_consistency_goal⌜ϕ⌝;
a (lemma_tac⌜∃γ•
        γ 0 = (0, 1)
∧       ∀i•γ(i+1) = let (a, b) = γ i in (b, a + b)
⌝);
(* *** Goal "1" *** *)
a (prove_∃_tac);
(* *** Goal "2" *** *)
a (∃_tac⌜λi•Fst(γ i)⌝);
a (rewrite_tac[]);
a (lemma_tac⌜γ 1 = γ(0 + 1) ∧ ∀i• γ(i + 2) = γ((i+1)+1)⌝);
(* *** Goal "2.1" *** *)
a (rewrite_tac[plus_assoc_thm]);
(* *** Goal "2.2" *** *)
a (pure_asm_rewrite_tac[let_def] THEN asm_rewrite_tac[]);
save_consistency_thm ⌜ϕ⌝ (pop_thm());
get_spec⌜ϕ⌝;
```

14.15 Supplementary Exercises

14.15.1 Linear Arithmetic

SML

```
(open_theory "usr013" handle _ => (open_theory "hol"; new_theory "usr013"));
new_theory "Shankar−Rushby−International";

set_pc "hol2";

set_goal([], ⌜∀ i:ℕ• i>7 ⇒ ∃ three five:ℕ• 3∗three + 5∗five = i⌝);
```

ProofPower output

```
Now 1 goal on the main goal stack

(∗ ∗∗∗ Goal "" ∗∗∗ ∗)

(∗ ?⊢ ∗)  ⌜∀ i• i > 7 ⇒ (∃ three five• 3 ∗ three + 5 ∗ five = i)⌝
```

First we do induction on ⌜i⌝ rewriting the resulting conclusions with the assumptions. This solves the induction base case and leaves two subgoals relating to the step case.

SML

```
a (strip_tac THEN induction_tac ⌜i:ℕ⌝ THEN asm_rewrite_tac[]);
```

ProofPower output

```
Tactic produced 2 subgoals:

(∗ ∗∗∗ Goal "2" ∗∗∗ ∗)

(∗  1 ∗)  ⌜3 ∗ three + 5 ∗ five = i⌝

(∗ ?⊢ ∗)  ⌜7 < i + 1 ⇒ (∃ three five• 3 ∗ three + 5 ∗ five = i + 1)⌝


(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)

(∗  1 ∗)  ⌜¬ 7 < i⌝

(∗ ?⊢ ∗)  ⌜7 < i + 1 ⇒ (∃ three five• 3 ∗ three + 5 ∗ five = i + 1)⌝
```

Now we prove the lemma ⓩi = 7⌝ by linear arithmetic and rewrite with it.

SML

```
(∗ ∗∗∗ Goal "1" ∗∗∗ ∗)
a (strip_tac THEN LEMMA_T ⌜i = 7⌝ rewrite_thm_tac
  THEN1 PC_T1 "lin_arith" asm_prove_tac[]);
```

ProofPower output
```
Tactic produced 1 subgoal:

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

(*  2 *)  ⌜¬ 7 < i⌝
(*  1 *)  ⌜7 < i + 1⌝

(* ?⊢ *)  ⌜∃ three five• 3 * three + 5 * five = 8⌝
```

Next we supply the obvious witness for this existential and prove the resulting subgoal automatically.

SML
```
a (MAP_EVERY ∃_tac [⌜1⌝,⌜1⌝] THEN prove_tac[]);
```

ProofPower output
```
Tactic produced 0 subgoals:
Current goal achieved, next goal is:

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

(*  1 *)  ⌜3 * three + 5 * five = i⌝

(* ?⊢ *)  ⌜7 < i + 1 ⇒ (∃ three five• 3 * three + 5 * five = i + 1)⌝
```

We now eliminate the variable ⌜i⌝ and strip down to the existential:

SML
```
(* *** Goal "2" *** *)
a (all_var_elim_asm_tac1 THEN strip_tac);
```

ProofPower output
```
Tactic produced 1 subgoal:

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

(*  1 *)  ⌜7 < (3 * three + 5 * five) + 1⌝

(* ?⊢ *)  ⌜∃ three′ five′
            • 3 * three′ + 5 * five′ = (3 * three + 5 * five) + 1⌝
```

Then we do a case split on whether *five* is zero:

SML
```
a (strip_asm_tac (∀_elim ⌜five⌝ ℕ_cases_thm)
        THEN asm_rewrite_tac[]);
```

ProofPower output
```
Tactic produced 2 subgoals:

(∗ ∗∗∗ Goal "2.2" ∗∗∗ ∗)

(∗  2 ∗)  ⌜7 < (3 ∗ three + 5 ∗ five) + 1⌝
(∗  1 ∗)  ⌜five = i + 1⌝

(∗ ?⊢ ∗)  ⌜∃ three′ five′
            • 3 ∗ three′ + 5 ∗ five′ = (3 ∗ three + 5 ∗ (i + 1)) + 1⌝


(∗ ∗∗∗ Goal "2.1" ∗∗∗ ∗)

(∗  2 ∗)  ⌜7 < (3 ∗ three + 5 ∗ five) + 1⌝
(∗  1 ∗)  ⌜five = 0⌝

(∗ ?⊢ ∗)  ⌜∃ three′ five′• 3 ∗ three′ + 5 ∗ five′ = 3 ∗ three + 1⌝
```

In this case it is most convenient first to prove that ⌜*three*≥*3*⌝ using linear arithmetic, and then rewrite this result to give a witness 3 less than *three*:

SML
```
(∗ ∗∗∗ Goal "2.1" ∗∗∗ ∗)
a (LEMMA_T ⌜three ≥ 3⌝ (strip_asm_tac o rewrite_rule [≤_def])
 THEN1 PC_T1 "lin_arith" asm_prove_tac[]);
```

ProofPower output
```
Tactic produced 1 subgoal:

(∗ ∗∗∗ Goal "2.1" ∗∗∗ ∗)

(∗  3 ∗)  ⌜7 < (3 ∗ three + 5 ∗ five) + 1⌝
(∗  2 ∗)  ⌜five = 0⌝
(∗  1 ∗)  ⌜3 + i = three⌝

(∗ ?⊢ ∗)  ⌜∃ three′ five′• 3 ∗ three′ + 5 ∗ five′ = 3 ∗ three + 1⌝
```

The witness ⌜i⌝ is now used in the existence proof:

SML
```
a (MAP_EVERY ∃_tac [⌜i⌝, ⌜five +2⌝]
 THEN PC_T1 "lin_arith" asm_prove_tac[]);
```

ProofPower output

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

(* *** Goal "2.2" *** *)

(*  2 *)  ⌜7 < (3 * three + 5 * five) + 1⌝
(*  1 *)  ⌜five = i + 1⌝

(* ?⊢ *)  ⌜∃ three′ five′
             • 3 * three′ + 5 * five′ = (3 * three + 5 * (i + 1)) + 1⌝
```

This subgoal is proved in a similar way but with different witnesses:

SML

```
(* *** Goal "2.2" *** *)
a (MAP_EVERY ∃_tac [⌜three+2⌝, ⌜i⌝]
  THEN PC_T1 "lin_arith" asm_prove_tac[]);
```

ProofPower output

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

SML

```
save_pop_thm("cents_thm");
```

ProofPower output

```
Now 0 goals on the main goal stack
val it = ⊢ ∀ i• i > 7 ⇒ (∃ three five• 3 * three + 5 * five = i) : THM
```

REFERENCES

[1] A.N.Whitehead and B.Russell. *Principia Mathematica.* Cambridge University Press, 1910. 3 vols.

[2] Alonzo Church. A Formulation of the Simple Theory of Types. *Journal of Symbolic Logic*, 5:56–, 1940.

[3] Michael J.C. Gordon. HOL:A Proof Generating System for Higher-Order Logic. In G. Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis.* Kluwer, 1987.

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

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

[6] Donald E. Knuth. Literate Programming. *The Computer Journal*, 27(2):97–111, 1984.

[7] L.Paulson. A Higher-order Implementation of Rewriting. *Science of Computer Programming*, 3:119–149, 1983.

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

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

[10] B. Russell. Mathematical Logic as based on the Theory of Types. *American Journal of Mathematics*, 30:222–262, 1908.

[11] J.M. Spivey. *The Z Notation: A Reference Manual, Second Edition.* Prentice-Hall, 1992.

[12] Jim Woodcock and Martin Loomes. *Software Engineering Mathematics.* Pitman, 1988.

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

[14] DS/FMU/IED/USR004. *ProofPower Tutorial Manual.* Lemma 1 Ltd., `http://www.lemma-one.com`.

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

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

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

[18] ds/fmu/ied/usr022. *ProofPower HOL Tutorial Transparencies.* Lemma 1 Ltd., `http://www.lemma-one.com`.

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

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

INDEX