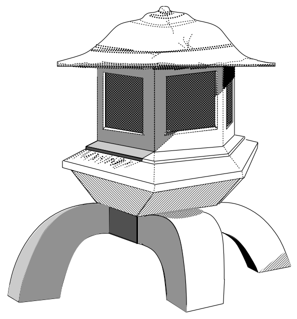
[For HOL-Omega Kananaskis-8] February 9, 2013

The HOL-Omega System TUTORIAL



Preface

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

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

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

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

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

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

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

Getting started

Experienced HOL users who are eager to get started with HOL-Omega will want to read chapters 1, 4, 5, and 10 through 12. Others who are not familiar with HOL should read chapters 1 through 5 and then select chapters with examples that match their interests. Chapter 1 explains how to get and install HOL-Omega. Once this is done, the potential HOL-Omega user should become familiar with the following subjects:

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

Chapters 2 and 3 introduce these topics. Then the new, additional concepts and features of the HOL-Omega logic (higher order logic extended with System *F*, kinds, and ranks) are first casually demonstrated, as an appetizer, and then discussed in more detail, in chapters 4 and 5. After this, a series of examples are presented using the HOL-Omega system. Chapters 6 through 9 use only the classical higher order logic subset of HOL-Omega, and are the same as for the existing HOL theorem prover. The new, extended features of HOL-Omega are demonstrated in examples in chapters 10 through 12.

Chapter 6 develops an extended example — Euclid's proof of the infinitude of primes— to illustrate how HOL-Omega is used to prove theorems.

Chapter 7 features another worked example: the specification and verification of a simple sequential parity checker. The intention is to accomplish two things: (i) to present another complete piece of work with HOL-Omega; and (ii) to give an idea of what it is like to use the HOL-Omega system for a tricky proof. Chapter 8 is a more extensive example: the proof of confluence for combinatory logic. Again, the aim is to present a complete piece of non-trivial work.

Chapter 9 gives an example of implementing a proof tool of one's own. This demonstrates the programmability of HOL-Omega: the way in which technology for solving specific problems can be implemented on top of the underlying kernel. With high-powered tools to draw on, it is possible to write prototypes very quickly.

Chapter 10 shows how one can construct an abstract data type in the HOL-Omega logic, where a new type representing an abstract algebra can be created where the only thing known about it is its defining property. This property might not completely specify its behavior in all circumstances, which allows for under-specification of the new type. This is a means for information hiding and proper modularization of large proofs.

Chapter 11 exercises the kind system of the new logic, exploiting the ability to have type variables which represent type operators, as well as the ability to quantify type variables over expressions, to explore some of the concepts of category theory, defining functors and natural transformations as a shallow embedding in the HOL-Omega logic, and showing how they may be combined in a fluid way.

Chapter 12 gives a thorough discussion and exercise of existential types and packages. It shows how these can be used to support in a practical and principled way good software engineering practices such as modularity and information hiding.

Chapter 13 briefly discusses some of the examples distributed with HOL-Omega in the `examples` directory.

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

Acknowledgements

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

Current edition

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

The material in the third edition constitutes a thorough re-working and extension of previous editions. The semantics by Andy Pitts (in *LOGIC*) had remained essentially unaltered until HOL-Omega, reflecting the fact that, although the HOL system has undergone continual development and improvement, the HOL logic had been unchanged since the first edition (1988) until the introduction of HOL-Omega in 2009.

Finally (*Peter Homeier writing*), I would like to give thanks to the Lord God Almighty for His inspiration, wisdom, and revelations. HOL-Omega is a gift from His hand.

Soli Deo Gloria.

Second edition

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

First edition

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

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

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

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

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

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

Contents

1 Getting and Installing HOL-Omega 11

1.1 Getting HOL-Omega 11

1.2 The `hol-info` mailing list 11

1.3 Installing HOL-Omega 12

1.3.1 Overriding `smart-configure` 14

2 Introduction to ML 17

2.1 How to interact with ML 17

3 The HOL Logic 21

3.1 Proof in HOL 27

3.2 Forward proof 29

3.2.1 Derived rules 31

3.3 Goal Oriented Proof: Tactics and Tacticals 34

3.3.1 Using tactics to prove theorems 38

3.3.2 Tacticals 39

3.3.3 Some tactics built into HOL 42

4 HOL-Omega Appetizers 47

4.1 Collections 47

4.1.1 Object-oriented collections 49

4.1.2 Fold operation 50

4.1.3 Map operation 55

4.1.4 Abstract collections 59

5 The HOL-Omega Logic 65

5.1 New notation 65

5.2 Proof in HOL-Omega 92

5.2.1 Primitive rules of inference 92

5.2.2 New axioms 95

5.2.3 Basic derived rules of inference 97

5.3 Backwards Proof 102

5.3.1 New Basic Tactics 103

5.3.2 Extended Tactics 105
5.3.3 Other Rules, Tactics, and Automation 106
5.4 Backwards Compatibility 106

6 Example: Euclid's Theorem 109
6.1 Divisibility 111
6.1.1 Divisibility and factorial 118
6.1.2 Divisibility and factorial (again!) 124
6.2 Primality 129
6.3 Existence of prime factors 129
6.4 Euclid's theorem 133
6.5 Turning the script into a theory 136
6.6 Summary 138

7 Example: a Simple Parity Checker 141
7.1 Introduction 141
7.2 Specification 142
7.3 Implementation 145
7.4 Verification 148
7.5 Exercises 152
7.5.1 Exercise 1 152
7.5.2 Exercise 2 153

8 Example: Combinatory Logic 155
8.1 Introduction 155
8.2 The type of combinators 155
8.3 Combinator reductions 155
8.4 Transitive closure and confluence 157
8.5 Back to combinators 168
8.5.1 Parallel reduction 168
8.5.2 Using RTC 169
8.5.3 Proving the RTCs are the same 170
8.5.4 Proving a diamond property for parallel reduction 175
8.6 Exercises 182

9 Proof Tools: Propositional Logic 185
9.1 Method 1: Truth Tables 185
9.2 Method 2: the DPLL Algorithm 186
9.2.1 Conversion to Conjunctive Normal Form 189
9.2.2 The Core DPLL Procedure 191
9.2.3 Performance 196

9.3 Extending our Procedure's Applicability 196

10 Example: Abstract Data Types 199
10.1 New term and type constants 201
10.1.1 New term constant definition 201
10.1.2 New term constant specification 202
10.1.3 New type constant definition 203
10.1.4 New type constant specification 204
10.2 Bit Vectors 205
10.2.1 Defining a new type 207
10.2.2 Abstraction and representation bijections 208
10.2.3 Defining new term constants of the new type 210
10.2.4 Induction on the new type 210
10.2.5 Combinator for recursive functions 212
10.2.6 Proof of initiality 215
10.2.7 Existence and creation of bit vector type 222
10.2.8 Creation of bit vector constants 224
10.2.9 Bit vectors as an abstract data type 225

11 Example: The Category of Types 227
11.1 Categories 227
11.2 Functors 228
11.2.1 Examples of functors 231
11.2.2 Composition of functors 235
11.3 Natural Transformations 238
11.3.1 Examples of Natural Transformations 239
11.3.2 Vertical composition of natural transformations 242
11.3.3 Horizontal composition of natural transformations 243
11.3.4 Composition of natural transformations with functors 245
11.4 Algebras and Initial Algebras 247
11.5 Catamorphisms 252

12 Example: Packages 257
12.1 Existential Types 258
12.2 Packages 260
12.3 Underlying Implementation of Packages 264
12.3.1 Package Axioms 264
12.4 Example: Counters 265
12.5 Example: Scheduling Queues 267

13 More Examples 291

Chapter 1

Getting and Installing HOL-Omega

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

1.1 Getting HOL-Omega

HOL-Omega is part of the HOL system. The HOL system has several branches; branch `HOL-Omega` contains the source of the HOL-Omega theorem prover. The naming scheme for HOL-Omega releases is ⟨*name*⟩-⟨*number*⟩; the release described here is Kananaskis-8.

HOL development uses the Git version control system. Git is freely available from `http://git-scm.com/`, and is well described in the book *ProGit* by Scott Chacon.

A fresh copy of the current developer version of HOL-Omega may be checked out into a fresh subdirectory called `hol-omega` by the following Git command:

```
git clone -b HOL-Omega git://github.com/mn200/HOL.git hol-omega
```

As a developer, to check out a fresh copy of HOL-Omega that one could edit and write back to the github repository, one would set up a SSH key with github and then do

```
git clone -b HOL-Omega git@github.com:mn200/HOL.git hol-omega
```

To set up a SSH key, see `https://help.github.com/articles/generating-ssh-keys`.

To become an HOL developer and have write access to the github repository, please contact one of the administrators listed at `http://sourceforge.net/projects/hol/`.

1.2 The `hol-info` mailing list

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

1.3 Installing HOL-Omega

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

Principal Files on the HOL-Omega Distribution Directory

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

The session in the box below shows a typical distribution directory. The HOL-Omega distribution has been placed in the directory `/Users/palantir/hol-omega/` on a Macintosh running OS X for a user `palantir`.

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

1
```
$ pwd
/Users/palantir/hol-omega
$ ls -F
COPYRIGHT   bin/          examples/   sigobj/       tools-poly/
INSTALL     cleanall*     help/       src/
Manual/     developers/   icon.gif*   std.prelude
README      doc/          merging     tools/
```

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

Now you will need to rebuild HOL-Omega from the sources.[2] Detailed installation instructions may be found at `http://hol.sourceforge.net/InstallKananaskis.html`.

Before beginning you must have a current version of Moscow ML or Poly/ML[3]. In the case of Moscow ML, you must have version 2.01. Moscow ML is available on the web from `http://www.dina.kvl.dk/~sestoft/mosml.html`. Poly/ML is available from `http://polyml.org`. When you have your ML system installed, and are in the root directory of the distribution, the next step is to run `smart-configure`. With Moscow ML, this looks like:

```
$ mosml < tools/smart-configure.sml                                    2
Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
- [opening file "tools/smart-configure-mosml.sml"]

HOL-Omega smart configuration.

Determining configuration parameters: OS mosmldir holdir
OS:                 macosx
mosmldir:           /Users/palantir/mosml/bin
holdir:             /Users/palantir/hol-omega
dynlib_available:   true

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

If you are using Poly/ML, then write

```
poly < tools/smart-configure.sml
```

instead.

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

The next step is to run the `build` program. This should result in a great deal of output as all of the system code is compiled and the theories built. Eventually, a HOL-Omega system[4] is produced in the `bin/` directory.

```
$ bin/build                                                            3
  ...
  ...
Uploading files to /Users/palantir/hol-omega/sigobj

Hol built successfully.
$
```

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

[3]Poly/ML cannot be used with HOL-Omega on Windows.

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

1.3.1 Overriding smart-configure

If smart-configure is unable to guess correct values for the various parameters (holdir, OS etc.) then you can create a file called to provide correct values. With Moscow ML, this should be config-override in the root directory of the HOL-Omega distribution. With Poly/ML, this should be poly-includes.ML in the tools-poly directory. In this file, specify the correct value for the appropriate parameter by providing an ML binding for it. All variables except dynlib_available must be given a string as a possible value, while dynlib_available must be either true or false. So, one might write

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

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

With this file in place, the smart-configure program will use the values specified there rather than those it attempts to calculate itself. The value given for the OS variable must be one of "unix", "linux", "solaris", "macosx" or "winNT".[5]

In extreme circumstances it is possible to edit the file tools/configure.sml yourself to set configuration variables directly. (If you are using Poly/ML, you must edit tools-poly/configure.sml instead.) At the top of this file various incomplete SML declarations are present, but commented out. You will need to uncomment this section (remove the (* and *) markers), and provide sensible values. All strings must be enclosed in double quotes.

The holdir value must be the name of the top-level directory listed in the first session above. The OS value should be one of the strings specified in the accompanying comment.

When working with Moscow ML, the mosmldir value must be the name of the directory containing the Moscow ML binaries (mosmlc, mosml, mosmllex etc). When working with Poly/ML, the poly string must be the path to the poly executable that begins an interactive ML session. The polymllibdir must be a path to a directory that contains the file libpolymain.a.

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

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

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

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

val CC:string       = "gcc";      (* C compiler                       *)
val GNUMAKE:string  = "make";     (* for bdd library and SMV          *)
val DEPDIR:string   = ".HOLMK";   (* where Holmake dependencies kept  *)
  ...
$
```

Now, at either this level (in the `tools` or `tools-poly` directory) or at the level above, the script `configure.sml` must be piped into the ML interpreter (i.e., `mosml` or `poly`). For example,

```
$ mosml < tools/configure.sml                                             6
Moscow ML version 2.01 (January 2004)
Enter ‘quit();’ to quit.
- > val mosmldir = "/home/palantir/mosml" : string
  val holdir = "/home/palantir/hol-omega" : string
  val OS = "linux" : string
- > val CC = "gcc" : string
  ...
Beginning configuration.
Removing old quotation filter from bin/
Making tools/mllex/mllex.exe
Making bin/Holmake.
  ...
Making bin/build.
Making hol-mode.el (for Emacs/XEmacs)
Generating bin/hol.
Generating bin/hol.noquote.
  ...
Number of states = 170
Number of distinct rows = 90
Approx. memory size of trans. table = 23040 bytes
Analysing filter.sml
Compiling filter.sml
Compiling quote-filter.sml
  ...
Analysing selftest.sml
Quote-filter built
Setting up the muddy library Makefile.

Finished configuration!
$
```

Chapter 2

Introduction to ML

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

2.1 How to interact with ML

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

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

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

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

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

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

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

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

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

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

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

```
$ bin/hol                                                          1

-----------------------------------------------------------------
       HOL-4 [Kananaskis 8 (built Fri Apr 12 15:34:35 2002)]

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

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

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

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

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

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

- hd it;
> val it = 2 : int

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 3

The HOL Logic

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

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

The HOL-Omega system is an extension of the HOL system. It is backwards compatible, so that virtually anything that can be done in HOL can also be done in exactly the same way in HOL-Omega. No logical power is lost. Rather, HOL-Omega goes beyond the logic of HOL with two conceptual extensions:

- Types can be abstracted by type variables (similar to how terms are abstracted by term variables in the lambda calculus).
  - Type operators are curried, so that they may take one argument at a time.
  - Every type has a *kind*; kinds manage type applications just as types manage term applications.
  - Type variables can represent type operators.
- Terms can be abstracted by type variables (similar to System *F*).
  - The type of such an abstraction is a *universal* type.
  - Such an abstraction may be applied as a function to a type argument.
  - Such applications are managed by classifying all types by *ranks*.

Since HOL-Omega is backwards compatible, it makes sense to learn HOL-Omega in two stages; first to study just the classical higher order logic of HOL, and then add to that the extensions provided by HOL-Omega. This chapter will concentrate on the HOL subset, and the next chapter will discuss the full HOL-Omega logic.

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

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

Terms of the HOL Logic

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

Terms of the HOL logic are represented in ML by an *abstract type* called `term`. They are normally input between double back-quote marks. For example, the expression ``` ``x /\ y ==> z`` ``` evaluates in ML to a term representing `x`$\wedge$`y`$\Rightarrow$`z`. Terms can be manipulated by various built-in ML functions. For example, the ML function `dest_imp` with ML type `term -> term * term` splits an implication into a pair of terms consisting of its antecedent and consequent, and the ML function `dest_conj` of type `term -> term * term` splits a conjunction into its two conjuncts. [1]

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

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

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

Terms of the HOL logic are quite similar to ML expressions, and this can at first be confusing. Indeed, terms of the logic have types similar to those of ML expressions. For

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

example, ``(1,2)`` is an ML expression with ML type term. The HOL type of this term is num # num. By contrast, the ML expression (``1``, ``2``) has type term * term.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(1 :num)

to

(2 :num)

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

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

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

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

$~

to

(1 :num)

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.1 Proof in HOL

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Exception raised at Thm.SPEC:

! Uncaught exception:
! HOL_ERR
```

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

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

The general form of a theorem is $t_1, \ldots, t_n$ |- t, where t_1, … , t_n are boolean terms called the *assumptions* and t is a boolean term called the *conclusion*. Such a theorem asserts that if its assumptions are true then so is its conclusion. Its truth conditions

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

are thus the same as those for the single term $(t_1$/\.../\$t_n)$==>$t$. Theorems with no assumptions are printed out in the form `|-` t.

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

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

3.2 Forward proof

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

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

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

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

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

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

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

[4]This is a slight over-simplification.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

3.2.1 Derived rules

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

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

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

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

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

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

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

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

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

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

The ML definition is:

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

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

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

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

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

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

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

An ML derived rule that implements this is:

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

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

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

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

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

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

3.2.1.1 Rewriting

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

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

and a theorem $\Delta \vdash t$ and repeatedly replaces instances of u_i in t by the corresponding instance of v_i until no further change occurs. The result is a theorem $\Gamma \cup \Delta \vdash t'$ where t' is the result of rewriting t in this way. The session below illustrates the use of `REWRITE_RULE`. In it the list of equations is the value `rewrite_list` containing the pre-proved theorems `ADD_CLAUSES` and `MULT_CLAUSES`. These theorems are from the theory `arithmetic`, so we must use a fully qualified name with the name of the theory as the first component to refer to them. (Alternatively, we could, as in the Euclid example of section 6, use `open` to bring declare all of the values in the theory at the top level.)

```
- open arithmeticTheory;                                          15

   ...

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

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

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

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

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

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

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

3.3 Goal Oriented Proof: Tactics and Tacticals

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

achieves a goal

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

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

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

Tactics are specified using the following notation:

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

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

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

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

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

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

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

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

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

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

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

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

3.3.1 Using tactics to prove theorems

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

val (gl,p) = T g

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

val th = p[]

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

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

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

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

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

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

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

When conducting a proof that involves many subgoals and tactics, it is necessary to keep track of all the justification functions and compose them in the correct order. While this is feasible even in large proofs, it is tedious. HOL provides a package for building and traversing the tree of subgoals, stacking the justification functions and applying them properly; this package was originally implemented for LCF by Larry Paulson. Its use is demonstrated in Chapter 6, and thoroughly documented in *DESCRIPTION*.

3.3.2 Tacticals

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

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

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

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

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

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

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

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

3.3.2.2 THEN : tactic -> tactic -> tactic

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

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

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

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

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

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

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

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

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

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

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

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

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

3.3.2.4 `REPEAT : tactic -> tactic`

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

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

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

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

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

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

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

[7]This is a minor simplification.

3.3.3 Some tactics built into HOL

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

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

3.3.3.1 `REWRITE_TAC : thm list -> tactic`

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

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

  where t' is obtained from t by rewriting with

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

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

- **Other rewriting tactics**:

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

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

3.3.3.2 `CONJ_TAC : tactic`

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

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

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

3.3.3.3 `EQ_TAC : tactic`

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

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

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

3.3.3.4 `DISCH_TAC : tactic`

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

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

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

3.3.3.5 `GEN_TAC : tactic`

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

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

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

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

3.3.3.6 `PROVE_TAC : thm list -> tactic`

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

3.3.3.7 `STRIP_TAC : tactic`

- **Summary:** Breaks a goal apart. `STRIP_TAC` removes one outer connective from the goal, using `CONJ_TAC`, `DISCH_TAC`, `GEN_TAC`, etc. If the goal is t_1`/\`$\cdots$`/\`t_n `==>` t then `STRIP_TAC` makes each t_i into a separate assumption.
- **Uses:** Useful for splitting a goal up into manageable pieces. Often the best thing to do first is `REPEAT STRIP_TAC`.

3.3.3.8 `ACCEPT_TAC : thm -> tactic`

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

  `T THENL[ACCEPT_TAC` th_1`,...,ACCEPT_TAC` th_n`]`

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

3.3.3.9 `ALL_TAC : tactic`

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

3.3.3.10 `NO_TAC : tactic`

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

Chapter 4

HOL-Omega Appetizers

This chapter will introduce the HOL-Omega logic, with the idea of motivating it by a series of examples. These examples are only discussed superficially, to showcase the new ideas, and not all details are pursued. A more complete description of the HOL-Omega extensions is provided in the next chapter in the tutorial. But these are presented as appetizers, to lightly show how the new features might be used to good effect.

4.1 Collections

To begin, HOL is blessed with a number of different types in the logic that represent different varieties of collections, like lists, sets, and bags. These are polymorphic types, written e.g. α `list`, where α is the type of the elements of the list. All these collections are similar, in that they all have an empty collection, they all have a way to insert a new element into a collection, they all have a way to measure the size of a collection, etc.

Suppose one wanted to represent the notion of a collection as an abstraction of the normal notions of a set or a list. In HOL there is no natural way to do this, but in HOL-Omega one could use a type operator variable to stand for the various collection types, and then create a record of some of the normal functions used on collections, as follows.

```
- new_theory "appetizers";                                        3
<<HOL message: Created theory "appetizers">>
> val it = () : unit
> set_trace "Unicode" 0;
val it = () : unit

- Hol_datatype `collection_ops =
     <| empty  : 'x 'col;
        insert : 'x -> 'x 'col -> 'x 'col;
        length : 'x 'col -> num |>`;
<<HOL message: Defined type: "collection_ops">>
> val it = () : unit
```

Here we have used the type variable `'col` as a variable to stand for the type operator we are talking about, whether `list`, `option`, or some other type. In HOL, type variables can only stand for entire types, like `num list`, but not type operators like just `list`. But here, `'col` is being used as a function on types, that takes a type `'x`, the type of the elements of the collection, and returns a type `'x 'col`, the type of collections of such elements. Such type operator variables are one of the new features of HOL-Omega.

Both `'col` and `'x` are free type variables in this definition, so the type being defined takes two arguments, e.g., `('col, 'x)collection_ops`. The order of the two arguments is by alphabetical order.

Now we can describe lists as collections:

```
- val list_ops = Define                                                          4
     `list_ops = <|empty := []:'a list; insert := CONS; length := LENGTH|>`;
Definition has been stored under "list_ops_def"
> val list_ops =
    |- list_ops = <|empty := []; insert := CONS; length := LENGTH|> : thm

- type_of ``list_ops``;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    ``:(list, 'a) collection_ops``
     : hol_type
```

The type of this collection is `(list, 'a) collection_ops`. The first argument is the type `list`, here being used without any type argument of its own. This is meaningful in HOL-Omega, although it may look weird to HOL users who are used to always seeing `list` with an argument, like `num list` or `'a list`. But here `list` is itself an argument, albeit a type operator alone, replacing `'col` in the definition of `collection_ops` above.

Here are sets described as collections:

```
- val set_ops = Define                                                           5
     `set_ops = <|empty := {}:'a set; insert := $INSERT; length := CARD|>`;
Definition has been stored under "set_ops_def"
> val set_ops = |- set_ops = <|empty := {}; insert := $INSERT; length := CARD|>
     : thm

- type_of ``set_ops``;
<<HOL message: inventing new type variable names: 'b>>
> val it =
    ``:(\'a. 'a -> bool, 'b) collection_ops``
     : hol_type
```

Note that the first argument to this set collection type is `\'a. 'a -> bool`. This is an *abstraction type*, similar to the normal lambda abstraction in terms, but this abstraction is within the type language of HOL-Omega. The scope of the lambda binding of `'a` in the type above is up to but not including the comma, which ends the first type argument.

But, you may ask, why does this type abstraction `\'a. 'a -> bool` appear in this collection type? The reason is that the type of sets in HOL, `'a set`, is actually a type abbreviation, not a real type. It is a feature of the parser and prettyprinter, not the actual logic as such. The abbreviation `'a set` stands for the real type `'a -> bool`. The

HOL-Omega system figures out the appropriate type to substitute for the type argument `'col` to create the type `'a -> bool`, and the substitution is [`'col` ↦ `\'a. 'a -> bool`]. The type resulting from the substitution is `'a (\'a. 'a -> bool)` (in postfix notation), which is equivalent to `'a -> bool` through type beta-reduction.

HOL contains not only lists and sets, but also bags, which are sometimes called multisets. Bags are like sets which can include multiple copies of its elements, whereas sets can only contain a single copy of each. Here are bags described as collections:

```
- load "bagLib";                                                        6
...
- val bag_ops = Define
     `bag_ops = <| empty := {||}:'a bag; insert := BAG_INSERT;
                  length := BAG_CARD|>`;
Definition has been stored under "bag_ops_def"
> val bag_ops =
    |- bag_ops = <|empty := {||}; insert := BAG_INSERT; length := BAG_CARD|> :
  thm

- type_of ``bag_ops``;
<<HOL message: inventing new type variable names: 'b>>
> val it =
    ``:(\'a. 'a -> num, 'b) collection_ops``
     : hol_type
```

Similar to sets, `'a bag` is a type abbreviation for `'a -> num`. In this case, HOL-Omega figures out that the correct type to substitute for `'col` is `\'a. 'a -> num`.

So we can represent lists, sets, and bags as collections using this record type with fields for these three common operations.

4.1.1 Object-oriented collections

In fact we can go further, and try to model collections in an object-oriented way, combining together the data values stored in the collection with the operations used to manipulate them.

```
- Hol_datatype `collection =                                            7
     <| this : 'x 'col;
        ops  : ('col,'x) collection_ops |>`;
```

Now we can define an operation to insert an element into a collection, without having to know what particular kind of collection it is.

```
- val insert_def =                                                        8
  Define ‘insert x (c:(’col,’x)collection) =
             <| this := c.ops.insert x c.this;
                ops  := c.ops |>‘;
Definition has been stored under "insert_def"
> val insert_def =
    |- !x c. insert x c = <|this := c.ops.insert x c.this; ops := c.ops|>
     : thm
```

Similarly, we can define an operation to measure the size of a collection.

```
- val length_def =                                                        9
  Define ‘length (c:(’col,’x)collection) = c.ops.length c.this‘;
Definition has been stored under "length_def"
> val length_def =
    |- !c. length c = c.ops.length c.this
     : thm
```

So we can use the same functions, `insert` and `length`, to manipulate any lists, sets, or bags, with the appropriate results for each type of collection.

4.1.2 Fold operation

But what if we want to add a “fold” operator, like the `FOLDR` function on lists:

```
- type_of ‘‘FOLDR‘‘;                                                      10
<<HOL message: inventing new type variable names: ’a, ’b>>
> val it =
    ‘‘:(’a -> ’b -> ’b) -> ’b -> ’a list -> ’b‘‘
     : hol_type

- listTheory.FOLDR;
> val it =
    |- (!f e. FOLDR f e [] = e) /\
       !f e x l. FOLDR f e (x::l) = f x (FOLDR f e l)
     : thm
```

We might add a new field `fold` to our new record of collection operations as follows.

```
- Hol_datatype ‘collection_ops =                                          11
     <| empty  : ’x ’col;
        insert : ’x -> ’x ’col -> ’x ’col;
        length : ’x ’col -> num;
        fold   : (’x -> ’y -> ’y) -> ’y -> ’x ’col -> ’y |>‘;
<<HOL message: Defined type: "collection_ops">>
> val it = () : unit
```

Then we can construct a record of this type using `FOLDR`.

```
- val list_ops = Define                                                   12
     ‘list_ops = <| empty := []:’a list; insert := CONS; length := LENGTH;
                    fold := FOLDR|>‘;
<<HOL message: inventing new type variable names: ’b>>
Definition has been stored under "list_ops_def"
> val list_ops =
    |- list_ops =
       <|empty := []; insert := CONS; length := LENGTH; fold := FOLDR|> : thm

- type_of ‘‘list_ops‘‘;
<<HOL message: inventing new type variable names: ’a, ’b>>
> val it =
    ‘‘:(list, ’a, ’b) collection_ops‘‘
     : hol_type
```

Wait, this is not what we wanted. There is a third type argument in `collection_ops` now, `’b`. This new type argument appears there because there are now three free type variables in the definition of `collection_ops`, `’col`, `’x`, and `’y`. The third argument `’y` is the type of the value computed and returned by `fold`.

But having the `’y` type variable free in this way *fails to be fully general*, as any particular instance of `fold` can produce only one type of result. No matter its arguments, no different type of result can be produced.

To see this problem more clearly, suppose we follow this development further, using this definition of `collection_ops`, and upon it defining the collection type and the fold operation on collections.

```
- Hol_datatype ‘collection =                                              13
     <| this : ’x ’col;
        ops  : (’col,’x,’y) collection_ops |>‘;
<<HOL message: Defined type: "collection">>
> val it = () : unit

- val fold_def = Define ‘fold f e c = c.ops.fold f e c.this‘;
<<HOL message: inventing new type variable names: ’a, ’b, ’c>>
Definition has been stored under "fold_def"
> val fold_def =
    |- !f e c. fold f e c = c.ops.fold f e c.this
     : thm
```

Now let’s make an example collection.

```
- val ex1 = ‘‘<| this := [1;8;27]; ops := list_ops |>‘‘;                  14
<<HOL message: inventing new type variable names: ’a>>
> val ex1 = ‘‘<|this := [1; 8; 27]; ops := list_ops|>‘‘ : term
```

But when we try to do a fold on this example, we see a type error.

```
- ``fold (\x y. x+y) 0 ^ex1``;                                            15

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

(fold (\(x :num) (y :num). x + y) (0 :num) :
   (list, num, num) collection -> num)

on line 16, characters 2-19

which has type

:(list, num, num) collection -> num

to

<|this := [(1 :num); (8 :num); (27 :num)];
  ops := (list_ops :(list, num, 'a) collection_ops)|>

between beginning of frag 1 and end of frag 1

which has type

:(list, num, 'a) collection

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

This example failed type-checking because the type of the result that the collection was able to provide (`'a`) was not the same as the type of the value that the actual fold function, `\x y.x+y`, was trying to return (`num`).

We could try to patch this up by manually instantiating this example.

```
- val ex1a = inst [``:'a`` |-> ``:num``] ex1;                             16
> val ex1a = ``<|this := [1; 8; 27]; ops := list_ops|>`` : term
- ``fold (\x y. x+y) 0 ^ex1a``;
> val it =
    ``fold (\x y. x + y) 0 <|this := [1; 8; 27]; ops := list_ops|>``
     : term
```

This does work and the term passes type-checking. But what if we try another example that returns a result of a different type?

```
- ``fold (\x y. EVEN x /\ y) T ^ex1a``;                              17

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

(fold (\(x :num) (y :bool). EVEN x /\ y) T :
   (list, num, bool) collection -> bool)

on line 21, characters 2-27

which has type

:(list, num, bool) collection -> bool

to

<|this := [(1 :num); (8 :num); (27 :num)];
  ops := (list_ops :(list, num, num) collection_ops)|>

between beginning of frag 1 and end of frag 1

which has type

:(list, num, num) collection

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

The type of the result that the collection was able to provide (`num`) was not the same as the type of the value that the fold function was trying to return (`bool`).

The point here is that the above version of fold is simply not general enough for normal use. What we really want is the following version.

```
- Hol_datatype `collection_ops =                                      18
     <| empty  : 'x 'col;
        insert : 'x -> 'x 'col -> 'x 'col;
        length : 'x 'col -> num;
        fold   : !'y. ('x -> 'y -> 'y) -> 'y -> 'x 'col -> 'y |>`;
<<HOL message: Defined type: "collection_ops">>
> val it = () : unit
```

In this new defintion of `collection_ops`, the type of the `fold` field begins with "`!'y.`". This indicates a *universal type*; the idea comes from a logic called System F. The `!'y.` universally quantifies `'y` over the body `('x -> 'y -> 'y) -> 'y -> 'x 'col -> 'y`. The quantification binds the occurrences of `'y` within the universal type, so that `'y` does not become a free type variable outside the binding, and thus not a free type variable of the `collection_ops` type. Then this version of the `collection_ops` type is created with just its normal two arguments `'col` and `'x`, not `'y`.

To create an example of this new type of fold operation, we need to provide a term whose type is the above universal type. Such a term is `\:'b. FOLDR`.

```
- val list_ops = Define                                                      19
   `list_ops = <|empty := []:'a list; insert := CONS; length := LENGTH;
                 fold := \:'b. FOLDR|>`;
Definition has been stored under "list_ops_def"
> val list_ops =
    |- list_ops =
       <|empty := []; insert := CONS; length := LENGTH;
         fold := (\:'b. FOLDR)|> : thm

- type_of ``list_ops``;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    ``:(list, 'a) collection_ops``
     : hol_type
```

The term `\:'b. FOLDR` is a *type abstraction term*. It abstracts a term, here `FOLDR`, not by a term variable, but by a type variable `'b`. This is a new variety of term not present in HOL, but added in HOL-Omega. The type of such a term is a universal type. Where the type of `FOLDR` is `('a -> 'b -> 'b) -> 'b -> 'a 'col -> 'b`, the type of `\:'b. FOLDR` is instead `!'b. ('a -> 'b -> 'b) -> 'b -> 'a 'col -> 'b`.

The use of a universal type and a type abstraction term here provides the generality we were looking for, so that `fold` can be used to return results of any desired type.

```
- Hol_datatype `collection =                                                 20
     <| this : 'x 'col;
        ops  : ('col,'x) collection_ops |>`;
<<HOL message: Defined type: "collection">>
> val it = () : unit

- val fold_def =
  Define `fold f (e:'b) (c:('col,'a)collection) = c.ops.fold f e c.this`;
Definition has been stored under "fold_def"
> val fold_def =
    |- !f e c. fold f e c = c.ops.fold f e c.this
     : thm
```

If we turn on the printing of the types of terms, we can see in more detail the types involved in the `fold` operation.

```
- show_types := true;                                                        21
> val it = () : unit
- fold_def;
> val it =
    |- !(f :'a -> 'b -> 'b) (e :'b) (c :('col :ty => ty, 'a) collection).
         fold f e c = c.ops.fold [:'b:] f e c.this
     : thm
```

Now in the definition of `fold`, we see `[:'b:]`. This indicates an application of the term `c.ops.fold` to the type `'b` as a type argument. It is like an application of a term to a term argument, except the argument is a type, not a term. In such a *type application term*, the operator has to have a universal type; in this case, the type of `c.ops.fold` is `!'b. ('a -> 'b -> 'b) -> 'b -> 'a 'col -> 'b`. The result of the type application is to substitute the type argument for the bound type variable throughout the term. In this case, the result has type `('a -> 'b -> 'b) -> 'b -> 'a 'col -> 'b`. It is therefore ready to take as its next arguments the terms `f`, `e`, and `c.this`.

The type arguments to terms are important for the logic, but in practice they tend to make terms harder to read, so by default their printing is turned off. Also, in many cases the user need not mention them when writing terms; the parser's type inference will try to deduce where they are needed, and then exactly which type argument should be inserted there. That is how the `[:'b:]` type argument was inserted into the definition of `fold` above.

This version of the fold operation can be used easily to construct folds returning different types, without any manual instantiations.

```
- show_types := false;                                                    22
> val it = () : unit
- val ex1 = ``<| this := [2;3;5;7]; ops := list_ops |>``;
> val ex1 = ``<|this := [2; 3; 5; 7]; ops := list_ops|>`` : term

- ``fold (\x y. x+y) 0 ^ex1``;
> val it =
    ``fold (\x y. x + y) 0 <|this := [2; 3; 5; 7]; ops := list_ops|>``
     : term

- ``fold (\x y. EVEN x /\ y) T ^ex1``;
> val it =
    ``fold (\x y. EVEN x /\ y) T <|this := [2; 3; 5; 7]; ops := list_ops|>``
     : term
```

4.1.3 Map operation

This seems to be working well. Let's try another extension, adding a "map" operation to the group of operations on collections. The basic idea of a map operation is to apply a function to every element of a collection, and from all of the results form a new collection. For lists, HOL contains the `MAP` function predefined, and there are similar functions for sets and bags.

```
- type_of ``MAP``;                                                        23
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it =
    ``:('a -> 'b) -> 'a list -> 'b list``
     : hol_type
```

```
- listTheory.MAP;                                                          24
> val it =
    |- (!f. MAP f [] = []) /\ !f h t. MAP f (h::t) = f h::MAP f t
     : thm
```

Suppose we try to extend the set of operations with an entry for `map`, using a universally quantified type in the same style as we did for `fold`.

```
- Hol_datatype `collection_ops =                                           25
     <| length : 'x 'col -> num;
        empty  : 'x 'col;
        insert : 'x -> 'x 'col -> 'x 'col;
        fold   : !'y. ('x -> 'y -> 'y) -> 'y -> 'x 'col -> 'y;
        map    : !'y. ('x -> 'y) -> 'x 'col -> 'y 'col |>`;
<<HOL message: Defined type: "collection_ops">>
> val it = () : unit
```

To fashion an example of this `map` operation, we need to provide a term whose type is the universal type `!'y. ('x -> 'y) -> 'x 'col -> 'y 'col`, such as `\:'b. MAP`.

```
- val list_ops = Define                                                    26
   `list_ops = <|empty := []:'a list; insert := CONS; length := LENGTH;
                 fold := \:'b. FOLDR; map := \:'b. MAP |>`;
Definition has been stored under "list_ops_def"
> val list_ops =
    |- list_ops =
       <|empty := []; insert := CONS; length := LENGTH;
         fold := (\:'b. FOLDR); map := (\:'b. MAP)|>
     : thm

- type_of ``list_ops``;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    ``:(list, 'a) collection_ops``
     : hol_type
```

Next we can recreate the type of collections, using this expanded record of operations.

```
- Hol_datatype `collection =                                               27
     <| this : 'x 'col;
        ops  : ('col,'x) collection_ops |>`;
<<HOL message: Defined type: "collection">>
> val it = () : unit
```

Now we define the "map" operation that takes a function and a collection and creates a new collection from the results.

```
- val map_def =                                                            28
  Define `map (f:'a -> 'b) c =
           <| this := c.ops.map f c.this;
              ops  := c.ops |> `;
```

Unfortunately, this definition runs into difficulties with the typing.

```
Exception raised at Preterm.typecheck:                                    29
on line 113, characters 15-30:

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

 _ record fupdatethis
  (K
     ((c :('col :ty => ty, 'a) collection).ops.map [:'b:] (f :'a -> 'b)
        c.this) :'b 'col -> 'b 'col)

between line 112, character 12 and line 113, character 30

which has type

:('col :ty => ty, 'b) collection -> ('col, 'b) collection

to

<|ops := (c :('col :ty => ty, 'a) collection).ops|>

on line 113, characters 15-30

which has type

:('col :ty => ty, 'a) collection

unification failure message: unify failed

! Uncaught exception:
! HOL_ERR
```

The details of the above error message are not important. The real problem here is that the type of the new collection created is `('col,'b)collection`, while the type of the original collection is `('col,'a)collection`. The new collection being formed has its `this` field given a value of the new collection type, but the `ops` field is given a record of operations on the old collection type, not the new one.

This problem can be resolved by using one more universal type for the `ops` field itself.

```
- Hol_datatype `collection =                                              30
     <| this : 'x 'col;
        ops  : !'x. ('col,'x) collection_ops |>`;
<<HOL message: Defined type: "collection">>
> val it = () : unit
```

Now the `map` function can be defined as we desire, with no type problems.

```
- val map_def =                                                          31
  Define ‘map (f:’a -> ’b) c =
            <| this := c.ops.map f c.this;
               ops  := c.ops |> ‘;
Definition has been stored under "map_def"
> val map_def =
    |- !f c. map f c = <|this := c.ops.map f c.this; ops := c.ops|>
     : thm
```

To check on the types involved, let’s turn on the display of types.

```
- show_types := true;                                                    32
> val it = () : unit

- map_def;
> val it =
    |- !(f :’a -> ’b) (c :(’col :ty => ty, ’a) collection).
         map f c =
         <|this := (c.ops [:’a:]).map [:’b:] f c.this; ops := c.ops|>
     : thm
```

Here we can see not only the type argument `[:’b:]` inserted for `map`, as was done before for `fold`, but also the operations record itself `c.ops` is given the type argument `[:’a:]`. The parser’s type inference was able to deduce the necessary type arguments from the actual user input and insert them in the appropriate places.

As a final example in this section, let’s consider an operation that takes two collections, which may use different underlying data structures, and combines their elements into a single collection. We can do this without expanding the definition of `collection_ops`, but just using the operations that are already present.

```
- val union_def =                                                        33
  Define ‘union (c1: (’col1,’a)collection) (c2: (’col2,’a)collection) =
            <| this := fold c2.ops.insert c2.this c1 : ’a ’col2;
               ops  := c2.ops |>‘;
Definition has been stored under "union_def"
> val union_def =
    |- !(c1 :(’col1 :ty => ty, ’a) collection)
          (c2 :(’col2 :ty => ty, ’a) collection).
         union c1 c2 =
         <|this := fold (c2.ops [:’a:]).insert c2.this c1; ops := c2.ops|>
     : thm

- type_of ‘‘union‘‘;
<<HOL message: inventing new type variable names: ’a, ’b, ’c>>
> val it =
    ‘‘:(’a :ty => ty, ’b) collection ->
       (’c :ty => ty, ’b) collection -> (’c, ’b) collection‘‘
     : hol_type
```

So the use of universal types provides the needed type polymorphism, which could not have been accomplished using simply the traditional higher order logic type system.

Much of the advantage of HOL-Omega comes because of the new universal types. The free type variables in classic HOL types could be thought of as being implicitly univerally quantified, as they can be substituted by any other type to form a type instance. But in HOL-Omega, the $\forall$ quantification can be found *within* a type, as in $(\forall\alpha.\alpha \to \alpha) \to$ `bool`. This use of the $\forall$ in the left hand side of a function type $(\to)$ is key to much of the new functionality of HOL-Omega.

4.1.4 Abstract collections

We have seen how one could create a very nice version of collections, modeled in an object-oriented way, so that the operations that obtain the size of a collection, fold over a collection, etc., are invoked the same whether the actual internal data structure is a list, set, or bag. But what that internal data structure is, is still apparent from the type of the collection.

```
- val ex1 = ``<| this := [2;3;5;7]; ops := list_ops |>``;                    34
> val ex1 = ``<|this := [2; 3; 5; 7]; ops := list_ops|>`` : term
- type_of ex1;
> val it =
    ``:(list, num) collection``
     : hol_type

- val ex2 = ``<| this := {2;3;5;7}; ops := set_ops |>``;
> val ex2 = ``<|this := {2; 3; 5; 7}; ops := set_ops|>`` : term
- type_of ex2;
> val it =
    ``:(\'b. 'b -> bool, num) collection``
     : hol_type
```

The internal data structure is visible as `list` in example `ex1` and as `\'b. 'b -> bool` in example `ex2`.

That internal data structure can be represented by a HOL-Omega type operator variable, and that is how a general routine could be written to handle arguments built using any collection structure, as was done above.

But suppose one wanted to completely hide the actual data structure used, abstracting that information away from the external use of the collection, considering it a detail of the implementation. This could be very useful in modularizing a proof, where certain parts of the proof know about the particular implementation data structure, but above a certain layer that information is hidden, and the rest of the proof cannot know or rely on that choice, but instead must work the same irrespective of what data structure is used. This makes it possible, at a later time, to change the implementation data structure to another structure, perhaps better suited to the task at hand, and to have that change not affect any of the proof work done above the layer where that choice was abstracted, like the edge of a module where internal implementation details cannot leak across the module boundary. This kind of information hiding is very helpful in creating large software systems that are still maintainable and modifiable, and the same ideas apply for large proofs as well.

To accomplish this information hiding, HOL-Omega provides a new variety of type called an *existential type*.

```
- ``:?'col. ('col, 'a) collection``;                                  35
> val it =
    ``:?'col :ty => ty. ('col, 'a) collection``
     : hol_type
- type_vars it;
> val it = [``:'a``] : hol_type list
```

Existential types are written in the type language, similar to universal types, but using an existential type operator. In the example above, the existential notation binds the type variable `'col` across the body of the type, `('col,'a)collection`, so that the free type variables of the type contain just the type variable `'a`, not `'col`.

Terms of existential type are called *packages*. They can be constructed as a special form using the `pack` keyword, as follows.

```
- val list_pack = ``pack (:list, <|this := [2;3;2]; ops := list_ops|>)``;   36
> val list_pack =
    ``pack (:list,<|this := [2; 3; 2]; ops := list_ops|>)``
     : term
- type_of list_pack;
> val it =
    ``:?'x :ty => ty. ('x, num) collection``
     : hol_type
```

The keyword `pack` is followed by a pair where the first element is a type, preceeded by a colon, and the second element is a term. The term, which normally involves the type mentioned, is packaged up so that the type mentioned is hidden, being replaced by a type variable, which becomes the bound type variable of the existential type of the resulting package.

In the case above, the fact that `list_pack` actually contains a list has been removed from the package's type, where `list` has been replaced by the type variable `'x`.

There is the possibility of ambiguity in the types when creating such a package. Given a pair of a type and a term as above, there many be multiple ways that a resulting existential type may be formed. In such cases, the ambiguity can be resolved by using a type annotation on the package. For example, in the session below two different packages are created from exactly the same ingredients, except that one of them has a type annotation. Note that the resulting packages have different existential types; they are therefore different packages.

```
- val list_pack2 =                                                        37
    ``pack (:list, <| this := [[2];[3;5];[7]]; ops := list_ops |> )``;
> val list_pack2 =
    ``pack (:list,<|this := [[2]; [3; 5]; [7]]; ops := list_ops|>)``
     : term
- type_of list_pack2;
> val it =
    ``:?'x :ty => ty. ('x, num 'x) collection``
     : hol_type

- val list_pack3 =
    ``pack (:list, <| this := [[2];[3;5];[7]]; ops := list_ops |> )
        : ?'x. ('x,num list) collection``;
> val list_pack3 =
    ``pack (:list,<|this := [[2]; [3; 5]; [7]]; ops := list_ops|>)``
     : term
- type_of list_pack3;
> val it =
    ``:?'x :ty => ty. ('x, num list) collection``
     : hol_type
```

We can construct packages of any kind of collection, and if the collections contain elements of the same type, then the packages themselves have the same type.

```
- val set_pack =                                                          38
    ``pack (:\'a.'a set, <| this := {2;3;2}; ops := set_ops |> )``;
> val set_pack =
    ``pack (:\'a. 'a -> bool,<|this := {2; 3; 2}; ops := set_ops|>)``
     : term
- type_of set_pack;
> val it =
    ``:?'x :ty => ty. ('x, num) collection``
     : hol_type

- val bag_pack =
    ``pack (:\'a.'a bag, <| this := {|2;3;2|}; ops := bag_ops |> )``;
> val bag_pack =
    ``pack (:\'a. 'a -> num,<|this := {|2; 3; 2|}; ops := bag_ops|>)``
     : term
- type_of bag_pack;
> val it =
    ``:?'x :ty => ty. ('x, num) collection``
     : hol_type
```

Since all these packages have the same type, it is easy to write routines to take them as arguments. The new feature needed is an extension of the `let ... in` form to deconstruct a package into a pair of a type variable and a term, where the type variable represents the actual type that was hidden, and where the term represents the body of the package, but where the hidden type is again represented by the type variable.

```
- val lengthp_def =                                                    39
  Define ‘lengthp (p: ?’col. (’col,’a)collection) =
            let (:’col, c) = p in
              c.ops.length c.this ‘;
Definition has been stored under "lengthp_def"
> val lengthp_def =
    |- !p. lengthp p = (let (:’col :ty => ty,c) = p in c.ops.length c.this)
     : thm
```

In the `let` form above, the package `p` (of type `?’col.(’col,’a)collection`) is unpacked into the pair of the type variable `’col` and the term variable `c`, where `c` has the type `(’col,’a)collection`. The scopes of both `’col` and `c` include the body of the `let...in` form. But the scope of `’col` also includes the term variable `c`, so that the `’col` that appears in the type of `c`, `(’col,’a)collection`, is that `’col` that was just bound. Both `’col` and `c` have no meaning outside the `let`, so in particular it is meaningless to have the body of the `let` return a value of a type involving `’col`. Such an escape of `’col` from its scope is prevented by the strong typing of the HOL-Omega logic.

Suppose we try to violate this rule, by defining an operation that returns the internal data structure of a collection. Such a definition produces the following error message:

```
- val this_def =                                                       40
  Define ‘this (p: ?’col. (’col,’a)collection) =
            let (:’col, c) = p in
              c.this ‘;
Exception raised at Preterm.typecheck:
roughly on line 85, characters 14-19:

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

(UNPACK :(!(’x :ty => ty). (’x, ’a) collection -> ’a (’col :ty => ty))
         -> (?(’x :ty => ty). (’x, ’a) collection) -> ’a ’col)

roughly on line 84, characters 16-29

to

\:’col :ty => ty. (\(c :(’col, ’a) collection). c.this)

roughly on line 85, characters 14-19

which has type

:!’col :ty => ty. (’col, ’a) collection -> ’a ’col

unification failure message: unify failed

! Uncaught exception:
! HOL_ERR
```

But as long as we don't violate the rules, we are fine, and can define operations to return packages newly constructed out of parts of other packages. Here is an example of an operation that takes a package as an argument, inserts an element, and then returns the result as a new package.

```
- val insertp_def =                                                          41
  Define `insertp (e:'a) (p: ?'col. ('col,'a)collection) =
            let (:'col, c) = p in
              pack (:'col,
                    <| this := c.ops.insert e c.this : 'a 'col;
                       ops  := c.ops |>) `;
Definition has been stored under "insertp_def"
> val insertp_def =
    |- !e p.
         insertp e p =
         (let (:'col :ty => ty,c) = p
          in
            pack
              (:'col :ty => ty,
                <|this := c.ops.insert e c.this; ops := c.ops|>))
     : thm
```

We can define operations to do folds and maps on collections using the operators `fold` and `map` defined before, but where the new operations now work on packages, where the data structure is hidden internally.

```
- val foldp_def =                                                            42
  Define `foldp (f:'a -> 'b -> 'b) (e:'b) (p: ?'col. ('col,'a)collection) =
            let (:'col, c) = p in
              fold f e c `;
Definition has been stored under "foldp_def"
> val foldp_def =
    |- !f e p. foldp f e p = (let (:'col :ty => ty,c) = p in fold f e c)
     : thm

- val mapp_def =
  Define `mapp (f:'a -> 'b) (p: ?'col. ('col,'a)collection) =
            let (:'col, c) = p in
              pack (:'col, map f c) `;
Definition has been stored under "mapp_def"
> val mapp_def =
    |- !f p.
         mapp f p =
         (let (:'col :ty => ty,c) = p in pack (:'col :ty => ty,map f c))
     : thm
```

In fact, we can actually build a single operation that takes any two collection pacakges and combines their elements, even if they happen to have different underlying data structures, like lists and bags. The result here is calculated to have the same underlying data structure as the second argument.

```
- val unionp_def =                                                  43
  Define ‘unionp (p1: ?’col. (’col,’a)collection)
                 (p2: ?’col. (’col,’a)collection) =
            let (:’col1, c1) = p1 in
            let (:’col2, c2) = p2 in
                pack (:’col2, union c1 c2) ‘;
Definition has been stored under "unionp_def"
> val unionp_def =
    |- !p1 p2.
         unionp p1 p2 =
         (let (:’col1 :ty => ty,c1) = p1 in
          let (:’col2 :ty => ty,c2) = p2
          in
            pack (:’col2 :ty => ty,union c1 c2))
     : thm
```

Using packages in this way makes it easier to modularize a large proof, by providing a way to hide the information about which particular types are being used at a lower level in the overall proof. This information hiding has major advantages for proof maintenance and modification over time.

Chapter 5

The HOL-Omega Logic

An earlier chapter covered the classic HOL logic. This chapter will discuss the HOL-Omega logic, and focus on its extensions and new features not present in classic HOL. In essence, these center on two main ideas, and what flows as a consequence from each:

- Types can be abstracted by type variables (similar to how terms are abstracted by term variables in the lambda calculus).
  - Type operators are curried, so that they may take one argument at a time.
  - Every type has a *kind*; kinds determine which type applications are sensible.
  - Type variables can represent type operators.
- Terms can be abstracted by type variables (similar to System *F*).
  - The type of such an abstraction is a *universal* type.
  - Such an abstraction may be applied as a function to a type argument.
  - Such applications are managed by classifying all types into *ranks*.

In this chapter, we will give the new notation used to write expressions of the HOL-Omega logic, how to construct these expressions by ML functions, and also discuss new HOL-Omega proof techniques. Only the most essential new elements are given here, being a tutorial. The full logic is described in detail in *DESCRIPTION*.

5.1 New notation

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

New terms of the HOL-Omega Logic

Variety of term	*HOL-Omega notation*	*Standard notation*	*Description*
$\forall$-type quantification	`!:`α`.`t	$\forall\alpha.\ t$	*for all* $\alpha : t$
$\exists$-type quantification	`?:`α`.`t	$\exists\alpha.\ t$	*for some* $\alpha : t$
λ-type abstraction	`\:`α`.`t	$\lambda\alpha.\ t$	*given* α*, yield* t
Type application	t `[:`σ`:]`	$t[\sigma]$	*apply* t *to type* σ

The forms `!:`α`.`t and `?:`α`.`t are straightforward analogs of the universal and existential quantifiers for terms `!`x`.`t and `?`x`.`t, except that instead of a term variable x being quantified over all values of the type of x, a type variable α is being quantified over all types of the *kind* of α. Kinds will be described later in this chapter. For both quantifiers, the body t must have boolean type, and the meaning of the quantification is that the body is always or sometimes true as α ranges over its domain. Similarly, `\:`α`.`t is an analog of term abstractions `\`x`.`t. Here the meaning of the abstraction is a function from the domain of α to the meaning of t, with free occurrences of α in t substituted by the function's argument. In each of these three forms, the type variable α may occur free in the body t, and such occurrences are considered bound by the type quantification or type abstraction. The form t `[:`σ`:]` is an analog of the normal application of a term function to a term argument, except that here the argument is a type, not a term.

The sequences `!:`, `?:`, `\:`, `[:`, and `:]` are each considered one symbol. The presence of the colon (`:`) in these is meant as a reminder that a type is involved, rather than a term.

Each of these forms can also handle multiple types, not just one:

New terms of the HOL-Omega Logic

Variety of term	*HOL$_\omega$ notation*	*Stand. notation*	*Description*
$\forall$-type quantification	`!:`$\alpha_1 \ldots \alpha_n$`.`t	$\forall\alpha_1 \ldots \alpha_n.\ t$	*for all* $\alpha_1, \ldots, \alpha_n : t$
$\exists$-type quantification	`?:`$\alpha_1 \ldots \alpha_n$`.`t	$\exists\alpha_1 \ldots \alpha_n.\ t$	*for some* $\alpha_1, \ldots, \alpha_n : t$
λ-type abstraction	`\:`$\alpha_1 \ldots \alpha_n$`.`t	$\lambda\alpha_1 \ldots \alpha_n.\ t$	*given* $\alpha_1, \ldots, \alpha_n$*, yield* t
Type application	t `[:`$\sigma_1, \ldots, \sigma_n$`:]`	$t[\sigma_1, \ldots, \sigma_n]$	*apply* t *to types* $\sigma_1, \ldots, \sigma_n$

As for HOL, terms of the HOL-Omega logic are represented in ML by the *abstract type* `term`. They are normally input between double back-quote marks. For example, the expression ``` ``!:'a. P [:'a:] ==> Q [:'a:]`` ``` evaluates in ML to a term representing $\forall$`'a. P[`'a`]` $\Rightarrow$ `Q[`'a`]`. The new terms can be manipulated by various built-in ML functions, similar to the ones seen previously. For example, the ML function `dest_tyforall` with ML type `term -> hol_type * term` splits a universal type quantification into a pair of a type and a term, where the type is the bound type variable and the term is the body of the quantification. Similarly, the ML function `dest_tycomb` of type `term -> term * hol_type` splits a type application into its term operator and its type operand. [1]

When ML values of type `term` are submitted to the ML interpreter, they are displayed back to the user, along with their ML type. The text that is displayed for a term is affected by several global settings. Among these are whether types within the term should be displayed, such as the types of variables, and whether the Unicode character set should be

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

used to display some term operators and type variables using special symbols and Greek letters. For clarity, in these examples of interactions with HOL-Omega, we will assume that the use of Unicode characters has been turned off by `set_trace "Unicode" 0`, as in the first session below. But in normal use, most users will probably wish to keep the default setting, in order to enjoy the more attractive display.

By default, most types are not displayed when terms are printed. In addition to the types of variables, this also includes the type arguments in type applications, such as `[:'a:]` in the term `P [:'a:]`, which appears as just `P`.

```
> set_trace "Unicode" 0;                                                    1
val it = () : unit
> ``!:'a. P [:'a:]``;
val it = ``!:'a. P`` : term
> dest_tyforall it;
val it = (``:'a``, ``P``) : hol_type * term
> dest_tycomb(#2 it);
val it = (``P``, ``:'a``) : term * hol_type
```

Alternatively, we can set `show_types` to `true` in order to see all type applications.

```
> show_types := true;                                                       2
val it = () : unit
> ``!:'a. P [:'a:]``;
val it =
   ``!:'a. (P :!'a. bool) [:'a:]``
   : term
> dest_tyforall it;
val it =
   (``:'a``,
    ``(P :!'a. bool) [:'a:]``)
   : hol_type * term
> dest_tycomb(#2 it);
val it =
   (``(P :!'a. bool)``,
    ``:'a``) : term * hol_type
```

We collectively call such type quantifications and type abstractions, whether single or multiple, *type binders*.

Note: There is an important restriction on forming such type binders. Given a form which binds a type variable α, such as `!:`$\alpha.t$, the body t cannot contain any free term variables x whose type contains (freely) the type variable α. If it did, then the variable x would also be a free variable of `!:`$\alpha.t$ (since a term variable x is never bound by a type binder like `!:`α). But then since x is visible outside the type binder, so is x's type, α, which is the very type variable bound by the type binder. But α has no meaning outside its type binder, so this makes no sense. Here is an example of violating this restriction.

```
> ``!:'a. (x:'a) = x``;                                                   3

Type variable scoping failure: the abstraction by the type variable

:'a

on line 2, characters 4-5

of the term

(x :'a) = x

roughly on line 2, characters 8-17

contains the free term variable

(x :'a)

at line 2, character 9

whose type contains freely the type variable being abstracted,

:'a

on line 2, characters 4-5
```

This restriction is only sensible, since if one could look at a free variable such as `x:'a` from outside the type binder, then one could also see the type variable `'a` (since it is a part of `x`), but `'a` is supposed to be hidden within the type binder `!:'a. (x:'a) = x`.

Syntax of HOL-Omega types The types of the HOL-Omega logic form an ML type called `hol_type`. Every term in the logic has a well-defined type, but unlike HOL, not every type has some term of which it is the type. In other words, there are some types which are not the type of any term. Expressions having the form ``` ``: ··· `` ``` evaluate to logical types. The built-in function `type_of` has ML type `term->hol_type` and returns the logical type of a term.

To try to minimise confusion between the logical types of HOL-Omega terms and the ML types of ML expressions, the former is referred to as *object language types* and the latter as *meta-language types*. For example, ``` ``!:'a.T`` ``` is an ML expression that has meta-language type `term` and evaluates to a term with object language type ``` ``:bool`` ```.

```
> ``!:'a.T``;                                                             4
val it = ``!:'a. T`` : term

> type_of it;
val it = ``:bool`` : hol_type
```

Term constructors HOL-Omega terms can be input, as above, by using explicit *quotation*, or they can be constructed by calling ML constructor functions.

Each of the new forms can be constructed, broken apart, or tested by ML functions:

ML constructors, destructors, and tests

HOL$_\omega$ notation	*Constructor*	*Destructor*	*Test*
`!:`$\alpha.t$	`mk_tyforall`	`dest_tyforall`	`is_tyforall`
`?:`$\alpha.t$	`mk_tyexists`	`dest_tyexists`	`is_tyexists`
`\:`$\alpha.t$	`mk_tyabs`	`dest_tyabs`	`is_tyabs`
t `[:`σ`:]`	`mk_tycomb`	`dest_tycomb`	`is_tycomb`

There are similar ML functions for the forms with multiple types:

ML constructors and destructors for multiple types

HOL$_\omega$ notation	*Constructor*	*Destructor*
`!:`$\alpha_1 \ldots \alpha_n.t$	`list_mk_tyforall`	`strip_tyforall`
`?:`$\alpha_1 \ldots \alpha_n.t$	`list_mk_tyexists`	`strip_tyexists`
`\:`$\alpha_1 \ldots \alpha_n.t$	`list_mk_tyabs`	`strip_tyabs`
t `[:`$\sigma_1, \ldots, \sigma_n$`:]`	`list_mk_tycomb`	`strip_tycomb`

Here are some examples of their use:

5

```
> val x = mk_tyforall(``:'a``, ``T``);
val x = ``!:'a. T`` : term

> val x2 = mk_tyforall(``:'b``, x);
val x2 = ``!:'b 'a. T`` : term

> is_tyforall x2;
val it = true : bool

> dest_tyforall x2;
val it = (``:'b``, ``!:'a. T``) : hol_type * term
> strip_tyforall x2;
val it = ([``:'b``, ``:'a``], ``T``)
   : hol_type list * term

> list_mk_tyforall([``:'c``,``:'b``,``:'a``], ``T``);
val it = ``!:'c 'b 'a. T`` : term
> strip_tyforall it;
val it = ([``:'c``, ``:'b``, ``:'a``], ``T``)
   : hol_type list * term
```

Varieties of terms The four different varieties of terms of HOL are still present in HOL-Omega: variables, constants, function applications (``$t_1$ $t_2$``), and lambda abstractions (``\\$x$.$t$``). In addition, HOL-Omega adds two new fundamental varieties: type applications (``$t$ [:$\sigma$:]``) and type abstractions (``\\:$\alpha$.$t$``). More complex terms, including the type quantifications ``!:$\alpha$.$t$`` and ``?:$\alpha$.$t$``, are just compositions of terms from this simple set.

It is foundational that every term has a type. The question then follows, what are the types of these two new varieties of terms? In particular, what is the type of a type abstraction term ``\\:$\alpha$.$t$``? This turns out to be a potent and valuable new variety of types, not present in HOL but introduced in HOL-Omega, called *universal types*.

Universal types A universal type is written as !α.σ, where α is a type variable and σ is a type expression. With Unicode it appears as $\forall\alpha.\sigma$. The meaning of such a universal type is an infinite collection of all the possible types that σ can represent, for all the possible values of α, where the collection is indexed by the values of α.

Do not confuse this universal type with the universal term expression !:α.t.

The type variable α usually will appear free in σ, but it does not have to. α is considered bound by the ! quantifier over the body σ, so α will never be a free type variable of !α.σ. The ML function type_vars returns a list of the free type variables in a type. Universal types may be constructed using the parser, or they may be constructed and taken apart by ML functions mk_univ_type and dest_univ_type.

```
> val ty1 = mk_univ_type(``:'a``, ``:'a -> 'a``);                    6
val ty1 = ``:!'a. 'a -> 'a``
   : hol_type

> type_vars ty1;
val it = [] : hol_type list

> dest_univ_type ty1;
val it = (``:'a``, ``:'a -> 'a``)
   : hol_type * hol_type

> val ty2 = ``:!'a. 'a -> 'b``;
val ty2 = ``:!'a. 'a -> 'b``
   : hol_type

> type_vars ty2;
val it = [``:'b``] : hol_type list
```

Types of terms We can now state the types of the two new varieties of terms, type applications and type abstractions.

Type abstraction terms have types which are universal types. In particular, if the term t has type σ, then the term \\:α.t has type !α.σ.

Type application terms have types determined in the following way. For a type application term t`[:`τ`:]` to be well-typed, it is required that the term t have a universal type, say `!`α`.`σ. Then the type of t`[:`τ`:]` is $\sigma[\tau/\alpha]$, where the type expression τ is substituted for all free occurences of α in σ.

```
> type_of ``\:'a. 3:num``;                                          7
val it = ``:!'a. num`` : hol_type

> type_of ``(f : !'a. 'a -> 'a) [: num :]``;
val it = ``:num -> num`` : hol_type

> ``(f : 'a -> 'a) [: num :]``;

Type inference failure: unable to form the application of the term

(f :'a -> 'a)

at line 17, character 3

to the type

:num :ty

on line 17, characters 20-22

since the term does not have a universal type.

unification failure message: unify failed
```

Ranks of types To ensure the soundness of the HOL-Omega logic, in addition to being well-typed, a term must also be *well-ranked*. In HOL-Omega, every type has a rank. The rank of a type is a natural number $0, 1, 2, ...$; it can never be negative. This is essentially a measure of how deeply universal (or existential) types are nested within the type. All the types in the classic HOL logic have no universal types and are of rank 0. Forming a universal type out of a bound type variable of rank 0 and a body of rank 0 or 1 will yield a type of rank 1. Higher rank types can be constructed as well. All the types that can be constructed in HOL-Omega can be classified into one of these ranks. The rank of a type may be obtained by the ML function `rank_of_type` of type `hol_type -> int`.

The purpose of ranks is to restrict what types can properly be the argument in a type application term. If one could apply a type abstraction term to its *own* type as an argument, this would introduce a circularity that could imperil the soundness of the logic. To prevent this, for every type application term t`[:`τ`:]`, where t has type `!`α`.`σ, it is required that the rank of τ be less than or equal to the rank of α. If it is greater than the rank of α, this will be caught and reported as an error.

By default, all type variables are given rank 0. The user can specify a particular rank for a type variable by annotating it with a *rank constraint on types,* as in `('a :<= 1)`.

The rank restriction on type applications is checked when terms are constructed by the ML function `mk_tycomb`.

```
> mk_tycomb(``f : !'a:<=0. 'a -> 'a``, ``:!'a:<=0. 'a -> 'a``)         8
# handle e => Raise e;

Exception raised at Term.mk_tycomb:
type application argument has rank exceeding that expected
Exception-
   HOL_ERR
  {message = "type application argument has rank exceeding that expected",
  origin_function = "mk_tycomb", origin_structure = "Term"} raised
```

The rank restriction on type applications is also checked when terms are parsed.

```
> ``(\:'a:<=0. \x:'a. x) [: !'a:<=0. 'a -> 'a :]``;                   9

Rank inference failure: unable to infer a rank for the application of the term

\:'a. (\(x :'a). x)

roughly on line 4, characters 20-21

which expects a type of rank 0

to the type

:!'a. 'a -> 'a

roughly on line 4, characters 38-42

which has rank 1

rank unification failure message: unify failed
Exception-
   HOL_ERR
  {message = "roughly on line 4, characters 38-42:\nfailed", origin_function =
  "typecheck", origin_structure = "Preterm"} raised
```

Kinds of types Just as every term has a type, in HOL-Omega every type has a *kind*. Kinds can be thought of as collections of type values, just as types can be thought of as collections of term values. Also, just as types are used to determine when an argument to a term function is properly well-typed, kinds are used to determine when a type argument to a type function is properly *well-kinded*. As in HOL-Omega terms and types are represented in ML as members of the ML types `term` and `hol_type`, so kinds are represented as members of the ML type `kind`.

The most common and basic kind is `ty`. This is the kind of every type in the logic of classic HOL. The ML function `kind_of : hol_type -> kind` shows the kind of a type:

```
> kind_of ``:bool``;                                          10
val it = ``::ty`` : kind

> kind_of ``:num``;
val it = ``::ty`` : kind

> kind_of ``:num -> num``;
val it = ``::ty`` : kind

> kind_of ``:num list -> (num list # bool)``;
val it = ``::ty`` : kind

> load "realLib";
val it = () : unit
> kind_of ``:real``;
val it = ``::ty`` : kind
```

The kinds above are printed like types, except that the contents of the quotation start with a double colon (`::`), instead of a single colon. In this fashion, the parser can be used to create kinds, just as it can create types and terms in the HOL-Omega logic.

There are multiple instances of the `ty` kind, one for each rank. Just as each instance of the term `NIL` has its type as an attribute, so each instance of `ty` has its rank as an attribute. The default rank is 0, or a kind's rank can be specified using a *rank constraint*, like `ty:3`. The rank of a kind is obtained by the ML function `rank_of : kind -> int`.

```
> ``::ty``;                                                   11
val it = ``::ty`` : kind
> rank_of it;
val it = 0 : rank

> ``::ty : 3``;
val it = ``::ty:3`` : kind
> rank_of it;
val it = 3 : rank
```

In the previous example of type application terms, that generated a rank error during parsing, if we omit the rank constraints, then the rank inference will attempt to satisfy the rank restriction on type applicatons by inferring that the type abstraction's bound type variable must have rank 1 to accomodate the rank 1 type argument, and succeed.

```
> show_types := true;                                         12
val it = () : unit
> ``(\:'a. \x:'a. x) [: !'a. 'a -> 'a :]``;
val it =
   ``(\:'a :(ty:1). (\(x :'a). x)) [:!'a. 'a -> 'a:]``
   : term
```

Here the printer shows the type abstraction term's bound type variable `'a` with an annotation, `'a :(ty:1)`. This is a *kind constraint*, saying that the kind of `'a` is `ty:1`, where the kind itself is annotated to have rank 1 by a rank constraint on kinds, `ty:1`. This means that the type variable `'a` also has rank 1.

So types may have constraints which are kinds, and kinds may have constraints which are ranks. In addition, we have seen how types may have contraints which are ranks.

Arrow kinds So far all the types we have seen have had the kind `ty`. The question is, are their any types with different kinds? There are, as we shall see with the `list` type.

```
> kind_of ``:num list``;                                              13
val it = ``::ty`` : kind

> kind_of ``:'a list``;
val it = ``::ty`` : kind

> kind_of ``:list``;
val it = ``::ty => ty`` : kind
```

This may look unexpected and strange to an experienced HOL user, since in HOL one can never have the `list` type name sitting alone like this. The `list` type is a type operator, and it expects exactly one argument. In HOL this argument must always be supplied immediately to create a type, but in HOL-Omega we consider `list` to be a type in its own right. It is distinguished from the classic HOL types that are not type operators by its kind, which is here reported to be `ty => ty`.

The kind `ty => ty` is an example of an *arrow kind*, written as a binary infix between two kinds which are the arguments to the arrow. These arguments may be any kinds, so they may themselves be arrow kinds. The arrow operator => is right associative, so the default parenthesization is to the right.

```
> ``::(ty => ty) => ty``;                                             14
val it = ``::(ty => ty) => ty`` : kind
> ``::(ty => ty) => (ty => ty)``;
val it = ``::(ty => ty) => ty => ty`` : kind
```

Now we can explore what are the kinds of some other type operators of HOL:

```
> kind_of ``:option``;                                                15
val it = ``::ty => ty`` : kind
> kind_of ``:fun``;
val it = ``::ty => ty => ty`` : kind
> kind_of ``:prod``;
val it = ``::ty => ty => ty`` : kind
> kind_of ``:sum``;
val it = ``::ty => ty => ty`` : kind
```

Given an arrow kind k_1 => k_2, k_1 is called the domain and k_2 is called the range.

Kinds are used to manage which type operators can be properly applied to which type arguments, just as types are used to manage which term functions can be properly applied to which term arguments. To be *well-kinded*, a type operator can only be applied to a type argument whose kind matches the type operator's domain. If this is violated, an exception is raised.

16

```
> kind_of ``:list``;
val it = ``::ty => ty`` : kind
> kind_of ``:num``;
val it = ``::ty`` : kind
> kind_of ``:num list``;
val it = ``::ty`` : kind
> kind_of ``:option list`` handle e => Raise e;

Kind inference failure: unable to infer a kind for the application of

:list : ty => ty

on line 38, characters 18-21

to

:option : ty => ty

on line 38, characters 11-16

kind unification failure message: unify failed

Exception raised at Pretype.kindcheck:
on line 38, characters 18-21:
failed
Exception-
   HOL_ERR
  {message = "on line 38, characters 18-21:\nfailed", origin_function =
  "kindcheck", origin_structure = "Pretype"} raised
```

Type operators with multiple arguments Type operators may have any number of arguments, zero or more, but for a particular type operator, the number of arguments it may take is fixed. In HOL this number is called the *arity* of the type operator, but in the HOL-Omega logic this information is represented within the type operator's kind.

17

```
> kind_of ``:fun``;
val it = ``::ty => ty => ty`` : kind
> kind_of ``:prod``;
val it = ``::ty => ty => ty`` : kind
> kind_of ``:sum``;
val it = ``::ty => ty => ty`` : kind
```

If desired, the HOL arity of a type can be obtained by applying `arity_of` to the type's kind. However, not all kinds fit this pattern, and cannot be considered as a simple arity.

```
> kind_of ``:fun``;                                                    18
val it = ``::ty => ty => ty`` : kind
> arity_of it;
val it = 2 : int

> ``::(ty => ty) => ty``;
val it = ``::(ty => ty) => ty`` : kind
> arity_of it handle e => Raise e;

Exception raised at Kind.arity_of:
not an arity kind
Exception-
   HOL_ERR
  {message = "not an arity kind", origin_function = "arity_of",
  origin_structure = "Kind"} raised
```

So the type operators `fun`, `prod`, and `sum`, which each take two type arguments, all have kind `ty => ty => ty`. The notation for this kind is meant to imply that the type operators `fun`, `prod`, and `sum` are *curried*, that is, that they may take their arguments one at a time. This is an extension beyond what is possible in HOL, where all type arguments must be included immediately. But in HOL-Omega, it is perfectly reasonable to defer such applications. The result of a partial application of arguments to a type operator is generally another type operator that can be further applied to more arguments, until the full number of arguments expected have been supplied.

```
> ``:num fun``;                                                        19
val it = ``:num fun`` : hol_type
> kind_of ``:num fun``;
val it = ``::ty => ty`` : kind
> ``:bool (num fun)``;
val it = ``:num -> bool`` : hol_type
> kind_of ``:bool (num fun)``;
val it = ``::ty`` : kind
```

Notice in ``` ``:bool (num fun)`` ``` that it was necessary to use parentheses. For backwards compatibility with HOL, by default the application of type operators to arguments associates to the left. If the parentheses had been omitted as in ``` ``:bool num fun`` ```, the type parser would have tried to first apply `num` as an operator to `bool`, and then to apply `fun` to the result. Since `num` is not an operator, this would have generated an error.

Maintaining the backwards compatibility with HOL guarantees that classic type expressions such as the following will parse correctly.

```
> ``:(num # num) list option``;                                        20
val it =
   ``:(num # num) list option``
   : hol_type
```

One can also use the traditional “tuple” notation for types to apply several arguments to a type operator. The arguments are listed left to right in the tuple before the type operator, so that the first argument to the type operator will be the left-most type in the tuple, with the rest of the arguments succeeding it to the right, as is normal in HOL.

```
> ``:(num,bool)fun``;                                    21
val it = ``:num -> bool`` : hol_type
```

Type applications The application of a type operator to a type argument is actually a new variety of type in HOL, called a *type application*. These type applications may be constructed using the parser as above, or constructed and destructed by the ML functions mk_app_type and dest_app_type. Remember that when parsed or printed, type operator application is *postfix*, not prefix as in the term language.

```
> mk_app_type(``:list``, ``:num``);                      22
val it = ``:num list`` : hol_type
> mk_app_type(``:option``, it);
val it = ``:num list option`` : hol_type
> dest_app_type it;
val it = (``:option``, ``:num list``)
   : hol_type * hol_type
```

All types in the classic HOL logic that have more than one argument are represented in HOL-Omega as a series of type applications. Such multiple type applications may be constructed or taken apart in one step by list_mk_app_type and strip_app_type.

```
> list_mk_app_type(``:fun``, [``:'a``,``:bool``]);       23
val it = ``:'a -> bool`` : hol_type
> strip_app_type it;
val it = (``:fun``, [``:'a``, ``:bool``])
   : hol_type * hol_type list
```

Type constants Simple type names like ``:bool``, ``:list``, and ``:fun`` are called *type constants*. Each type constant contains its name and its kind. HOL-Omega includes all of the type names of HOL as type constants. But the kind of a type constant in HOL-Omega may be any kind of HOL-Omega; it need not be an arity kind as for those in HOL. Instances of these type constants may be constructed or deconstructed by the ML functions mk_con_type, mk_thy_con_type, dest_con_type, and dest_thy_con_type. For example, the function mk_con_type has ML type string * kind -> hol_type.

Note: The classic HOL functions mk_type, mk_thy_type, dest_type, and dest_thy_type are supported in HOL-Omega for backwards compatibility. They will work just as before if given inputs in the classical HOL subset of HOL-Omega. However, these functions are deprecated, and should not be relied on to work as expected on the new types introduced in HOL-Omega. Their role is now replaced by the ML functions to construct type constants and type applications.

```
> val ty1 = mk_type("fun",[``:'a``,``:bool``]);                        24
val ty1 = ``:'a -> bool`` : hol_type
> dest_type ty1;
val it = ("fun", [``:'a``, ``:bool``])
   : string * hol_type list

> strip_app_type ty1;
val it = (``:fun``, [``:'a``, ``:bool``])
   : hol_type * hol_type list
> dest_con_type (fst it);
val it = ("fun", ``::ty => ty => ty``) : string * kind
```

Type variables Type variables in HOL-Omega include all the type variables in HOL. In addition, HOL-Omega supports type variables with any kind of HOL-Omega, whereas the HOL type variables can only have kind `ty`. A type variable with kind `ty => ty` holds values which are type operators like `list` or `option`. Thus, such a type variable with an arrow kind is called a *type operator variable*. As before, the names of type variables must begin with an apostrophe (`'`). Type variables may be constructed and deconstructed by the ML functions `mk_var_type` of type `string * kind -> hol_type` and `dest_var_type` of type `hol_type -> string * kind`.

For backwards compatibility the HOL functions `mk_vartype : string -> hol_type` and `dest_vartype : hol_type -> string` are supported, but they are deprecated.

Please take care to not confuse the similar names, `mk_var_type` and `mk_vartype`.

```
> val ty1 = mk_var_type("'a", ``::ty``);                               25
val ty1 = ``:'a`` : hol_type
> dest_var_type ty1;
val it = ("'a", ``::ty``) : Type.tyvar
> dest_vartype ty1;
val it = "'a" : string

> val ty2 = mk_var_type("'b", ``::(ty => ty) => ty``);
val ty2 = ``:'b :(ty => ty) => ty`` : hol_type
> mk_app_type(ty2, ``:list``);
val it = ``:list ('b :(ty => ty) => ty)`` : hol_type
> dest_var_type ty2;
val it = ("'b", ``::(ty => ty) => ty``) : Type.tyvar
> dest_vartype ty2 handle e => Raise e;

Exception raised at Type.dest_vartype:
type operator kind - use dest_var_type
Exception-
   HOL_ERR
  {message = "type operator kind - use dest_var_type", origin_function =
  "dest_vartype", origin_structure = "Type"} raised
```

Type abstractions Just as in HOL's term language we can form abstractions, `\x.x+1`, so in HOL-Omega's type language we can form type abstractions, such as `\'a.'a->bool`. This can also be written as `λ'a.'a->bool`. These represent functions from types to types, like type operators. Similarly to universal types, type abstractions $\backslash\alpha.\sigma$ bind the type variable α over the body of the type abstraction σ, so the free type variables of $\backslash\alpha.\sigma$ are the free type variables of σ except for α. The kind of a type abstraction $\backslash\alpha.\sigma$ is an arrow kind of the form k_1 => k_2, where k_1 is the kind of α and k_2 is the kind of σ.

```
> ``:\'a.'a -> bool``;                                              26
val it = ``:\'a. 'a -> bool``
   : hol_type
> type_vars ``:\'a.'a -> bool``;
val it = [] : hol_type list
> type_vars ``:\'a.'a -> 'b``;
val it = [``:'b``] : hol_type list
> kind_of ``:\'a.'a -> bool``;
val it = ``::ty => ty`` : kind
> kind_of ``:\'a. num 'a``;
val it = ``::(ty => ty) => ty`` : kind
```

Beta- and eta-reduction of types Type abstractions can be applied to type arguments the same way that type operator constants are applied to type arguments. The new feature is that we now see beta-redexes, e.g. `(num (\'a.'a -> bool))`, completely within the type language. (Remember that in HOL-Omega's type language, function application is *postfix*, not prefix as in the term language.) The ML function `beta_conv_ty` will reduce a type that is such a beta-redex. This is done by substituting the type argument for the bound type variable within the type abstraction's body, and replacing the beta redex by the substituted body. Going further, the function `deep_beta_ty` will reduce all beta-redexes that are within a type, repeatedly until none are left.

The semantics of HOL-Omega is that a type which is a beta redex has the same meaning as the result of reducing the beta redex. Thus `(num (\'a.'a -> bool))` is identified with `num -> bool`, and so the result can always be used in place of the beta redex.

In a parallel fashion, types are also identified up to eta-reduction, so that `\'a.'a list` is identified with `list`. The ML function `eta_conv_ty` will reduce a type that is such an eta-redex, and the function `deep_eta_ty` will reduce all eta-redexes that are within a type, repeatedly until none are left. Usually, one will want to use the function `deep_beta_eta_ty` to reduce all beta- or eta-redexes within a type, completely.

In fact, the deep beta- and eta-reduction of types is performed automatically by the parser, so if one types in a type expression which has a beta redex in it, it will disappear before the time the type expression is printed. If there is a need for an un-beta-reduced type, one can create types with beta redexes in them by hand using `mk_app_type`.

The main thing to remember about type beta- and eta-reduction is, it should never be necessary for the user to worry about this. HOL-Omega will always treat the types correctly, whether or not they have been reduced, for all normal tasks like rewriting. While the result of such reduction is of course visually different from the starting type, both the original and the result types are equivalent according to the HOL-Omega logic.

```
> ``:num (\'a.'a -> bool)``;                                                      27
val it = ``:num -> bool`` : hol_type

> val ty1 = mk_app_type(``:\'a.'a -> bool``, ``:num``);
val ty1 =
   ``:num (\'a. 'a -> bool)``
   : hol_type
> beta_conv_ty ty1;
val it = ``:num -> bool`` : hol_type

> val ty2 = mk_app_type(``:list``,mk_app_type(``:option``,ty1));
val ty2 =
   ``:num (\'a. 'a -> bool) option list``
   : hol_type
> beta_conv_ty ty2 handle e => Raise e;

Exception raised at Type.beta_conv_ty:
not a type beta redex
Exception-
   HOL_ERR
  {message = "not a type beta redex", origin_function = "beta_conv_ty",
  origin_structure = "Type"} raised
> deep_beta_ty ty2;
val it =
   ``:(num -> bool) option list``
   : hol_type
```

Existential types There is one final variety of types, called *existential types*. Analogous to universal types, they are written as $?\alpha.\sigma$ or as $\exists\alpha.\sigma$, where α is a type variable and σ is a type expression. Like both universal types and type abstractions, the type variable α is bound over the body σ. The free type variables of the existential type are the free type variables of the body except for the bound type variable.

We mention existential types here for completeness, but the full meaning and usefulness of existential types is deferred until chapter 12 on packages. This advanced topic is best understood in a concentrated and focused manner.

Varieties of types Whereas in HOL there are two varieties of types, namely type variables and type combinations with zero or more arguments, in HOL-Omega these are replaced by six varieties: type constants, type variables, type applications, type abstractions, universal types, and existential types. What variety a type may be can be detected by the ML functions listed in the following table.

ML test functions to identify type varieties

Variety of type	*HOL$_\omega$ notation*	*Test function*
Type constant	τ	`is_con_type`
Type variable	α	`is_var_type`
Type application	$\sigma_{arg}\ \sigma_{opr}$	`is_app_type`
Type abstraction	`\`α`.`σ	`is_abs_type`
Universal type	`!`α`.`σ	`is_univ_type`
Existential type	`?`α`.`σ	`is_exist_type`

Type comparison Because of the introduction of bound type variables in type abstractions, universal types, and existential types, the comparison of two types for equality cannot be handled by ML equality as is done in the HOL system. For example, the types `\'a.'a -> 'a` and `\'b.'b -> 'b` are not the same by ML equality, but they are the same type in the HOL-Omega logic.

This comparison is properly tested by the ML function `eq_ty : hol_type -> hol_type -> bool`, and the use of ML equality for this test is deprecated in most cases. The exception is in those situations when one knows that at least one of the two types is a type variable; then ML equality will suffice to test for equality of types, because the kinds that are now part of the type variables may themselves be properly tested for equivalence by simple ML equality.

```
> ``:\'a.'a -> 'a`` = ``:\'b.'b -> 'b``;                     28
val it = false : bool
> eq_ty ``:\'a.'a -> 'a`` ``:\'b.'b -> 'b``;
val it = true : bool
> ``:'a : 'k => ty:1`` = ``:'b : 'k => ty:1``;
val it = false : bool
> ``:'a : 'k => ty:1`` = ``:'a : 'k => ty``;
val it = false : bool
```

Kind comparison When comparing two kinds to see if they are equivalent, the structure of kinds is simple enough that ML equality suffices.

However, when checking the application of a type operator to a type argument, or when checking if a type application term's bound type variable to its actual type argument, strict ML equality between kinds is not appropriate, because we wish to allow some flexibility with regards to ranks. If the type argument is of the same or a lower rank than expected, that is acceptable; the ranks do not have to be precisely equal.

This check is performed by the infix ML operator `:>=:`. This relation between kinds is defined recursively according to the following rules:

$$\frac{r_1 \geq r_2}{\texttt{ty} : r_1 \ \texttt{:>=:}\ \texttt{ty} : r_2} \qquad \frac{\kappa_1 = \kappa_2,\ \ r_1 = r_2}{\kappa_1 : r_1 \ \texttt{:>=:}\ \kappa_2 : r_2} \qquad \frac{k_1 = k_2,\ \ k_1' \ \texttt{:>=:}\ k_2'}{k_1 \texttt{ => } k_1' \ \texttt{:>=:}\ k_2 \texttt{ => } k_2'}$$

So a type operator σ of kind k_1 => k_2 can be applied as a function to a type argument τ of kind k_τ only if k_1 `:>=:` k_τ. For example, `list` cannot be applied to `option` because `list` expects a type of kind `ty`, and `option` has kind `ty => ty`.

```
> kind_of ``:option``;                                                    29
val it = ``::ty => ty`` : kind
> kind_of ``:list``;
val it = ``::ty => ty`` : kind
> fst(kind_dom_rng(kind_of ``:list``));
val it = ``::ty`` : kind
> fst(kind_dom_rng(kind_of ``:list``)) :>=: kind_of ``:option``;
val it = false : bool
```

A type operator can be applied to a type argument of a lower rank, but not to one of a higher rank.

```
> ``::ty:1`` :>=: ``::ty:0``;                                             30
val it = true : bool
> ``::ty:0`` :>=: ``::ty:1``;
val it = false : bool
```

If the domain of the kind of the type operator is itself an arrow kind, then it's domain has to be exactly equal to the domain of the kind of the type argument. The ranges can differ in rank, but not the domains.

```
> ``::ty:1 => ty:0`` :>=: ``::ty:0 => ty:0``;                             31
val it = false : bool
> ``::ty:0 => ty:1`` :>=: ``::ty:0 => ty:0``;
val it = true : bool
```

Rank comparison Rank comparisons are performed simply as ML comparisons between integers, whether equality, greater than, etc. Ranks are simply integers, except that they may never be negative.

Kind variables In type abstraction terms like `\:`α`. !x:`α`. x=x`, the kind of the bound type variable α is often `ty`, but in fact can have any kind in the HOL-Omega logic, since type abstractions are a fundamental form of the logic. As another example, in the term `\:`α`. !x:list `α`. x=x`, the bound type variable α has the kind `(ty => ty) => ty`.

Similarly, we would like to form type quantification terms like `!:`α`. !x:list `α`. x=x`. But type quantification terms are not fundamental forms of the HOL-Omega logic. Instead, they are applications of a term constant, either `!:` or `?:`, to a type abstraction term. `!:` and `?:` are new term constants defined in theory `bool`.

But just what is the type of `!:`? In the expression `!:`α`. !x:`α`. x=x`, the type of `!:` must be `(!'a:ty. bool) -> bool`. But in the expression `!:`α`. !x:list `α`. x=x`, the type of `!:` must be `(!'a:(ty => ty) => ty. bool) -> bool`. These two types are clearly not the same. How can one term constant have both these types?

```
> dest_comb ``!:'a. !x:'a. x=x``;                                32
val it =
   (``$!:``, ``\:'a. !x. x = x``)
   : term * term
> type_of (fst it);
val it = ``:(!'a. bool) -> bool``
   : hol_type
> dest_comb ``!:'a. !x:list 'a. x=x``;
val it =
   (``$!:``,
    ``\:'a :(ty => ty) => ty. !x. x = x``)
   : term * term
> type_of (fst it);
val it =
   ``:(!('a :(ty => ty) => ty). bool) -> bool``
   : hol_type
```

The answer is that the primitive type of `$!:` is `(!'a:'k. bool) -> bool`, which contains a *kind variable* `'k`. This kind variable has a rank, and may be instantiated with any kind of that rank or less, to form a *kind instance* of a type or term. Thus both of the type quantifications above are legal, with the kind variable providing the necessary flexibility.

```
> prim_mk_const{Thy = "bool", Name = "!:"};                     33
val it =
   ``($!: :(!('a :'k). bool) -> bool)``
   : term
> inst_kind [``::'k`` |-> ``::(ty => ty) => ty``] it;
val it =
   ``($!: :(!('a :(ty => ty) => ty). bool) -> bool)``
   : term
```

Kind variables are the third variety of kinds, along with the base kind `ty` and arrow kinds. The names of kind variables are like the names of type variables, in that they must start with an apostrophe ('). Kind variables also have a rank as an attribute, which limits what kinds may be substituted for the kind variable.

The rank of a base kind or of a kind variable is taken directly from the kind. The rank of an arrow kind is the maximum of the ranks of the arrow kind's domain and range.

Despite the names of kind variables and type variables looking the same, there is no confusion between them in the HOL-Omega logic. One could use the same name for both a type variable and a kind variable within the same expression without problems.

Varieties of kinds In HOL-Omega there are three varieties of kinds: the "type" kind `ty`, kind variables, and arrow kinds. What variety a kind may be can be detected by the ML functions listed in the following table.

ML test functions to identify kind varieties

Variety of kind	*HOL$_\omega$ notation*	*Test function*
Type kind	`ty`	`is_type_kind`
Kind variable	`'k, 'l, ...`	`is_var_kind`
Arrow kind	k_1 => k_2	`is_arrow_kind`

Universe polymorphism So we have seen how a type operator can be applied to type arguments of lower rank, but not of higher rank. This restriction is necessary for the simplicity of the semantics, but in practice it could have turned out to be quite restrictive indeed. For example, the `list` type operator has kind `ty => ty`, taking an argument which is a type of rank 0 to a result type of rank 0. This is fine for types of rank 0, like `bool` or `num`, but what if we wish to form lists where the type of the argument is `!'a.bool` of rank 1? Without any further flexibility, one would need to define a second version of `list`, say `list1 : ty:1 => ty:1`, and then for lists of elements of rank 2 one would need `list2`, and so on, an infinity of versions of the `list` type operator, each with their own distinct versions of `NIL` and `CONS` for each rank $0, 1, 2, \cdots$. We would also have to prove again each of the list theorems, duplicating the entire list library afresh for each new rank. This would be extremely cumbersome, but after finishing all this work, we would have gained no real new understanding or insight about our applications.

To overcome this practical problem, HOL-Omega contains a very powerful feature, that a type constant which is originally defined as one rank can have instances of higher rank. Thus the type constant `list` of kind `ty => ty` can be "promoted" to kind `ty:1 => ty:1`, or to `ty:2 => ty:2`, or to `ty:3 => ty:3`, etc., raising all of the ranks in the kind uniformly. This feature is called *universe polymorphism* or *rank polymorphism*.

This rank promotion is done automatically in the parser when there is a need to satisfy a rank restriction. The rank of the type constant is increased to whatever rank is necessary to be able to accept its argument, if possible. Such promotions are inferred automatically by the parser's rank inference algorithm.

```
> ``:(!'a. bool) list``;                                      34
val it = ``:(!'a. bool) list`` : hol_type
> fst(dest_app_type it);
val it = ``:list`` : hol_type
> kind_of it;
val it = ``::ty:1 => ty:1`` : kind
> rank_of it;
val it = 1 : rank
```

These are considered different instances of the original type constant, differing only in rank, and are thus called *rank instances*.

If we constrain the list type operator to be of rank 0, we prevent any promotion, and the result is that the list operator has insufficient rank for this argument:

```
> ``:(!'a. bool) (list:<=0)``;                                              35

Rank inference failure: unable to infer a rank for the application of

:list : ty => ty

on line 51, characters 20-23

which expects a type of rank 0

to

:!'a. bool

roughly on line 51, characters 9-12

which has rank 1

rank unification failure message: unify failed
Exception-
   HOL_ERR
  {message = "on line 51, characters 20-23:\nfailed", origin_function =
  "kindcheck", origin_structure = "Pretype"} raised
```

Just as type constants can be promoted to higher ranks than their original definitions, so can term constants. The parser will perform this promotion automatically when rank inference determines there is a need. Just as for types, these are considered different rank instances of the original term constants.

Substitutions One of the most important operations on the syntax of the HOL-Omega logic is the substitution of expressions for variables. Only free variables are affected by a substitution; all bound variables are unchanged. Just as for term substitutions, if a type substitution might cause the capture of a type variable, the corresponding bound type variable is automatically renamed. Because of the addition of kinds and ranks to the existing sorts of terms and types, there are four varieties of substitutions in HOL-Omega:

- term expressions for (free) term variables,
- type expressions for (free) type variables,
- kind expressions for kind variables, and

- a rank expression for the unique rank variable.

A rank substitution is represented by a simple nonnegative integer, say n. It stands for the substitution of the unique rank variable r_0 by the rank expression $r_0 + n$, thus raising by n all of the ranks in the object of the substitution.

The other varieties of substitutions are represented by ML lists of {`redex,residue`} pairs. Such pairs are easily created by the infix `|->` operator, which constructs a record with the two fields `redex` and `residue`. As a list, the substitution may contain 0, 1, or more such pairs, all of the same sort.

```
- [``x:num`` |-> ``y + 7``];                                         36
> val it = [{redex = ``x``, residue = ``y + 7``}] :
  {redex : term, residue : term} list
- [``:'a`` |-> ``:num -> bool``, ``:'b:ty => ty`` |-> ``:list``];
> val it =
    [{redex = ``:'a``, residue = ``:num -> bool``},
     {redex = ``:'b :ty => ty``, residue = ``:list``}] :
  {redex : hol_type, residue : hol_type} list
- [``::'k`` |-> ``::'l => ty``];
> val it = [{redex = ``::'k``, residue = ``::'l => ty``}] :
  {redex : kind, residue : kind} list
```

Proper substitutions In HOL, a substitution of term expressions for term variables requires that for each {`redex,residue`} pair, the redex and the residue have exactly the same type. In HOL-Omega, there are similar restrictions for term, type, and kind substitutions for them to be called *proper*. Proper substitutions have the welcome property that when they are applied to valid terms, types, or kinds, that the result is still a valid term, type, or kind, respectively. By "valid" here we mean that it is well-typed, well-kinded, and well-ranked. Proper substitutions maintain these well-formedness conditions, and that makes the semantics of substitution simple and its implementation efficient.

Substitutions of kind expressions for kind variables are proper only if for each redex and residue, the rank of the redex $\geq$ the rank of the residue.

Substitutions of type expressions for type variables are proper only if for each redex and residue, the kind of the redex `:>=:` the kind of the residue, where the ML operator `:>=:` was described earlier in the section on kind comparisons. So a lower-rank type expression may be substituted for a higher-rank type variable, as long as the kinds otherwise are the same.

This flexibility with regards to ranks is also extended to proper term substitutions, where the types of each redex and residue are compared using the ML operator `ge_ty`, which also includes the alpha, beta, and eta conversions of the types involved.

```
- ge_ty ``:!'a:ty:1. 'a -> 'b:ty:2`` ``:!'c:ty:1. 'c -> 'b:ty:1``;      37
> val it = true : bool
- ge_ty ``:!'a:ty:1. 'a -> 'b:ty:1`` ``:!'c:ty:1. 'c -> 'b:ty:2``;
> val it = false : bool
```

The relation ge_ty is defined by the following rules, where for clarity we use $\geq$ between types as an infix version of ge_ty. Here $[\alpha,\alpha'/\alpha',\alpha]$ swaps free occurrences of α and α'.

$$\frac{\alpha = \alpha',\ k = k'}{\alpha : k \ \geq\ \alpha' : k'} \qquad \frac{\sigma_{opr} \geq \sigma'_{opr},\ \sigma_{arg} \geq \sigma'_{arg}}{\sigma_{arg}\ \sigma_{opr} \ \geq\ \sigma'_{arg}\ \sigma'_{opr}} \qquad \frac{k = k',\ \sigma \geq \sigma'[\alpha,\alpha'/\alpha',\alpha]}{\forall \alpha{:}k.\ \sigma \ \geq\ \forall \alpha'{:}k'.\ \sigma'}$$

$$\frac{\tau = \tau',\ k \texttt{ :>=: } k'}{\tau : k \ \geq\ \tau' : k'} \qquad \frac{k = k',\ \sigma \geq \sigma'[\alpha,\alpha'/\alpha',\alpha]}{\lambda \alpha{:}k.\ \sigma \ \geq\ \lambda \alpha'{:}k'.\ \sigma'} \qquad \frac{k = k',\ \sigma \geq \sigma'[\alpha,\alpha'/\alpha',\alpha]}{\exists \alpha{:}k.\ \sigma \ \geq\ \exists \alpha'{:}k'.\ \sigma'}$$

$$\frac{\begin{array}{c}(\sigma_1,\ldots,\sigma_n)\tau \text{ is a } \textsf{head-humble} \text{ type},\ n \geq 0, \\ \text{name of } \tau = \text{name of } \tau',\ \forall i \in \{1,\ldots,n\}.\ \sigma_i \geq \sigma'_i\end{array}}{(\sigma_1,\ldots,\sigma_n)\tau \ \geq\ (\sigma'_1,\ldots,\sigma'_n)\tau'}$$

The last rule above refers to *head-humble* types, which are described next.

Humble types Consider the type `bool`. It contains exactly two values, `T` and `F`. It is a type of rank 0. If it is promoted to rank 1, it still contains the same two values. If it is promoted further, it still contains the same two values. Thus, when it is promoted its meaning and behavior do not change. Essentially all these types are the same type. We would like the HOL-Omega logic to recognize this, and consider all these the same type, for simplicity and easy of use.

By contrast, consider the type `!'a:ty:0.'a -> 'a`. It contains all functions that take first a type of kind `ty:0` and then a value of that type and return that same value. This is a type of rank 1. If it is promoted one rank, it becomes `!'a:ty:1.'a -> 'a`, which is a type of rank 2. It contains all functions that take first a type of kind `ty:1` and then a value of that type and return that same value. The important fact is that this is not the same set of functions as those contained by `!'a:ty:0.'a -> 'a`. It is much larger. We cannot consider these two types the same.

How do we distinguish those types that are unchanging under promotion from those that are not? By introducing the notion of *humble types*.

To support humble types, every type constant has a flag associated with it in the environment, indicating if the constant is humble or not.

Then a *head-humble type* is one whose kind is a type kind of some rank (ty:r), and which is of the form $(\sigma_1,\ldots,\sigma_n)\tau$ for $n \geq 0$, where τ is a type constant whose humble flag in the environment is true. That is, the type must simply be a series of zero or more type applications headed by a type constant, which is very like the traditional types of HOL. Note that a type constant alone is only a head-humble type if it has a kind ty:r.

Given this, we define a *humble type* as a head-humble type where all of the type arguments σ_i are humble types, for $1 \leq i \leq n$.

For this subset of all HOL-Omega types, we can know that their meaning and behavior will not change when they are promoted. Therefore the HOL-Omega logic identifies these promoted types as the same type, and this is reflected in the definition of ge_ty.

Applying substitutions The ML operators that apply substitutions are defined in the core structures of the HOL-Omega system, `Term`, `Type`, `Kind`, and `Rank`. Most of these operations can be used directly, but those on types or kinds may need to be qualified with the structure name, e.g. `Type.inst_rank` or `Kind.pure_inst_kind`.

ML operators to apply substitutions

	on Terms	*on Types*	*on Kinds*	*on Ranks*
Structure of opr.'s	`Term.`	`Type.`	`Kind.`	`Rank.`
Term substitution	`subst`	—	—	—
Type substitution	`pure_inst` `inst`	`pure_type_subst` `type_subst`	—	—
Kind substitution	`pure_inst_kind` `inst_kind`	`pure_inst_kind` `inst_kind`	`pure_inst_kind` `inst_kind`	—
Rank substitution	`inst_rank`	`inst_rank`	`inst_rank`	`promote`

In the above table, the dash (—) indicates no such substitution is possible. In table entries where there are two operator names, the one with "`pure_`" in the name indicates the normal substitution, and the other indicates an "aligning" substitution operation.

The aligning substitution operations perform the same as the pure ones if given a proper substitution as their argument. But they will often accept an improper substitution, which the pure version could not, and interpret it as a combination of substitutions, where the given substitution is analyzed to determine what "lower" sorts of auxiliary substitutions are needed to repair the given substitution, and make it proper.

For example, the type substitution θ = `[``:'a`` |-> ``:!'b.bool``]` is not proper; the rank of the residue `!'b.bool` is 1, which is greater than the rank of the redex `'a`, 0. Given $\theta$, the operator `pure_inst` will fail, raising an exception.

```
> val theta = [``:'a`` |-> ``:!'b.bool``];                          38
val theta = [{redex = ``:'a``, residue = ``:!'b. bool``}]
            : {redex: hol_type, residue: hol_type} list
> pure_inst theta ``[]:('a -> 'c) list`` handle e => Raise e;

Exception raised at Term.pure_inst:
kind of redex does not contain kind of residue
```

However, if this substitution is given to `inst`, it will create the auxiliary rank substitution 1, and repair the given θ to θ' = `[``:'a:ty:1`` |-> ``:!'b.bool``]`. Then `inst` will first apply the rank substitution 1, followed by $\theta'$. Any free occurrence of `'a` in the object term will first be lifted to `'a:ty:1`, and then successfully replaced by `!'b.bool`. Of course any other types in the object term will also be lifted by one rank.

```
> inst theta ``[]:('a -> 'c) list``;                                39
val it = ``([] :((!'b. bool) -> ('c :(ty:1))) list)`` : term
```

Type checking The type checking performed by the HOL-Omega system is considerably expanded from that of HOL. In addition to type checking to ensure that the expression is well-typed, the parser now performs kind checking and rank checking to ensure that the expression is well-kinded and well-ranked. Nevertheless, backwards compatibility has been maintained, so that virtually all expressions that parse correctly in HOL will also parse correctly and to the same results in HOL-Omega.

However, if the new types and terms of HOL-Omega are used, there are a few issues for the user to be aware of. First, because of the increased strength and complexity of the type language, type checking is now potentially incomplete. This means that type checking may fail to discover all the types of an expression's subterms and report an error, even if a suitable set of types might exist that would be consistent and correct. This is an inherent feature of the logic, and cannot be eliminated in general.

Consider the term ``M (M 3) = T``. This term is typeable in the logic. But the parser may not realize that the variable M should have a universal type, but instead conclude this is a typing error, and throw an exception. Even if a type argument is provided to M, indicating it has a universal type, the parser may still fail to correctly type the term.

40

```
> ``M [:bool:] (M 3) = T``;

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

(M :!'a. 'a -> 'a) [:bool:]

in compiler-generated text

which has type

:bool -> bool

to

(3 :num)

at line 33, character 16

unification failure message: unify failed
! Uncaught exception:
! HOL_ERR
> ``(M:!'a. 'a -> bool) [:bool:] (M 3) = T``;
val it =
    ``(M :!'a. 'a -> bool) [:bool:] (M [:num:] (3 :num)) = T``
      : term
```

The inherent incompleteness of type inference means that no type inference algorithm can correctly solve all cases. While the existing type inference algorithm implemented in HOL-Omega will work in many normal cases, neither it nor any possible improvement will solve all type inference situations properly.

However, there is a simple discipline that if the user will follow it, then the type checking should become complete. The discipline is as follows. For every term variable that appears in the expression to be parsed, if the term variable's intended type includes universal or existential types, then the user should annotate that term variable at one of its occurrences in the expression with a type constraint giving its type explicitly. If this discipline is followed, then if a solution exists, the type checking should complete correctly.

41
```
> ``(M:!'a.'a->bool) (M 3) = T``;
val it =
   ``(M :!'a. 'a -> bool) [:bool:] (M [:num:] (3 :num)) <=> T``
   : term
```

If the type annotations are of a size that is burdensome, then the type abbreviation facility of HOL-Omega can ease the task. Type abbreviations are not actual new types in the logic, but they are abbreviations that are parsed and printed as if they were types.

42
```
> type_abbrev ("iset", ``:!'a. 'a -> bool``);
val it = () : unit

> ``(M:iset) (M 3) = T``;
val it =
   ``(M :iset) [:bool:] (M [:num:] (3 :num)) <=> T``
   : term
```

Here is another example where a variable needs to be used with different types in the same expression. `M` seems to need to have the three different types `'a -> 'c`, `'b -> 'c`, and `'a -> 'b`, but a variable can only have one type within the same expression.

43
```
> ``!:'a 'b 'c. !(f:'a -> 'b) (g:'b -> 'c). M (g o f) = M g o M f``;
<<HOL message: inventing new type variable names: 'b>>

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

(M :('a -> 'c) -> 'b)

at line 58, character 54

to

(g :'b -> 'c)

at line 58, character 56

unification failure message: unify failed
```

Providing the universal type of M explicitly by a type coercion helps the type inference infer all types correctly.

```
> ``!:'a 'b 'c. !(f:'a -> 'b) (g:'b -> 'c).                          44
#     (M:!'a 'b. ('a->'b) -> 'a 'F -> 'b 'F) (g o f) = M g o M f``;
val it =
   ``!:'a 'b 'c.
    !(f :'a -> 'b) (g :'b -> 'c).
      (M :!'d 'e. ('d -> 'e) -> 'd ('F :ty => ty) -> 'e 'F) [:'a, 'c:]
        (g o f) =
      M [:'b, 'c:] g o M [:'a, 'b:] f``
   : term
```

Again, the use of type abbreviations can clarify the development.

```
> type_abbrev ("functor", ``:\'F. !'a 'b. ('a -> 'b) -> 'a 'F -> 'b 'F``); 45
val it = () : unit
> ``!:'a 'b 'c. !(f:'a -> 'b) (g:'b -> 'c).
#     (M:'F functor) (g o f) = M g o M f``;
val it =
   ``!:'a 'b 'c.
    !(f :'a -> 'b) (g :'b -> 'c).
      (M :('F :ty => ty) functor) [:'a, 'c:] (g o f) =
      M [:'b, 'c:] g o M [:'a, 'b:] f``
   : term
```

The kind system is simple enough that kind inference is complete and should always find the right kinds if possible. However, the rank system is more complex, as it involves natural numbers. The algorithm for rank inference in HOL-Omega is incomplete, so while it usually works successfully for most expressions, it is possible that rank inference for some expressions may fail even when a proper assignment of ranks does exist.

```
> ``:(!'b. 'b 'a) 'a``;                                              46

Rank inference failure: unable to infer a rank for the application of

:'a :ty => ty

on line 67, characters 16-17

which expects a type of rank 0

to

:!'b. 'b ('a :ty => ty)

roughly on line 67, characters 12-10

which has rank 1

rank unification failure message: unify failed
```

In such cases, the user will need to annotate some of his type variables or kinds with kind or rank constraints, respectively, to supply the information needed by the parser.

```
> ``:(!'b:ty:0. 'b 'a) ('a : ty:1 => ty:0)``;                          47
val it =
   ``:(!'b. 'b ('a :(ty:1 => ty))) 'a``
   : hol_type
```

5.2 Proof in HOL-Omega

The next sections discuss the new proof facilities in HOL-Omega. Since HOL-Omega is backwards compatible with HOL, every valid proof in HOL will also be a valid proof in HOL-Omega, and every tool that HOL provides for constructing proofs also works the same, given the same inputs, in HOL-Omega. However, because of the increased expressiveness of the HOL-Omega logic, many existing proof tools are extended in their function, and many new proof tools are added. This section will focus on these additions and extensions, expecting that the reader is already familiar with HOL. Many of the new facilities are analogs of HOL tools; for example, where a HOL tool dealt with quantification of term variables, there is a corresponding HOL-Omega tool that deals with quantification of type variables. We present the tools of HOL-Omega in stages: first the fundamental axioms and rules of inference, then some basic derived rules of inference, followed by more complex forward reasoning rules, then new tactics for backwards reasoning, and finally we describe broadly the extensions of the major library packages.

To begin with, where HOL has five axioms and eight primitive rules of inference, the HOL-Omega logic adds three new axioms, giving eight in all, and four new primitive inference rules, giving twelve in all. We'll get to the axioms in a moment, but let us first examine the new primitive rules of inference.

5.2.1 Primitive rules of inference

There are four new primitive rules of inference:

- `TY_ABS`
- `TY_BETA_CONV`
- `INST_RANK`
- `INST_KIND`

TY_ABS: The `TY_ABS` rule of inference is *type abstraction congruence*. In natural deduction notation this is:

$$\frac{\Gamma \;\vdash\; t_1 = t_2}{\Gamma \;\vdash\; (\lambda\alpha.\, t_1) = (\lambda\alpha.\, t_2)}$$

- where the type variable α is not free in Γ, and
- α is not free in any free term variable of t_1 or t_2.

This rule is represented in ML by a function `TY_ABS`, which is an analog of the HOL rule `ABS`. `TY_ABS` takes as arguments a type variable ``` ``:$\alpha$`` ``` and a theorem `|-` t_1 `=` t_2 and returns the theorem `|- (\:`α`.` t_1`) = (\:`α`.` t_2`)`. As expected, the type variable ``` ``:$\alpha$`` ``` will not be free in the resulting theorem.

```
> val th1 = combinTheory.K_DEF;                                        1
val th1 =
   |- (K :'a -> 'b -> 'a) = (\(x :'a) (y :'b). x)
   : thm
> val th2 = TY_ABS ``:'b`` th1;
val th2 =
   |- (\:'b. (K :'a -> 'b -> 'a)) = (\:'b. (\(x :'a) (y :'b). x))
   : thm
> val th3 = TY_ABS ``:'a`` th2;
val th3 =
   |- (\:'a 'b. (K :'a -> 'b -> 'a)) = (\:'a 'b. (\(x :'a) (y :'b). x))
   : thm
> type_vars_in_term (concl th3);
val it = [] : hol_type list
```

TY_BETA_CONV: The `TY_BETA_CONV` rule of inference is *type beta conversion*. In natural deduction notation this is:

$$\frac{}{\Gamma \vdash (\lambda\alpha.\ t)[\sigma] = t[\sigma/\alpha]}$$

- $t[\sigma/\alpha]$ denotes the result of substituting the type σ for free occurrences of the type variable α in t, where `kind_of` α `:>=: kind_of` σ, with the restriction that no free type variables in σ become bound after substitution.

This rule is represented in ML by a function `TY_BETA_CONV`, which is an analog of the HOL rule `BETA_CONV`. `TY_BETA_CONV` takes as an argument a term of the form `(\:`α`.` t`)[:`σ`:]` and returns the theorem `|- (\:`α`.` t`)[:`σ`:] =` $t[\sigma/\alpha]$.

```
> TY_BETA_CONV ``(\:'a. K:'a -> 'b -> 'a) [:'c -> 'd:]``;               2
val it =
   |- (\:'a. (K :'a -> 'b -> 'a)) [:'c -> 'd:] =
   (K :('c -> 'd) -> 'b -> 'c -> 'd)
   : thm
> TY_BETA_CONV ``(\:'a 'b. (\(x :'a) (y :'b). x)) [:'b -> 'c:]``;
val it =
   |- (\:'a 'd. (\(x :'a) (y :'d). x)) [:'b -> 'c:] =
   (\:'d. (\(x :'b -> 'c) (y :'d). x))
   : thm
```

INST_RANK: The `INST_RANK` rule of inference is *universe polymorphism* (or *rank polymorphism*) at the level of theorems. In natural deduction notation this is:

$$\frac{\Gamma \vdash t}{\Gamma[r_0 + r/r_0] \vdash t[r_0 + r/r_0]}$$

- Here r_0 is the one and only rank variable, and r is a nonnegative integer; and
- $t[r_0 + r/r_0]$ denotes the result of substituting the rank $r_0 + r$ for all occurrences of the rank variable r_0 in t, and $\Gamma[r_0 + r/r_0]$ denotes the result of substituting the rank $r_0 + r$ for all occurrences of the rank variable r_0 in Γ. This has the effect of increasing the ranks of all types and kinds by r. Note that r may legitimately be 0, which does not change the ranks at all.

This rule is represented in ML by a function `INST_RANK`, which has no analog in HOL. `INST_RANK` takes as arguments an integer r, which must be equal to or greater than zero, and a theorem $\Gamma \vdash t$, and returns the theorem $\Gamma[r_0 + r/r_0]$ `|-` $t[r_0 + r/r_0]$. This is a reflection of the fact that any mathematical development that was performed at some rank level, could have been performed just as well at the next rank one level up. Here r_0, the unique rank variable, is present but hidden and not printed in all HOL-Omega ranks. Thus the kind printed as `ty:1` is actually the kind `ty` at rank $r_0 + 1$. If this kind is instantiated with $[r_0 \mapsto r_0 + 2]$, then the rank $r_0 + 1$ is transformed to $(r_0+2)+1 = r_0+(2+1) = r_0+3$, and the effect is to raise the kind `ty:1` to `ty:3`. When this rank instantiation occurs, it must be done consistently throughout all of a theorem, including its hypotheses.

```
> set_trace "assumptions" 1;                                              3
val it = () : unit
> ASSUME ``xs = MAP (f:'a -> 'b) ys``;
val it =
    [(xs :'b list) = MAP (f :'a -> 'b) (ys :'a list)]
|- (xs :'b list) = MAP (f :'a -> 'b) (ys :'a list)
   : thm
> INST_RANK 3 it;
val it =
    [(xs :('b :(ty:3)) list) = MAP (f :('a :(ty:3)) -> 'b) (ys :'a list)]
|- (xs :('b :(ty:3)) list) = MAP (f :('a :(ty:3)) -> 'b) (ys :'a list)
   : thm
> set_trace "assumptions" 0;
val it = () : unit
```

INST_KIND: The `INST_KIND` rule of inference is *kind polymorphism* at the level of theorems. In natural deduction notation this is:

$$\frac{\Gamma \vdash t}{\Gamma[\theta] \vdash t[\theta]}$$

- where θ is a proper kind substitution, that is, a kind substitution of the form $[\kappa_1 \mapsto k_1,\ \kappa_2 \mapsto k_2,\ \ldots,\ \kappa_n \mapsto k_n]$ $(0 \leq n)$, where the κ_i are kind variables and the k_i are kinds, and where for each $i \in \{1, \ldots, n\}$, $\texttt{rank\_of}\ \kappa_i \geq \texttt{rank\_of}\ k_i$; and
- where $t[\theta]$ denotes the result of substituting the kind k_i for all occurrences of the kind variable κ_i in t, and $\Gamma[\theta]$ denotes the result of substituting the kind k_i for all occurrences of the kind variable κ_i in Γ, for all $i \in \{1, \ldots, n\}$.

This rule is represented in ML by a function INST_KIND, which has no analog in HOL. INST_KIND takes as arguments a kind substitution θ and a theorem $\Gamma \vdash t$, and returns the theorem $\Gamma[\theta]$ |- $t[\theta]$. When this kind instantiation occurs, it must be done consistently throughout all of a theorem, including its hypotheses.

Similarly, the existing primitive inference rules INST_TYPE and INST from HOL are modified to expect proper substitutions as arguments, using :>=: and ge_ty to check.

5.2.2 New axioms

HOL-Omega contains all of the axioms of HOL, and adds three more. Two of these new axioms, UNPACK_PACK_AX and PACK_ONTO_AX, have to do with packages, so we will delay their discussion until chapter 12, to deal with that subject completely at one time.

The remaining new axiom is the Law of Type Eta Conversion, which is bound to the ML name TY_ETA_AX. It is an analog of the HOL Law of Eta Conversion, ETA_AX.

```
> show_types := true;                                          4
val it = () : unit
> ETA_AX;
val it =
   |- !(t :'a -> 'b). (\(x :'a). t x) = t
   : thm
- TY_ETA_AX;
val it =
   |- !(t :!'a :'k. 'a ('b :('k => ty:1))). (\:'a :'k. t [:'a:]) = t
   : thm
```

Here the universally quantified variable t has type !'a :'k. 'a ('b :('k => ty:1)), a universal type binding the type variable 'a:'k over the body 'a ('b :('k => ty:1)). The symbol '\:' is the abstraction notation 'λ' for types over terms. The notation ''a:'k' means that the kind of the type variable 'a is the kind variable 'k, whose rank is the default rank, 0. The type variable 'b is a type operator, of kind 'k => ty:1, meaning it expects a type argument of kind 'k, and yields a type of kind ty:1 which has rank 1. Note that TY_ETA_AX has one free type variable, 'b, and one free kind variable, 'k.

The presence of the free kind variable 'k and free type operator variable 'b give this axiom the flexibility to apply to any possible instance of type eta redexes. To see how this works, consider the following example:

```
> val tm = ``\:'c. (m : !'d. 'd -> 'd list) [:'c:]``;                    5
val tm =
   ``\:'c. (m :!'d. 'd -> 'd list) [:'c:]``
   : term
> val th = REWRITE_CONV [TY_ETA_AX] tm;
val th =
   |- (\:'c. (m :!'d. 'd -> 'd list) [:'c:]) = m
   : thm
```

In this apparently simple example of rewriting and type eta reduction, there is a lot of detailed machinery going on quietly behind the scenes to accomplish this result. We will show an example of how this works here; the casual reader can skip this example if desired.

To successfully rewrite the term tm1 by the theorem TY_ETA_AX involves first matching the left-hand-side of TY_ETA_AX to the term tm1. The matching is done by the HOL-Omega function om_match_term, which is an expanded version of the HOL function match_term. It takes two terms as inputs, a pattern term and a target term, compares them, and returns a tuple of four substitutions to be applied together to the pattern term to make it the same as the target term.

```
> val ptm = lhs(snd(dest_forall(concl TY_ETA_AX)));                      6
val ptm =
   ``\:'a :'k. (t :!'a :'k. 'a ('b :('k => ty:1))) [:'a:]``
   : term

> set_trace "print_tyabbrevs" 0;
val it = () : unit
> val (tmS,tyS,kdS,rkS) = om_match_term ptm tm;
val kdS = [{redex = ``::'k``, residue = ``::ty``}] : (kind, kind) Term.subst
val rkS = 0 : rank
val tmS =
   [{redex = ``(t :!'a. 'a (\'d. 'd -> 'd list))``,
     residue = ``(m :!'d. 'd -> 'd list)``}]
   : (term, term) Term.subst
val tyS =
   [{redex = ``:'b :(ty => ty:1)``,
     residue = ``:\'d. 'd -> 'd list``}]
   : (hol_type, hol_type) Term.subst
```

These four substitutions are of different sorts; the first one in the tuple is on terms, the second on types, the third on kinds, and the last on ranks. Each substitution is guarranteed to be proper. (The rank substitution is represented simply by a integer, being how many ranks to promote the pattern term.) These substitutions are intended to be applied in a strict order, where the rank substitution is applied first, followed by the substitutions for kinds, types, and terms, in exactly that order.

In the example above, the rank substitution is 0, so the ranks do not change.

```
> rkS;                                                                   7
val it = 0 : rank
> val tm1 = inst_rank rkS ptm;
val tm1 =
   ``\:'a :'k. (t :!'a :'k. 'a ('b :('k => ty:1))) [:'a:]``
   : term
```

Next, `'k` is substituted by `ty`.

```
> kdS;                                                                   8
val it = [{redex = ``::'k``, residue = ``::ty``}] : (kind, kind) Term.subst
> val tm2 = inst_kind kdS tm1;
val tm2 =
   ``\:'a. (t :!'a. 'a ('b :(ty => ty:1))) [:'a:]``
   : term
```

Then `'b` of kind `ty => ty:1` is instantiated to `\'d:ty. 'd -> 'd list` of kind `ty => ty`. This is proper because even though these two kinds are not equal, they satisfy the relationship ``` ``::ty => ty:1`` :>=: ``::ty => ty`` ```.

```
> tyS;                                                                   9
val it =
   [{redex = ``:'b :(ty => ty:1)``,
     residue = ``:\'d. 'd -> 'd list``}]
   : (hol_type, hol_type) Term.subst
> val tm3 = inst tyS tm2;
val tm3 =
   ``\:'a. (t :!'a. 'a (\'d. 'd -> 'd list)) [:'a:]``
   : term
```

Finally the term variable `t`, which now has type `!'a. 'a (\'d. 'd -> 'd list)`, is substituted by the term `m : !'d. 'd -> 'd list`, which has the same type as `t` because types are identified up to alpha-beta-eta conversion in the type language.

```
> tmS;                                                                  10
val it =
   [{redex = ``(t :!'a. 'a (\'d. 'd -> 'd list))``,
     residue = ``(m :!'d. 'd -> 'd list)``}]
   : (term, term) Term.subst
> val tm4 = subst tmS tm3;
val tm4 =
   ``\:'a. (m :!'d. 'd -> 'd list) [:'a:]``
   : term
```

5.2.3 Basic derived rules of inference

Derived rules of inference are ML functions that are not primitive rules of inference, but are defined using the primitive rules of inference or other derived rules. Because of the

LCF architecture of HOL-Omega, all such derived rules are guaranteed to be sound if all of the primitive rules of inference and axioms are sound.

There is a family of rules that express the natural deduction rules for the new constructs of HOL-Omega at a basic level. Some have already been given (`TY_ABS` and `TY_BETA_CONV`). This section discusses 6 new basic rules of inference and one new instantiation rule. These are so foundational that even though they are theoretically derivable, for efficiency's sake they are implemented directly in the HOL-Omega kernel.

- `TY_COMB`
- `TY_SPEC` and `TY_GEN`
- `TY_EXISTS` and `TY_CHOOSE`
- `TY_EXT`
- `INST_ALL`

TY_COMB: The `TY_COMB` rule of inference is *congruence of term-type combinations*. In natural deduction notation this is:

$$\frac{\Gamma \ \vdash \ f = g}{\Gamma \ \vdash \ f[\sigma] = g[\sigma]}$$

- where $t[\sigma]$ denotes the application of the term t (which must have a universal type, say $\forall\alpha.\tau$) to the type argument σ, where σ and α must have the same kind.

```
TY_COMB : thm -> hol_type -> thm
```

This rule is represented in ML by a function `TY_COMB`, which takes as arguments a theorem `|- f = g` and a type σ, and returns the theorem `|- f [:`σ`:] = g [:`σ`:]`. This is an analog of HOL's term combination rule `AP_THM`, and similar to `MK_COMB`, considering that the equality of the type arguments to `f` and `g` is immediately decidable.[2]

```
> val th1 = REWRITE_CONV[TY_ETA_AX] ``\:'a. (\:'b. []:'b list) [:'a:]``;     11
val th1 =
    [] |- (\:'a. (\:'b. ([] :'b list)) [:'a:]) = (\:'b. ([] :'b list))
   : thm
> TY_COMB th1 ``:'c # 'd list``;
val it =
    []
|- (\:'a. (\:'b. ([] :'b list)) [:'a:]) [:'c # 'd list:] =
   (\:'b. ([] :'b list)) [:'c # 'd list:]
   : thm
```

[2]In a more complex type system, such as with dependent types, an analog to `MK_COMB` would need an input theorem like `|-` $\sigma = \tau$, and would produce the result `|-` f `[:`σ`:] =` g `[:`τ`:]`.

TY_SPEC: The TY_SPEC rule of inference is *type specialization* (or ∀-type-elimination). In natural deduction notation this is:

$$\frac{\Gamma \vdash \forall \alpha .\, t}{\Gamma \vdash t[\sigma/\alpha]}$$

- $t[\sigma/\alpha]$ denotes the result of substituting the type σ for free occurrences of the type variable α in t, where σ and α must have the same kind, and with the restriction that no free type variables in σ become bound after substitution.

This rule is represented in ML by a function `TY_SPEC`, which takes as arguments a type σ and a theorem `|- !:`$\alpha.t$, and returns the theorem `|-` $t[\sigma/\alpha]$, the result of substituting σ for α in t. This is an analog of HOL's universal term specialization rule `SPEC`.

TY_GEN: Another rule of inference is *type generalization* (or ∀-type-introduction). In standard natural deduction notation this is:

$$\frac{\Gamma \vdash t}{\Gamma \vdash \forall \alpha .\, t}$$

- This rule has the necessary restriction that α must not be free in the hypotheses Γ.

This rule is represented in ML by a function `TY_GEN`, which takes as arguments a type variable ``` ``: ```α``` `` ``` and a theorem `|-` t and returns the theorem `|- !:`$\alpha.t$. There is no compulsion that α should be free in t. This is an analog of HOL's universal term generalization rule `GEN`.

```
> val th1 = CONJUNCT1 listTheory.MAP;                                    12
val th1 =
    [] |- !(f :'a -> 'b). MAP f ([] :'a list) = ([] :'b list)
   : thm
> val th2 = TY_GEN ``:'b`` th1;
val th2 =
    [] |- !:'b. !(f :'a -> 'b). MAP f ([] :'a list) = ([] :'b list)
   : thm
> val th3 = TY_GEN ``:'a`` th2;
val th3 =
    [] |- !:'a 'b. !(f :'a -> 'b). MAP f ([] :'a list) = ([] :'b list)
   : thm
> val th4 = TY_SPEC ``:num`` th3;
val th4 =
    [] |- !:'b. !(f :num -> 'b). MAP f ([] :num list) = ([] :'b list)
   : thm
> val th5 = TY_SPEC ``:bool`` th4;
val th5 =
    [] |- !(f :num -> bool). MAP f ([] :num list) = ([] :bool list)
   : thm
```

TY_EXISTS: HOL-Omega also supports existential quantification of types over terms, including a rule of inference for *existential type generalization* (or ∃-type-introduction). In standard natural deduction notation this is:

$$\frac{\Gamma \ \vdash \ p[\sigma/\alpha]}{\Gamma \ \vdash \ \exists\alpha.\, p}$$

- where α must not be free in the hypotheses Γ, and
- $p[\sigma/\alpha]$ must be the same as the conclusion of the original theorem.

```
TY_EXISTS : term * hol_type -> thm -> thm
```

This rule is represented in ML by a function `TY_EXISTS`, which takes as arguments a pair of a term and a type, and then a theorem, where the term is a type-existentially quantified pattern indicating the desired form of the result ``?:$\alpha.\, p$``, the type ``:$\sigma$`` is the witness for the existential quantifier, and the input theorem has the form `|-` $p[\sigma/\alpha]$. `TY_EXISTS` returns the result theorem `|-` `?:`$\alpha.p$. This is an analog of HOL's existential term generalization rule `EXISTS`.

```
> CONJUNCT1 BOOL_EQ_DISTINCT;                                    13
val it =  [] |- T <=/=> F : thm
> EXISTS (``?y. T <=/=> y``, ``F``) it;
val it =  [] |- ?(y :bool). T <=/=> y : thm
> EXISTS (``?x y. x <=/=> y``, ``T``) it;
val it =
    [] |- ?(x :bool) (y :bool). x <=/=> y
   : thm
> TY_EXISTS (``?:'a. ?(x:'a) (y:'a). x <> y``, ``:bool``) it;
val it =
    [] |- ?:'a. ?(x :'a) (y :'a). x <> y
   : thm
```

TY_CHOOSE:

As a converse to the last rule, another rule of inference is *existential type specialization* (or ∃-type-elimination). In standard natural deduction notation this is:

$$\frac{\Gamma_1 \ \vdash \ \exists\alpha.s, \quad \Gamma_2 \cup \{s[\beta/\alpha]\} \ \vdash \ t}{\Gamma_1 \cup \Gamma_2 \ \vdash \ t}$$

- where the existentially specialized type variable β must not be free in Γ_1, Γ_2, or t.

```
TY_CHOOSE : hol_type * thm -> thm -> thm
```

This rule is represented in ML by a function `TY_CHOOSE`, which takes as arguments a pair of a type variable and a type-existential theorem, and then a second theorem, where the first theorem has the form `|-` $\exists\alpha.s$, and the type variable β is fresh, not appearing free in the theorems except for the single hypothesis $s[\beta/\alpha]$ in the second theorem. `TY_CHOOSE` returns a theorem with the conclusion of the second theorem, and whose hypotheses are those of the two theorems except for $s[\beta/\alpha]$, which is eliminated. This is an analog of HOL's existential term specialization rule `CHOOSE`.

TY_EXT: Another rule of inference is *type extensionality*. In standard natural deduction notation this is:

$$\frac{\Gamma\ \vdash\ \forall\alpha.\ t_1[\alpha] = t_2[\alpha]}{\Gamma\ \vdash\ t_1 = t_2}$$

- where $t[\alpha]$ denotes the application of the term t (which must have a universal type, say $\forall\beta.\tau$) to the type variable argument α, where α and β must have the same kind.

```
TY_EXT : thm -> thm
```

This rule is represented in ML by a function `TY_EXT`, which takes as argument a theorem of the form `|-` $\forall\alpha.$ `t1 [:`α`:] = t2 [:`α`:]`. `TY_EXT` returns a theorem of the form `|- t1 = t2`. This is an analog of HOL's extensionality rule `EXT`.

```
> ASSUME ``!:'a. (\:'b.?x:'b.T) [:'a:] = (\:'b.T) [:'a:]``;                14
val it =
    [.] |- !:'a. (\:'b. ?(x :'b). T) [:'a:] <=> (\:'b. T) [:'a:]
  : thm
> TY_EXT it;
val it =
    [.] |- (\:'b. ?(x :'b). T) = (\:'b. T)
  : thm
```

INST_ALL: This rule combines rank, kind, type, and term substitution efficiently.

```
INST_ALL :
      (term,term)subst * (hol_type,hol_type)subst * (kind,kind)subst * int
      -> thm -> thm
```

This rule is represented in ML by a function `INST_ALL`, which takes as arguments a tuple of substitutions for terms, types, kinds, and ranks, and a theorem, and returns a theorem where the substitutions have been uniformly performed on the theorem's hypotheses and conclusion, in the order of ranks first, then kinds, then types, and finally terms. `INST_ALL` $(\theta_t, \theta_\sigma, \theta_k, \theta_r)$ `th` is identical in effect to

`INST` θ_t `(INST_TYPE` θ_σ `(INST_KIND` θ_k `(INST_RANK` θ_r `th)))`.

INST_ALL may be more efficient than the above, as it minimizes the number of passes over the structure of the terms and types in th when performing the substitutions.

```
> val tmS = [``t:!'a. 'a -> 'a list`` |-> ``m:!'a. 'a -> 'a list``];          15
val tmS =
   [{redex = ``(t :!'a. 'a -> 'a list)``,
     residue = ``(m :!'a. 'a -> 'a list)``}]
   : {redex: term, residue: term} list
> val tyS = [``:'b: ty => ty:1`` |-> ``:\'a. 'a -> 'a list``];
val tyS =
   [{redex = ``:'b :(ty => ty:1)``,
     residue = ``:\'a. 'a -> 'a list``}]
   : {redex: hol_type, residue: hol_type} list
> val kdS = [``::'k`` |-> ``::ty``];
val kdS = [{redex = ``::'k``, residue = ``::ty``}]
   : {redex: kind, residue: kind} list
> val rkS = 0;
val rkS = 0 : int
> TY_ETA_AX;
val it =
   |- !(t :!'a :'k. 'a ('b :('k => ty:1))). (\:'a :'k. t [:'a:]) = t
   : thm
> INST_ALL (tmS,tyS,kdS,rkS) (SPEC_ALL TY_ETA_AX);
val it =
   |- (\:'a. (m :!'a. 'a -> 'a list) [:'a:]) = m
   : thm
> INST tmS(INST_TYPE tyS(INST_KIND kdS(INST_RANK rkS(SPEC_ALL TY_ETA_AX))));
val it =
   |- (\:'a. (m :!'a. 'a -> 'a list) [:'a:]) = m
   : thm
```

5.3 Backwards Proof

As in HOL, proof may be accomplished by first positing a goal to be proved, and then working backwards, applying tactics to reduce that goal to simpler subgoals. Then the same is done to each subgoal in turn, each generating zero or more new subgoals of its own. Hopefully, eventually each subgoal has been reduced to zero subgoals and is then solved, whereupon the subgoal package wraps the whole process up and creates an actual, accredited HOL-Omega theorem of the original goal.

We will discuss first some of the new, basic tactics that have been added in HOL-Omega, and then a number of existing tactics that have been suitably extended. This is only indicates some of the HOL-Omega additions; many more features have been added than can be covered in this tutorial.

5.3.1 New Basic Tactics

Here are some of the new tactics in HOL-Omega:

- TY_GEN_TAC
- TY_EXISTS_TAC

5.3.1.1 TY_GEN_TAC : tactic

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

$$\frac{\texttt{!:}\alpha\texttt{.}t[\alpha]}{t[\alpha'/\alpha]}$$

  Where α' is a variant of α not free in the goal or the assumptions. Here $t[\alpha]$ simply means that the type variable α may appear free in the term t, and $t[\alpha'/\alpha]$ means t with α' substituted for all the free occurrences of α.

- **Uses:** Solving type-universally quantified goals. REPEAT TY_GEN_TAC strips off all type-universal quantifiers. STRIP_TAC (see below) applies TY_GEN_TAC to type-universally quantified goals.

16

```
> g `!:'a. !x:'a. x = x`;
val it =
   Proof manager status: 1 proof.
1. Incomplete goalstack:
     Initial goal:

     !:'a. !(x :'a). x = x


   : proofs
> e(TY_GEN_TAC);
OK..
1 subgoal:
val it =
!(x :'a). x = x

   : proof
```

5.3.1.2 TY_EXISTS_TAC : hol_type -> tactic

- **Summary:** Supplies a witness for a type-existential quantifier.

$$\frac{\texttt{?:}\alpha\texttt{.}t[\alpha]}{t[\sigma/\alpha]}$$

Where σ is the first argument of TY_EXISTS_TAC. The type σ is provided as the explicit witness to satisfy the type-existential quantification.

- **Uses:** Solving type-existentially quantified goals.

17

```
> g `(!:'a. t[:'a:]) ==> (?:'a. t[:'a:])`;
val it =
   Proof manager status: 1 proof.
1. Incomplete goalstack:
     Initial goal:

     (!:'a. (t :!'a. bool) [:'a:]) ==> ?:'a. t [:'a:]



   : proofs
> e(DISCH_TAC);
OK..
1 subgoal:
val it =

?:'a. (t :!'a. bool) [:'a:]
------------------------------------
  !:'a. (t :!'a. bool) [:'a:]

   : proof
> e(TY_EXISTS_TAC ``:'a``);
OK..
1 subgoal:
val it =

(t :!'a. bool) [:'a:]
------------------------------------
  !:'a. (t :!'a. bool) [:'a:]

   : proof
> e(ASM_REWRITE_TAC[]);
OK..

Goal proved.
 [.] |- (t :!'a. bool) [:'a:]

Goal proved.
 [.] |- ?:'a. (t :!'a. bool) [:'a:]
val it =
   Initial goal proved.
|- (!:'a. (t :!'a. bool) [:'a:]) ==> ?:'a. t [:'a:]
   : proof
```

5.3.2 Extended Tactics

Many of the existing tactics of HOL have been extended to work with the new forms of HOL-Omega. In large part, this is due to the matching of terms and types being extended in the same way that has been seen previously, e.g., for rewriting. Thus the tactics `MATCH_ACCEPT_TAC` and `MATCH_MP_TAC` have the same behaviors as before, but also will successfully match their pattern terms against kind and rank instances.

In some cases a tactic's functionality has been extended beyond matching in ways that are natural for the new forms.

- `STRIP_TAC`
- `SIMP_TAC` and other simplifier tactics

5.3.2.1 `STRIP_TAC : tactic`

In HOL, `STRIP_TAC` combines the effects of `GEN_TAC`, `CONJ_TAC`, and `DISCH_TAC`, choosing which depending on what the current goal is. In addition, if a new assumption is added that is a conjunction, it is broken into two new assumptions, and if a new assumption is added that has an existential quantifier, that quantifier is stripped off and the bound variable renamed if necessary to be fresh.

In HOL-Omega, `STRIP_TAC` adds the effect of `TY_GEN_TAC`, and also, if a new assumption is added that has a type-existential quantifier, the quantifier is stripped off and the bound type variable renamed if necessary to be fresh.

5.3.2.2 `SIMP_TAC` and other simplifier tactics

The simplifier is too complex to be treated in any complete way in this tutorial, but it is appropriate to mention here that just as in HOL the core set of simplifications (`bool_ss`) performs beta reductions of terms, the core set of simplifications in HOL-Omega adds type-beta reductions of terms, essentially building in the effects of `TY_BETA_CONV`. In addition, just as in HOL the core set of simplifications contains simple reductions concerning universal and existential quantification, like `(!x:'a. t) = t`, the core set in in HOL-Omega adds the corresponding simple reductions concerning type-universal and type-existential quantification, such as `(!:'a:'k. t) = t`. Also, just as the HOL simplification set `ETA_ss` performs eta reductions of terms, so in HOL-Omega it also performs type eta reductions, essentially building in the effect of `TY_ETA_CONV`. As covered in chapter 12 on packages, the axiom `UNPACK_PACK_AX` is also added to the core simplification set, so that package reduction is automatically included as well.

Since these extensions are built-in as part of the core simplification set or `ETA_ss`, it is not necessary to create or mention any new simplification sets in order to access this additional simplifier functionality for proofs in HOL-Omega.

5.3.3 Other Rules, Tactics, and Automation

There are many more new inference rules, conversions, and tactics that have been added to HOL-Omega beyond what has been presented here. For more on these, the reader is directed to *DESCRIPTION*. As has been said, all the tools of HOL work just as before in HOL-Omega given the same inputs. In addition, in general the tools of HOL have been revised for HOL-Omega to be sensitive to and properly handle the new forms of the extended logic. In particular, this includes matching tactics like `MATCH_ACCEPT_TAC` and `MATCH_MP_TAC`, all the rewriting tactics, the resolution tactics `RES_TAC` and `IMP_RES_TAC`, the simplifier, and the datatype definition tools. There are some exceptions, e.g. the quotient library, as work continues on revising the extensive HOL library code. But for the most part, the existing facilities of HOL can be used without worry and they will generally just do the right thing. Examples of their use will be seen in the chapters that follow.

5.4 Backwards Compatibility

Significantly, despite all of this expansion of the logic, all of the tools perform exactly as before if given inputs in the original HOL logic. In particular, existing projects built using HOL should just build correctly in HOL-Omega, with only very slight and rare exceptions.

Of those rare exceptions, the most frequently encountered is a problem with parsing terms of the logic that have type annotations as part of a list of bound variable names. While in HOL it is fine to say

```
> ``\x:bool (lst:'a list). T``;                                              18
val it = ``\x lst. T`` : term
```

in HOL-Omega this causes a parsing error:

```
> ``\x:bool (lst:'a list). T``;                                              19
Exception-
   HOL_ERR
  {message =
  "on line 9, characters 11-13:\nType parsing failure with remaining input:
     lst:'a list). T",
  origin_function = "parse_type", origin_structure = "Parse"} raised
```

The reason for the parsing errors is that since the type language in HOL-Omega is more expressive, the type parser tries to interpret the term variable `(lst:'a list)` as if it were a type operator to apply to `bool`, which of course it is not.

Alternatively, whereas in HOL-Omega it is fine to say

```
> ``\x:bool (list). T``;                                                     20
val it = ``\x. T`` : term
```

but in HOL the name `list` is differently interpreted as a term variable name:

```
> ``\x:bool (list). T``;                                    21
<<HOL message: inventing new type variable names: 'a>>
val it = ``\x list. T`` : term
```

The real problem in both cases here is that the HOL-Omega type parser tries to reach farther than the HOL type parser in order to gather in the entire type, before looking for the next term variable in the list. The way to fix these sort of problems is simply to enclose the term variable with its type annotation within parentheses, as

```
> ``\(x:bool) (lst:'a list). T``;                           22
val it = ``\x lst. T`` : term
```

This is by far the most common issue in re-running HOL scripts in HOL-Omega, and even this issue has been extremely rare in practice, since most people already use parentheses around their variables with type annotations in such variable lists. The scarcity of such issues demonstrates the degree of backwards compatibility achieved.

Chapter 6

Example: Euclid's Theorem

In this chapter, we prove in HOL that for every number, there is a prime number that is larger, i.e., that the prime numbers form an infinite sequence. This proof has been excerpted and adapted from a much larger example due to John Harrison, in which he proved the $n = 4$ case of Fermat's Last Theorem. The proof development is intended to serve as an introduction to performing high-level interactive proofs in HOL.[1] Many of the details may be difficult to grasp for the novice reader; nonetheless, it is recommended that the example be followed through in order to gain a true taste of using HOL to prove non-trivial theorems.

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

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

Let's get started. First, we start the system, with the command `<holdir>/bin/hol`. We then "`open`" the arithmetic theory; this means that all of the ML bindings from the HOL theory of arithmetic are made available at the top level.

```
- open arithmeticTheory;                                          1
  ...
```

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

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

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

[1]The proofs discussed below may be found in `examples/euclid.sml` of the HOL distribution.

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

We want to treat `divides` as a (non-associating) infix:

```
- set_fixity "divides" (Infix(NONASSOC, 450));                    3
```

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

```
- val prime_def =                                                  4
    Define `prime p = ~(p=1) /\ !x. x divides p ==> (x=1) \/ (x=p)`;

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

That concludes the definitions to be made. Now we "just" have to prove that there are infinitely many prime numbers. If we were coming to this problem fresh, then we would have to go through a not-well-understood and often tremendously difficult process of finding the right lemmas required to prove our target theorem.[2] Fortunately, we are working from an already completed proof and can devote ourselves to the far simpler problem of explaining how to prove the required theorems.

Proof tools The development will illustrate that there is often more than one way to tackle a HOL proof, even if one has only a single (informal) proof in mind. In this example, we often *find* proofs by using the rewriter `RW_TAC` to unwind definitions and perform basic simplifications, often reducing a goal to its essence.

```
RW_TAC;                                                            5
val it = fn :simpset -> thm list -> term list * term ->
             (term list * term) list * (thm list -> thm)
```

The ML type of `RW_TAC` is `:simpset -> thm list -> tactic`.[3] When `RW_TAC` is applied to a *simpset*—for this example it will always be `arith_ss`—and a list of theorems, the theorems will be added to the simpset as supplementary rewrite rules. We will see that `arith_ss` is also somewhat knowledgeable about arithmetic.[4] Sometimes simplification

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

[3]Unfortunately, the MoscowML system does not print out the type of tactics in its abbreviated form.

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

with `RW_TAC` proves the goal immediately. Often however, we are left with a goal that requires some study before one realizes what lemmas are needed to conclude the proof. Once these lemmas have been proven, or located in ancestor theories, `METIS_TAC`[5] can be invoked with them, with the expectation that it will find the right instantiations needed to finish the proof. Note that these two operations, simplification and resolution-style automatic proof search, will not suffice to perform all the proofs in this example; in particular, our development will also need case analysis and induction.

Finding theorems This raises the following question: how does one find the right lemmas and rewrite rules to use? This is quite a problem, especially since the number of ancestor theories, and the theorems in them, is large. There are several possibilities

- The help system can be used to look up definitions and theorems, as well as proof procedures; for example, an invocation of

  ```
  help "arithmeticTheory"
  ```

  will display all the definitions and theorems that have been stored in the theory of arithmetic. However, the complete name of the item being searched for must be known before the help system is useful, so the following two search facilities are often more useful.

- `DB.match` allows the use of patterns to locate the sought-for theorem. Any stored theorem having an instance of the pattern as a subterm will be returned.

- `DB.find` will use fragments of names as keys with which to lookup information.

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

6.1 Divisibility

We start by proving a number of theorems about the `divides` relation. We will see that each of these initial theorems can be proved with a single invocation of `METIS_TAC`. Both

[5]`METIS_TAC` performs resolution-style first-order proof search.

RW_TAC and METIS_TAC are quite powerful reasoners, and the choice of a reasoner in a particular situation is a matter of experience. The major reason that METIS_TAC works so well is that divides is defined by means of an existential quantifier, and METIS_TAC is quite good at automatically instantiating existentials in the course of proof. For a simple example, consider proving $\forall x.\ x$ divides 0. A new proposition to be proved is entered to the proof manager via "g", which starts a fresh goalstack:

```
- g `!x. x divides 0`;                                              6

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

     : proofs
```

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

```
- e (RW_TAC arith_ss [divides_def]);                                7

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

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

```
- e (EXISTS_TAC ``0``);                                             8

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

Then a simplification step finishes the proof.

```
- e (RW_TAC arith_ss []);                                           9
OK..

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

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

What just happened here? The application of RW_TAC to the goal decomposed it to an empty list of subgoals; in other words the goal was proved by RW_TAC. Once a goal has been proved, it is popped off the goalstack, prettyprinted to the output, and the theorem becomes available for use by the level of the stack. When all the sub-goals required by *that* level are proven, the corresponding goal at that level can be proven too. This 'unwinding' process continues until the stack is empty, or until it hits a goal with more than one remaining unproved subgoal. This process may be hard to visualize,[6] but that doesn't matter, since the goalstack was expressly written to allow the user to ignore such details.

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

10

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

- e (RW_TAC arith_ss [divides_def] THEN EXISTS_TAC ``0``
     THEN RW_TAC arith_ss []);
OK..

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

We have seen one way to prove the theorem. However, as mentioned earlier, there is another: one can let METIS_TAC expand the definition of divides and find the required instantiation for x' from the theorem MULT_CLAUSES.[7]

11

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

- e (METIS_TAC [divides_def, MULT_CLAUSES]);
OK..
metis: r[+0+10]+0+0+0+1+2#
> val it =
    Initial goal proved.
    |- !x. x divides 0
```

As it runs, METIS_TAC prints out some possibly interesting diagnostics. In any case, having done our proof inside the goalstack package, we now want to have access to the theorem value that we have proved. We use the top_thm function to do this, and then use drop to dispose of the stack:

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

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

12

```
- val DIVIDES_0 = top_thm();

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

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

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

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

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

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

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

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

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

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

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

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

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

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

(*DIVIDES_LE*)

```
!m n. m divides n ==> m <= n \/ (n = 0)
-----------------------------------------
RW_TAC arith_ss [divides_def]
   THEN Cases_on `x`
   THEN RW_TAC arith_ss [MULT_CLAUSES]
```

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

```
- g `!m n . m divides n ==> m <= n \/ (n = 0)`;                      13
>   ...

- e (RW_TAC arith_ss [divides_def]);

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

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

```
- e (Cases_on `m`);                                                  14
OK..
2 subgoals:
> val it =
    SUC n <= SUC n * x \/ (SUC n * x = 0)

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

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

```
- e (RW_TAC arith_ss []);                                            15
OK..

Goal proved.
  ...

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

Let's try `RW_TAC` again:

```
- e (RW_TAC arith_ss []);                                              16
OK..
1 subgoal:
> val it =
    SUC n <= x * SUC n \/ (x = 0)
```

The right disjunct has been simplified; however, the left disjunct has failed to expand the definition of multiplication in the expression SUC $n * x$, which would have been convenient. In fact, it has changed it to $x *$ SUC n, which is inconvenient. Why, when `arith_ss` and hence `RW_TAC` is supposed to be expert in arithmetic? The answer is twofold: first, the recursive clauses for addition and multiplication are not in `arith_ss` because uncontrolled application of them by the rewriter seems, in general, to make some proofs *more* complicated, rather than simpler; second, the simplifier will rearrange arithmetical terms to make some automated proofs simpler. So the absence of the recursive clauses for multiplication means that SUC $n * x$ does not expand to $(n * x) + x$; instead, the rearrangement yields $x *$ SUC n. OK, so let's add in the definition of multiplication. This uncovers a new problem: how to locate this definition. The function

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

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

```
- DB.match ["arithmetic"] ``SUC n * x``;                               17

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

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

```
- DB.match [] ``SUC n * x = M``;                                          18

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

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

```
- b();                                                                    19
  ...

e (RW_TAC arith_ss [MULT_CLAUSES]);

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

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

```
- b();                                                                    20
> val it =
    SUC n <= SUC n * x \/ (SUC n * x = 0)

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


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

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

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

```
- e (Cases_on `x` THEN RW_TAC arith_ss [MULT_CLAUSES]);                    21
OK..

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

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

```
- val DIVIDES_LE = store_thm (                                              22
     "DIVIDES_LE",
     ``!m n. m divides n ==> m <= n \/ (n = 0)``,
     RW_TAC arith_ss  [divides_def]
       THEN Cases_on `x`
       THEN RW_TAC arith_ss  [MULT_CLAUSES]);

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

Storing theorems in our script record of the session in this style (rather than with the goalstack) results in a more concise script, and also makes it easier to turn our script into a theory file, as we do in section 6.5.

6.1.1 Divisibility and factorial

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

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

A polished proof of *DIVIDES_FACT* is the following[8]:

[8]This and subsequent proofs use the theorems proved on page 114, which were added to the ML environment after being proved.

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

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

```
- g `!m n. 0 < m /\ m <= n ==> m divides (FACT n)`;                    23

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

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

```
- LESS_EQ_EXISTS;                                                       24
> val it = |- !m n. m <= n = (?p. n = m + p) : thm

- e (RW_TAC arith_ss [LESS_EQ_EXISTS]);
OK..
1 subgoal:
> val it =
    m divides FACT (m + p)
    ------------------------------------
       0 < m
```

Now we induct on p:

```
- e (Induct_on ‘p‘);                                                    25
OK..
2 subgoals:
> val it =
    m divides FACT (m + SUC p)
    ------------------------------------
      0.  0 < m
      1.  m divides FACT (m + p)

    m divides FACT (m + 0)
   ------------------------------------
      0 < m
```

The first goal can obviously be simplified:

```
- e (RW_TAC arith_ss []);                                               26
OK..
1 subgoal:
> val it =
    m divides FACT m
    ------------------------------------
      0 < m
```

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

```
- e (Cases_on ‘m‘);                                                     27
OK..
2 subgoals:
> val it =
    SUC n divides FACT (SUC n)
    ------------------------------------
      0 < SUC n

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

Here the first sub-goal goal has an assumption that is false. We can demonstrate this to the system by using the DECIDE function to prove a simple fact about arithmetic (namely, that no number x is less than itself), and then passing the resulting theorem to METIS_TAC, which can combine this with the contradictory assumption.[9]

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

```
- e (METIS_TAC [DECIDE ``!x. ~(x < x)``]);                      28
OK..
metis: r[+0+4]#

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

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

Using the theorems identified above the remaining sub-goal can be proved with `RW_TAC`.

```
- e (RW_TAC arith_ss [FACT, DIVIDES_LMUL, DIVIDES_REFL]);       29
OK..

Goal proved.   ...

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

This last step, namely the invocation of `RW_TAC`, could also have been accomplished with `METIS_TAC`. Note that the only difference is the use of `DIVIDES_LMUL` in the simplifier *versus* `DIVIDES_RMUL` in `METIS_TAC`. This is due to the already mentioned algebraic rearrangement of arithmetical terms in the simplifier.

```
- b();                                                          30
> ...

- e (METIS_TAC [FACT, DIVIDES_RMUL, DIVIDES_REFL]);
OK..

Goal proved.  ...
```

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

```
- e (RW_TAC arith_ss [FACT, ADD_CLAUSES]);                      31

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

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

```
- e (METIS_TAC [DIVIDES_RMUL]);                                        32
OK..
metis: r[+0+5]+0+0+0+0+1#

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

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

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

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

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

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

```
- restart();                                                           33
> ...

- e (RW_TAC arith_ss [LESS_EQ_EXISTS]
       THEN Induct_on `p` THENL
       [RW_TAC arith_ss [] THEN Cases_on `m` THENL
          [METIS_TAC [DECIDE ``!x. ~(x < x)``],
           RW_TAC arith_ss [FACT, DIVIDES_LMUL, DIVIDES_REFL]],
        RW_TAC arith_ss [FACT, ADD_CLAUSES] THEN METIS_TAC [DIVIDES_RMUL]]);
OK..
metis: r[+0+5]+0+0+0+0+1#
metis: r[+0+4]#
> val it =
    Initial goal proved.
    |- !m n. 0 < m /\ m <= n ==> m divides FACT n
```

For many users, this would be the end of dealing with this proof: the tactic can now be packaged into an invocation of prove[10] or store_thm and that would be the end of it. However, another user might notice that this tactic could be shortened.

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

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

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

```
RW_TAC arith_ss [LESS_EQ_EXISTS]
  THEN Induct_on `p`
  THEN RW_TAC arith_ss [FACT, ADD_CLAUSES]
  THENL [Cases_on `m` THEN
           METIS_TAC[DECIDE ``!x. ~(x<x)``,FACT,DIVIDES_RMUL,DIVIDES_REFL],
         METIS_TAC [DIVIDES_RMUL]]
```

Finally, pushing this revisionism nearly to its limit, we'd like there to be only a single invocation of METIS_TAC to finish the proof off. The semantics of THEN and ALL_TAC

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

come to our rescue: we will split on the construction of m in the base case, as in the current incarnation of the tactic, but we will let the inductive case pass unaltered through the THENL. This is achieved by using ALL_TAC, which is a tactic that acts as an identity function on the goal.

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

The result is that there will be three subgoals emerging from the THENL: the two sub-cases in the base case and the unaltered step case. Each is proved with a call to METIS_TAC. Are we now done? Not necessarily. For example, the explicit case split, namely Cases_on ‘m‘, can be replaced by providing the *cases* theorem for natural numbers (num_CASES) to METIS_TAC. With this, the case split on m will be automatically generated by METIS_TAC as it searches for the proof. Hence we can shorten the tactic again.

```
- num_CASES;                                                          34
> val it = |- !m. (m = 0) \/ ?n. m = SUC n : thm

- restart();
- e (RW_TAC arith_ss [LESS_EQ_EXISTS]
     THEN Induct_on ‘p‘
     THEN METIS_TAC [DECIDE ‘‘!x. ~(x < x)‘‘, FACT, num_CASES,
                     DIVIDES_RMUL, DIVIDES_LMUL, DIVIDES_REFL, ADD_CLAUSES]);
```

We have now finished our exercise in tactic revision. Certainly, it would be hard to foresee that this final tactic would prove the goal; the required lemmas for the final invocation of METIS_TAC have been found by an incremental process of revision.

6.1.2 Divisibility and factorial (again!)

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

```
- restart();                                                                35

- e (Induct_on `n - m`);

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

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

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

```
- b();                                                                      36
  ...

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

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

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

Looking at the first goal, we reason that if $0 = n - m$ and $m \leq n$, then $m = n$. We can prove this fact, using `DECIDE_TAC`[11] and add it to the hypotheses by use of the infix operator "`by`":

[11]`DECIDE_TAC` is a decision procedure for a useful class of arithmetical formulas.

```
- e (`m = n` by DECIDE_TAC);                                     37
OK..
1 subgoal:
> val it =
    m divides FACT n
    ------------------------------------
      0.  0 = n - m
      1.  0 < m
      2.  m <= n
      3.  m = n
```

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

```
- e (RW_TAC arith_ss []);                                        38

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

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

```
    `?k.  m = SUC k` by <tactic>
```

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

When such situations occur, it is often best to start a new proof for the required lemma. This can be done simply by invoking "g" again. A new goalstack will be created and stacked upon the current one[12] and an overview of the extant proof attempts will be printed:

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

```
- g ‘!m. 0 < m ==> ?k. m = SUC k‘;                                      39

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


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

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

Our new goal can be proved quite quickly. Once we have proved it, we can bind it to an ML name and use it in the previous proof, by some sleight of hand with the “`before`”[13] function.

```
- e (Cases THEN RW_TAC arith_ss []);                                    40

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

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

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

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

```
- p ();                                                                 41

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

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

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

```
- e (‘?k. m = SUC k‘ by                                                    42
     METIS_TAC[lem] THEN RW_TAC arith_ss [FACT, DIVIDES_LMUL, DIVIDES_REFL]);
OK..
metis: r[+0+6]+0+0+0+0+0+1#

Goal proved.   ...

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

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

```
- e (‘0 < n‘ by DECIDE_TAC THEN ‘?k. n = SUC k‘ by METIS_TAC [lem]);      43
OK..
metis: r[+0+8]+0+0+0+0+0+0+2#
1 subgoal:
> val it =
    m divides FACT n
    ------------------------------------
      0.  !n m. (v = n - m) ==> 0 < m /\ m <= n ==> m divides FACT n
      1.  SUC v = n - m
      2.  0 < m
      3.  m <= n
      4.  0 < n
      5.  n = SUC k
```

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

```
- e (RW_TAC arith_ss [FACT, DIVIDES_RMUL]);                                44
OK..

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

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

6.2 Primality

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

(*NOT_PRIME_0*)
```
~prime 0
-------------------------------------
RW_TAC arith_ss [prime_def,DIVIDES_0]
```

(*NOT_PRIME_1*)
```
~prime 1
---------------------------
RW_TAC arith_ss [prime_def]
```

(*PRIME_2*)
```
prime 2
---------------------------------------------------------------
RW_TAC arith_ss [prime_def]
 THEN METIS_TAC [DIVIDES_LE, DIVIDES_ZERO,
                 DECIDE ``~(2=1)``, DECIDE ``~(2=0)``,
                 DECIDE ``x <= 2 = (x=0) \/ (x=1) \/ (x=2)``]
```

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

6.3 Existence of prime factors

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

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

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

```
- g ‘!n. ~(n = 1) ==> ?p. prime p /\ p divides n‘;                    45

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

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

```
- e (RW_TAC arith_ss [] THEN Cases_on ‘prime n‘);                      46
OK..
2 subgoals:
> val it =
    ?p. prime p /\ p divides n
    ------------------------------------
      0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
      1.  ~(n = 1)
      2.  ~prime n

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

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

```
- e (METIS_TAC [DIVIDES_REFL]);                                        47
OK..
metis: r[+0+7]+0+0+0+0+1#

Goal proved.  ...
```

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

```
- e (`?x. x divides n /\ ~(x=1) /\ ~(x=n)` by METIS_TAC [prime_def]);     48
OK..
metis: r[+0+11]+0+0+0+0+0+0+1+1+1+1+0+1+1#
1 subgoal:
> val it =
    ?p. prime p /\ p divides n
    ------------------------------------
      0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
      1.  ~(n = 1)
      2.  ~prime n
      3.  x divides n
      4.  ~(x = 1)
      5.  ~(x = n)
```

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

```
- e (`x < n \/ (n=0)` by METIS_TAC [DIVIDES_LE,LESS_OR_EQ]);     49
OK..
metis: r[+0+14]+0+0+0+0+0+0+0+0+0+0+1+0+1#
2 subgoals:
> val it =
    ?p. prime p /\ p divides n
    ------------------------------------
      0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
      1.  ~(n = 1)
      2.  ~prime n
      3.  x divides n
      4.  ~(x = 1)
      5.  ~(x = n)
      6.  n = 0

    ?p. prime p /\ p divides n
    ------------------------------------
      0.  !m. m < n ==> ~(m = 1) ==> ?p. prime p /\ p divides m
      1.  ~(n = 1)
      2.  ~prime n
      3.  x divides n
      4.  ~(x = 1)
      5.  ~(x = n)
      6.  x < n
```

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

```
- e (METIS_TAC [DIVIDES_TRANS]);                                       50
OK..
metis: r[+0+11]+0+0+0+0+0+0+0+1+0+4+1+0+3+0+2+2+1#

Goal proved.
```

The remaining goal can be clarified by simplification:

```
- e (RW_TAC arith_ss  []);                                             51
OK..
1 subgoal:
> val it =
    ?p. prime p /\ p divides 0
    ------------------------------------
      0.  !m. m < 0 ==> ~(m = 1) ==> ?p. prime p /\ p divides m
      1.  ~(0 = 1)
      2.  ~prime 0
      3.  x divides 0
      4.  ~(x = 1)
      5.  ~(x = 0)

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

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

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

```
- e (METIS_TAC [PRIME_2]);                                             52
OK..
metis: r[+0+6]#

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

Again, work now needs to be done to compose and perhaps polish a single tactic from the individual proof steps, but we will not describe it.[14] Instead we move forward, because our ultimate goal is in reach.

6.4 Euclid's theorem

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

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

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

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

```
- g `!n. ?p. n < p /\ prime p`;                                   53

- e (SPOSE_NOT_THEN STRIP_ASSUME_TAC);
OK..
1 subgoal:
> val it =
    F
    ------------------------------------
      !p. n < p ==> ~prime p
```

Thus we have the hypothesis that all p beyond a certain unspecified n are not prime, and our task is to show that this cannot be. At this point we take advantage of Euclid's

[14]Indeed, the tactic can be simplified into complete induction followed by an invocation of METIS_TAC with suitable lemmas.

great inspiration and we build an explicit term from n. In the informal proof we are asked to 'consider' the term FACT $n + 1$.[15] This term will have certain properties (i.e., it has a prime factor) that lead to contradiction. Question: how do we 'consider' this term in the formal HOL proof? Answer: by instantiating a lemma with it and bringing the lemma into the proof. The lemma and its instantiation are:[16]

```
- PRIME_FACTOR;                                                            54
> val it = |- !n. ~(n = 1) ==> (?p. prime p /\ p divides n) : thm

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

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

```
- e (MP_TAC (SPEC ``FACT n + 1`` PRIME_FACTOR));                           55
OK..
1 subgoal:
> val it =
    (~(FACT n + 1 = 1) ==> ?p. prime p /\ p divides FACT n + 1) ==> F
    ------------------------------------
      !p. n < p ==> ~prime p
```

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

```
- e (RW_TAC arith_ss []);                                                  56
OK..
2 subgoals:
> val it =
    ~prime p \/ ~(p divides FACT n + 1)
    ------------------------------------
      0.  !p. n < p ==> ~prime p
      1.  prime p

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

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

[15]The HOL parser thinks FACT $n + 1$ is equivalent to (FACT $n) + 1$.

[16]The function SPEC implements the rule of universal specialization.

```
- e (RW_TAC arith_ss [FACT_LESS, DECIDE ``!x. ~(x=0) = 0 < x``]);     57
OK..
Goal proved.  ...
```

Notice the 'on-the-fly' use of DECIDE to provide an *ad hoc* rewrite. Looking at the remaining goal, one might think that our aim, to prove falsity, has been lost. But this is not so: a goal $\neg P \vee \neg Q$ is logically equivalent to $P \Rightarrow Q \Rightarrow$ F. In the following invocation, we use the equality $\vdash A \Rightarrow B = \neg A \vee B$ as a rewrite rule oriented right to left by use of GSYM.[17]

```
- IMP_DISJ_THM;                                                        58
> val it = |- !A B. A ==> B = ~A \/ B : thm

- e (RW_TAC arith_ss [GSYM IMP_DISJ_THM]);
OK..
1 subgoal:
> val it =
    ~(p divides FACT n + 1)
    ------------------------------------
      0.  !p. n < p ==> ~prime p
      1.  prime p
     : goalstack
```

We can quickly proceed to show that p divides (FACT n), and thus that $p = 1$, hence that p is not prime, at which point we are done. This can all be packaged into a single invocation of METIS_TAC:

```
- e (METIS_TAC [DIVIDES_FACT, DIVIDES_ADDL, DIVIDES_ONE,               59
                NOT_PRIME_1, NOT_LESS, PRIME_POS]);
metis: r[+0+12]+0+0+0+0+0+0+0+1+1+0+0+0+0+1+1+1+1+4#

Goal proved.
 [..] |- ~(p divides FACT n + 1)

Goal proved.
 [.] |- ~prime p \/ ~(p divides FACT n + 1)

Goal proved.
 [.]
|- (~(FACT n + 1 = 1) ==> ?p. prime p /\ p divides FACT n + 1) ==> F

Goal proved.
 [.] |- F
> val it =
    Initial goal proved.
    |- !n. ?p. n < p /\ prime p : goalstack
```

[17] Loosely speaking, GSYM swaps the left and right hand sides of any equations it finds.

Euclid's theorem is now proved, and we can rest. However, this presentation of the final proof will be unsatisfactory to some, because the proof is completely hidden in the invocation of the automated reasoner. Well then, let's try another proof, this time employing the so-called 'assertional' style. When used uniformly, this can allow a readable linear presentation that mirrors the informal proof. The following proves Euclid's theorem in the assertional style. We think it is fairly readable, certainly much more so than the standard tactic proof just given.[18]

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

6.5 Turning the script into a theory

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

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

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

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

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

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

```
$ cd examples/                                                   1
$ ../bin/hol < euclid.sml
  ...

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

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

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

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

```
    open HolKernel boolLib Parse bossLib
```

The next line that poses a difficulty is

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

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

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

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

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

and at the end of the file:

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

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

```
$ mkdir euclid                                                    2
$ mv euclidScript.sml euclid
$ cd euclid
$ ../../bin/Holmake
Analysing euclidScript.sml
Trying to create directory .HOLMK for dependency files
Compiling euclidScript.sml
Linking euclidScript.uo to produce theory-builder executable
<<HOL message: Created theory "euclid".>>
Definition has been stored under "divides_def".
Definition has been stored under "prime_def".
Meson search level: .....
Meson search level: .................
 ...
Exporting theory "euclid" ... done.
Analysing euclidTheory.sml
Analysing euclidTheory.sig
Compiling euclidTheory.sig
Compiling euclidTheory.sml
```

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

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

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

6.6 Summary

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

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

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

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

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

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

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

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

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

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

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

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

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

Chapter 7

Example: a Simple Parity Checker

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

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

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

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

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

7.1 Introduction

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

7.2 Specification

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

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

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

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

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

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

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

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

and

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

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

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

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

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

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

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

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

```
- expand(REPEAT GEN_TAC THEN STRIP_TAC);                                 3
OK..
1 subgoal:
> val it =
    !t. out t = PARITY t inp
    ------------------------------------
      0.  out 0 = T
      1.  !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
```

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

```
- expand Induct;                                                          4
OK..
2 subgoals:
> val it =
    out (SUC t) = PARITY (SUC t) inp
    ------------------------------------
      0.  out 0 = T
      1.  !t. out (SUC t) = (if inp (SUC t) then ~out t else out t)
      2.  out t = PARITY t inp

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

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

```
- expand(ASM_REWRITE_TAC[PARITY_def]);                                    5
OK..

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

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

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

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

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

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

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

The ML function

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

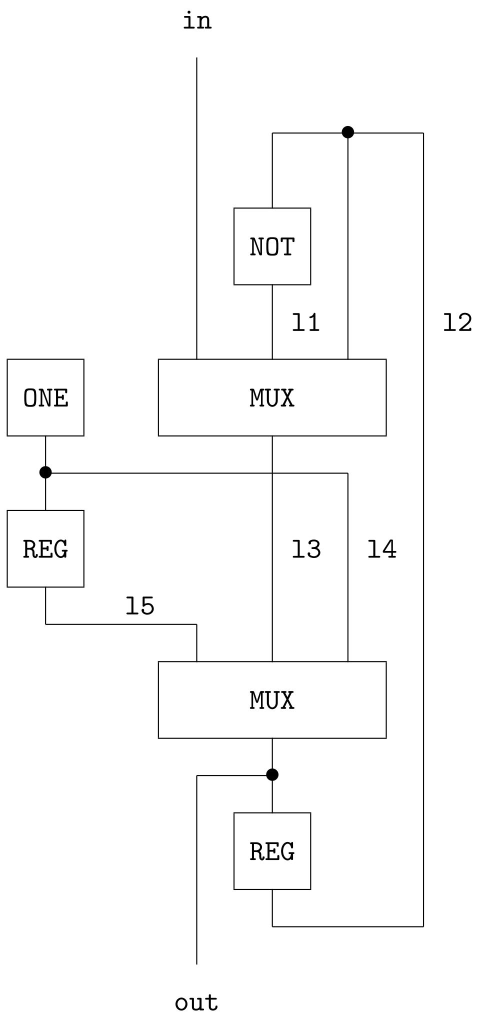
returns the theorem just proved (i.e. on the top of the theorem stack) in the current theory, and we bind this to the ML name `UNIQUENESS_LEMMA`.

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

7.3 Implementation

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

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



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

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

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

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

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

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

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

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

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

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

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

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

7.4 Verification

The following theorem will eventually be proved:

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

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

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

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

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

```
- expand(PURE_REWRITE_TAC                                                14
           [PARITY_IMP_def, ONE_def, NOT_def, MUX_def, REG_def] THEN
         REPEAT STRIP_TAC);
OK..
2 subgoals:
> val it =
    out (SUC t) = (if inp (SUC t) then ~out t else out t)
    ____________________________________
      0.  !t. l1 t = ~l2 t
      1.  !t. l3 t = (if inp t then l1 t else l2 t)
      2.  !t. l2 t = (if t = 0 then F else out (t - 1))
      3.  !t. l4 t = T
      4.  !t. l5 t = (if t = 0 then F else l4 (t - 1))
      5.  !t. out t = (if l5 t then l3 t else l4 t)

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

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

```
- expand(PROVE_TAC []);                                                  15
OK..
Meson search level: .....

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

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

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

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

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

```
- e (PAT_ASSUM ``!t. out t = X t``                                    16
       (fn th => REWRITE_TAC [SPEC ``SUC t`` th]));
<<HOL message: inventing new type variable names: 'a, 'b.>>
OK..
1 subgoal:
> val it =
    (if l5 (SUC t) then l3 (SUC t) else l4 (SUC t)) =
    (if inp (SUC t) then ~out t else out t)
    ------------------------------------
      0.  !t. l1 t = ~l2 t
      1.  !t. l3 t = (if inp t then l1 t else l2 t)
      2.  !t. l2 t = (if t = 0 then F else out (t - 1))
      3.  !t. l4 t = T
      4.  !t. l5 t = (if t = 0 then F else l4 (t - 1))
```

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

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

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

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

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

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

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

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

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

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

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

This completes the proof of the parity checking device.

7.5 Exercises

Two exercises are given in this section: Exercise 1 is straightforward, but Exercise 2 is quite tricky and might take a beginner several days to solve.

7.5.1 Exercise 1

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

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

This is formalized in HOL by the definition:

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

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

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

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

[4]i.e. a definition of the same form as that of PARITY_IMP on page 148.

7.5.2 Exercise 2

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

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

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

3. Verify the correctness of your implementation in HOL.

Chapter 8

Example: Combinatory Logic

8.1 Introduction

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

The HOL sessions assume that the Unicode trace is *on* (as it is by default), meaning that even though the inputs may be written in pure ASCII, the output still uses nice Unicode output (symbols such as ∀ and ⇒). The Unicode symbols could also be used in the input.

8.2 The type of combinators

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

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

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

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

8.3 Combinator reductions

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

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

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

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

$$\frac{x \;\to\; x'}{x\,y \;\to\; x'\,y}$$

$$\frac{y \;\to\; y'}{x\,y \;\to\; x\,y'}$$

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

```
- set_fixity "-->" (Infix(NONASSOC, 450));                          3
> val it = () : unit
```

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

Our next step is to actually define the relation with the `xHol_reln` function. In addition to a quotation specifying the rules for the new relation, it requires a name to use as the stem for the theorems it proves. We pick the string `"redn"`.[1] The `xHol_reln` function for doing this returns three separate theorems, and we bind the first (the "rules" theorem) and third (the "cases" theorem):

[1]The related function `Hol_reln` can be used if the system's choice of stem is acceptable. In this case, `Hol_reln` can't cope with the non-alphanumeric characters in `-->` and will raise an error.

```
val (redn_rules, _, redn_cases) = xHol_reln "redn"                    4
   `(!x y f. x --> y   ==>    f # x --> f # y) /\
    (!f g x. f --> g   ==>    f # x --> g # x) /\
    (!x y.   K # x # y --> x) /\
    (!f g x. S # f # g # x --> (f # x) # (g # x))`;
> val redn_rules =
    |- (∀x y f. x --> y ⇒ f # x --> f # y) ∧
       (∀f g x. f --> g ⇒ f # x --> g # x) ∧
       (∀x y. K # x # y --> x) ∧
       ∀f g x. S # f # g # x --> f # x # (g # x) : thm
  val redn_cases =
    |- ∀a0 a1.
         a0 --> a1 ⇔
         (∃x y f. (a0 = f # x) ∧ (a1 = f # y) ∧ x --> y) ∨
         (∃f g x. (a0 = f # x) ∧ (a1 = g # x) ∧ f --> g) ∨
         (∃y. a0 = K # a1 # y) ∨
         ∃f g x. (a0 = S # f # g # x) ∧ (a1 = f # x # (g # x))
    : thm
```

In addition to proving these three theorems for us, the inductive definitions package has also saved them to disk.

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

```
- PROVE [redn_rules] ``S # (K # x # x) --> S # x``;                   5
Meson search level: ...
> val it = |- S # (K # x # x) --> S # x : thm
```

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

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

8.4 Transitive closure and confluence

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

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

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

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

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

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

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

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

```
val _ = hide "RTC";                                                    6

val (RTC_rules, _, RTC_cases) = Hol_reln `
    (!x.     RTC R x x) /\
    (!x y z. R x y /\ RTC R y z ==> RTC R x z)`;
<<HOL message: inventing new type variable names: 'a>>
> val RTC_rules =
    |- ∀R. (∀x. RTC R x x) ∧
           ∀x y z. R x y ∧ RTC R y z ⇒ RTC R x z : thm
  val RTC_cases =
    |- ∀R a0 a1. RTC R a0 a1 ⇔ (a1 = a0) ∨
                               ∃y. R a0 y ∧ RTC R y a1 : thm
```

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

```
- val confluent_def = Define                                           7
    `confluent R =
       !x y z. RTC R x y /\ RTC R x z ==>
               ?u. RTC R y u /\ RTC R z u`;
```

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

$$\begin{tikzcd}
& x \arrow[dl, "*"'] \arrow[dr, "*"] & \\
y & & z
\end{tikzcd}$$

$$\begin{tikzcd}
& x \arrow[dl, "*"'] \arrow[dr, "*"] & \\
y \arrow[dr, dotted, "*"'] & & z \arrow[dl, dotted, "*"] \\
& u &
\end{tikzcd}$$

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

```
- val normform_def = Define‘normform R x = !y. ~(R x y)‘;                 8
<<HOL message: inventing new type variable names: ’a, ’b>>
Definition has been stored under "normform_def".
> val normform_def = |- ∀R x. normform R x ⇔ ∀y. ¬R x y : thm
```

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

We can now prove the following:

```
- g ‘!R. confluent R ==>                                                   9
         !x y z.
           RTC R x y /\ normform R y /\
           RTC R x z /\ normform R z ==> (y = z)‘;
<<HOL message: inventing new type variable names: ’a>>
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ∀R.
           confluent R ⇒
           ∀x y z. RTC R x y ∧ normform R y ∧ RTC R x z ∧ normform R z ⇒
                   (y = z)
```

We rewrite with the definition of confluence:

```
- e (RW_TAC std_ss [confluent_def]);                                      10
OK..
1 subgoal:
> val it =
    y = z
    ------------------------------------
      0.  ∀x y z. RTC R x y ∧ RTC R x z ⇒ ∃u. RTC R y u ∧ RTC R z u
      1.  RTC R x y
      2.  normform R y
      3.  RTC R x z
      4.  normform R z
```

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

```
- e (`?u. RTC R y u /\ RTC R z u` by PROVE_TAC []);                    11
OK..
Meson search level: .........
1 subgoal:
> val it =
    y = z
    ------------------------------------
      0.  ∀x y z. RTC R x y ∧ RTC R x z ⇒ ∃u. RTC R y u ∧ RTC R z u
      1.  RTC R x y
      2.  normform R y
      3.  RTC R x z
      4.  normform R z
      5.  RTC R y u
      6.  RTC R z u
```

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

```
- e (PROVE_TAC [normform_def, RTC_cases]);                              12
OK..
Meson search level: ..........

Goal proved. [...]
> val it =
    Initial goal proved.
    |- ∀R.
         confluent R ⇒
         ∀x y z.
           RTC R x y ∧ normform R y ∧ RTC R x z ∧ normform R z ⇒
           (y = z)
```

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

```
val confluent_normforms_unique = store_thm(                             13
  "confluent_normforms_unique",
  ``!R. confluent R ==>
        !x y z. RTC R x y /\ normform R y /\
                RTC R x z /\ normform R z ==> (y = z)``,
  RW_TAC std_ss [confluent_def] THEN
  `?u. RTC R y u /\ RTC R z u` by PROVE_TAC [] THEN
  PROVE_TAC [normform_def, RTC_cases]);
```

· · · ◇ · · ·

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

```
- val diamond_def = Define                                              14
    ‘diamond R = !x y z. R x y /\ R x z ==> ?u. R y u /\ R z u‘;
<<HOL message: inventing new type variable names: ’a>>
Definition has been stored under "diamond_def".
> val diamond_def =
    |- ∀R. diamond R ⇔ ∀x y z. R x y ∧ R x z ⇒ ∃u. R y u ∧ R z u
     : thm
```

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

```
val confluent_diamond_RTC = store_thm(                                  15
  "confluent_diamond_RTC",
  ‘‘!R. confluent R = diamond (RTC R)‘‘,
  RW_TAC std_ss [confluent_def, diamond_def]);
```

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

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

Graphically, this is hoping that from

$$\begin{tikzcd}
& x \arrow[dl] \arrow[dr] & \\
y \arrow[dr, dotted] & & z \arrow[dl, dotted] \\
& u &
\end{tikzcd}$$

we will be able to conclude

$$\begin{tikzcd}
& & x \arrow[dl] \arrow[dr] & & \\
& y \arrow[dr, dotted] \arrow[dl, dashed] & & z \arrow[dl, dotted] \arrow[dr, dashed] & \\
p \arrow[ddrr, dotted] & & u & & q \arrow[ddll, dotted] \\
& & & & \\
& & r & &
\end{tikzcd}$$

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

$$\begin{tikzcd}
 & & x \arrow[dl] \arrow[dr] & \\
 & y \arrow[dr, dotted] \arrow[ddl, dashed] & & z \arrow[dl, dotted] \\
 & & u \arrow[ddl, dotted] & \\
p \arrow[dr, dotted] & & & \\
 & r & &
\end{tikzcd}$$

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

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

```
- g `!R. diamond R ==>                                                  16
         !x p. RTC R x p ==>
               !z. R x z ==>
                   ?u. RTC R p u /\ RTC R z u`;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ∀R.
           diamond R ⇒
           ∀x p. RTC R x p ⇒ ∀z. R x z ⇒ ∃u. RTC R p u ∧ RTC R z u
```

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

```
- e (GEN_TAC THEN STRIP_TAC);                                           17
OK..
1 subgoal:
> val it =
    ∀x p. RTC R x p ⇒ ∀z. R x z ⇒ ∃u. RTC R p u ∧ RTC R z u
    ------------------------------------
      diamond R
```

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

Now we can use the induction principle for reflexive and transitive closure (alternatively, we perform a "rule induction"). To do this, we use the `Induct_on` command that is also used to do structural induction on algebraic data types (such as numbers and lists). We provide the name of the constant whose induction principle we want to use, and the tactic does the rest:

```
- e (Induct_on ‘RTC‘);                                                          18
OK..
1 subgoal:
> val it =
    (∀x z. R x z ⇒ ∃u. RTC R x u ∧ RTC R z u) ∧
    ∀x x’ p.
      R x x’ ∧ RTC R x’ p ∧ (∀z. R x’ z ⇒ ∃u. RTC R p u ∧ RTC R z u) ⇒
      ∀z. R x z ⇒ ∃u. RTC R p u ∧ RTC R z u
    ------------------------------------
      diamond R
```

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

```
- e (REPEAT STRIP_TAC);                                                         19
OK..
2 subgoals:
> val it =
    ∃u. RTC R p u ∧ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  R x x’
      2.  RTC R x’ p
      3.  ∀z. R x’ z ⇒ ∃u. RTC R p u ∧ RTC R z u
      4.  R x z

    ∃u. RTC R x u ∧ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  R x z
```

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

```
- e (PROVE_TAC [RTC_rules]);                                          20
OK..
Meson search level: .....

Goal proved. [..] |- ∃u. RTC R p u ∧ RTC R z u
Remaining subgoals:
> val it =
    ∃u. RTC R p u ∧ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  R x x'
      2.  RTC R x' p
      3.  ∀z. R x' z ⇒ ∃u. RTC R p u ∧ RTC R z u
      4.  R x z
```

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

```
- e (`?v. R x' v /\ R z v` by PROVE_TAC [diamond_def]);               21
OK..
Meson search level: ...........
1 subgoal:
> val it =
    ∃u. RTC R p u ∧ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  R x x'
      2.  RTC R x' p
      3.  ∀z. R x' z ⇒ ∃u. RTC R p u ∧ RTC R z u
      4.  R x z
      5.  R x' v
      6.  R z v
```

Now we can apply our induction hypothesis (assumption 3) to complete the long, lop-sided strip of the diamond. We will conclude that there is a u such that RTC $R\ p\ u$ and RTC $R\ v\ u$. We actually need a u such that RTC $R\ z\ u$, but because there is a single R-step between z and v we have that as well. All we need to provide `PROVE_TAC` is the rules for RTC:

```
- e (PROVE_TAC [RTC_rules]);                                          22
OK..
Meson search level: .......

Goal proved. [...]
> val it =
    Initial goal proved.
    |- ∀R.
         diamond R ⇒
         ∀x p. RTC R x p ⇒ ∀z. R x z ⇒ ∃u. RTC R p u ∧ RTC R z u
```

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

```
val R_RTC_diamond = store_thm(                                    23
  "R_RTC_diamond",
  ``!R. diamond R ==>
         !x p. RTC R x p ==>
               !z. R x z ==>
                   ?u. RTC R p u /\ RTC R z u``,
  GEN_TAC THEN STRIP_TAC THEN Induct_on `RTC` THEN
  REPEAT STRIP_TAC THENL [
    PROVE_TAC [RTC_rules],
    `?v. R x' v /\ R z v` by PROVE_TAC [diamond_def] THEN
    PROVE_TAC [RTC_rules]
  ]);
```

···◇···

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

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

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

```
- g `!R. diamond R ==> !x y. RTC R x y ==>                        24
                             !z. RTC R x z ==>
                                 ?u. RTC R y u /\ RTC R z u`;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ∀R.
           diamond R ⇒
           ∀x y. RTC R x y ⇒ ∀z. RTC R x z ⇒ ∃u. RTC R y u ∧ RTC R z u
```

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

```
- e (GEN_TAC THEN STRIP_TAC THEN Induct_on ‘RTC‘ THEN REPEAT STRIP_TAC);   25
OK..
2 subgoals:
> val it =
    ∃u. RTC R y u ∧ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  R x x’
      2.  RTC R x’ y
      3.  ∀z. RTC R x’ z ⇒ ∃u. RTC R y u ∧ RTC R z u
      4.  RTC R x z

    ∃u. RTC R x u ∧ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  RTC R x z
```

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

```
- e (PROVE_TAC [RTC_rules]);   26
OK..
Meson search level: ...

Goal proved. [...]

Remaining subgoals:
> val it =
    ∃u. RTC R y u ∧ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  R x x’
      2.  RTC R x’ y
      3.  ∀z. RTC R x’ z ⇒ ∃u. RTC R y u ∧ RTC R z u
      4.  RTC R x z
```

This goal is very similar to the one we saw earlier. We have the top of a (“lop-sided”) diamond in assumptions 1 and 4, so we can infer the existence of a common destination for x' and z:

```
- e (‘?v. RTC R x’ v /\ RTC R z v‘ by PROVE_TAC [R_RTC_diamond]);    27
OK..
Meson search level: ...........
1 subgoal:
> val it =
    ∃u. RTC R y u ∧ RTC R z u
    ------------------------------------
      0.  diamond R
      1.  R x x’
      2.  RTC R x’ y
      3.  ∀z. RTC R x’ z ⇒ ∃u. RTC R y u ∧ RTC R z u
      4.  RTC R x z
      5.  RTC R x’ v
      6.  RTC R z v
```

At this point in the last proof we were able to finish it all off by just appealing to the rules for RTC. This time it is not quite so straightforward. When we use the induction hypothesis (assumption 3), we can conclude that there is a u to which both y and v can connect in zero or more steps, but in order to show that this u is reachable from z, we need to be able to conclude RTC $R\ z\ u$ when we know that RTC $R\ z\ v$ (assumption 6 above) and RTC $R\ v\ u$ (our consequence of the inductive hypothesis). We leave the proof of this general result as an exercise, and here assume that it is already proved as the theorem RTC_RTC.

```
- e (PROVE_TAC [RTC_rules, RTC_RTC]);                                28
Meson search level: .......

Goal proved. [...]
> val it =
    Initial goal proved.
    |- ∀R.
         diamond R ⇒
         ∀x y. RTC R x y ⇒ ∀z. RTC R x z ⇒ ∃u. RTC R y u ∧ RTC R z u
```

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

```
val diamond_RTC_lemma = prove(                                       29
  ‘‘!R.
       diamond R ==>
       !x y. RTC R x y ==> !z. RTC R x z ==> ?u. RTC R y u /\ RTC R z u‘‘,
  GEN_TAC THEN STRIP_TAC THEN Induct_on ‘RTC‘ THEN
  REPEAT STRIP_TAC THENL [
    PROVE_TAC [RTC_rules],
    ‘?v. RTC R x’ v /\ RTC R z v‘ by PROVE_TAC [R_RTC_diamond] THEN
    PROVE_TAC [RTC_RTC, RTC_rules]
  ]);
val diamond_RTC = store_thm(
  "diamond_RTC",
  ‘‘!R. diamond R ==> diamond (RTC R)‘‘,
  PROVE_TAC [diamond_def,diamond_RTC_lemma]);
```

8.5 Back to combinators

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

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

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

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

$$\begin{tikzcd} & \mathsf{K\,S\,(K\,K\,K)} \arrow[dl] \arrow[dr] & \\ \mathsf{S} & & \mathsf{K\,S\,K} \arrow[dl] \\ & \mathsf{S} & \end{tikzcd}$$

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

This is a problem. We are going to have to take another approach. We will define another reduction strategy (*parallel reduction*), and prove that its reflexive, transitive closure is actually the same relation as our original's reflexive and transitive closure. Then we will also show that parallel reduction has the diamond property. This will establish that its reflexive, transitive closure has it too. Then, because they are the same relation, we will have that the reflexive, transitive closure of our original relation has the diamond property, and therefore, our original relation will be confluent.

8.5.1 Parallel reduction

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

```
- set_fixity "-||->" (Infix(NONASSOC, 450));                    30
> val it = () : unit
```

Then we can define parallel reduction itself. The rules look very similar to those for $\rightarrow$. The difference is that we allow the reflexive transition, and say that an application of $x\ u$ can be transformed to $y\ v$ if there are transformations taking x to y and u to v. This

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

is why we must have reflexivity incidentally. Without it, a term like $(\mathsf{K}\ x\ y)\,\mathsf{K}$ couldn't reduce because while the LHS of the application ($\mathsf{K}\ x\ y$) can reduce, its RHS (K) can't.

```
- val (predn_rules, _, predn_cases) = xHol_reln "predn"                    31
      `(!x. x -||-> x) /\
       (!x y u v. x -||-> y /\ u -||-> v
                          ==>
                  x # u -||-> y # v) /\
       (!x y. K # x # y -||-> x) /\
       (!f g x. S # f # g # x -||-> (f # x) # (g # x))`;
> val predn_rules =
    |- (∀x. x -||-> x) ∧
       (∀x y u v. x -||-> y ∧ u -||-> v ⇒ x # u -||-> y # v) ∧
       (∀x y. K # x # y -||-> x) ∧
       ∀f g x. S # f # g # x -||-> f # x # (g # x) : thm
  val predn_cases =
    |- ∀a0 a1.
         a0 -||-> a1 ⇔
         (a1 = a0) ∨
         (∃x y u v. (a0 = x # u) ∧ (a1 = y # v) ∧
                    x -||-> y ∧ u -||-> v) ∨
         (∃y. a0 = K # a1 # y) ∨
         ∃f g x. (a0 = S # f # g # x) ∧ (a1 = f # x # (g # x))
    : thm
```

8.5.2 Using RTC

Now we can set up nice syntax for the reflexive and transitive closures of our two relations. We will use ASCII symbols for both that consist of the original symbol followed by an asterisk. Note also how, in defining the two relations, we have to use the $ character to "escape" the symbols' usual fixities. This is exactly analogous to the way in which ML's op keyword is used. First, we create the desired symbol for the concrete syntax, and then we "overload" it so that the parser will expand it to the desired form.

```
- set_fixity "-->*" (Infix(NONASSOC, 450));                                32
> val it = () : unit

- overload_on ("-->*", ``RTC $-->``);
> val it = () : unit
```

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

```
- set_fixity "-||->*" (Infix(NONASSOC, 450));                              33
> val it = () : unit

- overload_on ("-||->*", ``RTC $-||->``);
> val it = () : unit
```

Incidentally, in conjunction with PROVE we can now automatically demonstrate relatively long chains of reductions:

```
- PROVE [RTC_rules, redn_rules] ``S # K # K # x -->* x``;                    34
Meson search level: ......
> val it = |- S # K # K # x -->* x : thm

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

(The latter sequence is seven reductions long.)

8.5.3 Proving the RTCs are the same

We start with the easier direction, and show that everything in $\rightarrow^*$ is in $\dashv\!|\!\!\mapsto^*$. Because RTC is monotone (which fact is left to the reader to prove), we can reduce this to showing that $x \rightarrow y \supset x \dashv\!|\!\!\mapsto y$.

Our goal:

```
- g `!x y. x -->* y ==> x -||->* y`;                                          35
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ∀x y. x -->* y ⇒ x -||->* y
```

We back-chain using our monotonicity result:

```
- e (MATCH_MP_TAC RTC_monotone);                                              36
OK..
1 subgoal:
> val it =
    ∀x y. x --> y ⇒ x -||-> y
```

Now we can induct over the rules for $\rightarrow$:

```
- e (Induct_on `$-->`);                                                       37
OK..
1 subgoal:
> val it =
    (∀x y f. x --> y ∧ x -||-> y ⇒ f # x -||-> f # y) ∧
    (∀f g x. f --> g ∧ f -||-> g ⇒ f # x -||-> g # x) ∧
    (∀x y. K # x # y -||-> x) ∧
    ∀f g x. S # f # g # x -||-> f # x # (g # x)
```

We could split the 4-way conjunction apart into four goals, but there is no real need. It is quite clear that each follows immediately from the rules for parallel reduction.

```
- e (PROVE_TAC [predn_rules]);                                    38
OK..
Meson search level: ...........

Goal proved. [...]
> val it =
    Initial goal proved.
    |- ∀x y. x -->* y ⇒ x -||->* y : goalstack
```

Packaged into a tidy little sub-goal-package-free parcel, our proof is

```
val RTCredn_RTCpredn = store_thm(                                  39
  "RTCredn_RTCpredn",
  ``!x y. x -->* y   ==>   x -||->* y``,
  MATCH_MP_TAC RTC_monotone THEN
  Induct_on `$-->` THEN PROVE_TAC [predn_rules]);
```

· · · ◇ · · ·

Our next proof is in the other direction. It should be clear that we will not just be able to appeal to the monotonicity of RTC this time; one step of the parallel reduction relation can not be mirrored with one step of the original reduction relation. It's clear that mirroring one step of the parallel reduction relation might take many steps of the original relation. Let's prove that then:

```
- g `!x y. x -||-> y   ==>   x -->* y`;                              40
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ∀x y. x -||-> y ⇒ x -->* y
```

This time our induction will be over the rules defining the parallel reduction relation.

```
- e (Induct_on `$-||->`);                                           41
OK..
1 subgoal:
> val it =
    (∀x. x -->* x) ∧
    (∀x y x' y'. x -||-> y ∧ x -->* y ∧ x' -||-> y' ∧ x' -->* y' ⇒
                 x # x' -->* y # y') ∧
    (∀x y. K # x # y -->* x) ∧
    ∀f g x. S # f # g # x -->* f # x # (g # x)
```

There are four conjuncts here, and it should be clear that all but the second can be proved immediately by appeal to the rules for the transitive closure and for → itself.

We could split apart the conjunctions and enter a THENL branch. However, we'd need to repeat the same tactic three times to quickly close three of the four branches. Instead, we use the TRY tactical to try applying the same tactic to all four branches. If our tactic fails on branch #2, as we expect, TRY will protect us against this failure and let us proceed.

```
e (REPEAT CONJ_TAC THEN                                                    42
   TRY (PROVE_TAC [RTC_rules, redn_rules]));
OK..
Meson search level: ....
Meson search level: ....
Meson search level: ............................
Meson search level: ..
1 subgoal:
> val it =
    ∀x y x' y'. x -||-> y ∧ x -->* y ∧ x' -||-> y' ∧ x' -->* y' ⇒
                x # x' -->* y # y'
```

Note that wrapping TRY around PROVE_TAC is not always wise. It can often take the PROVE_TAC tactic an extremely long time to exhaust its search space, and then give up with a failure. Here, "we got lucky".

Anyway, what of this latest sub-goal? If we look at it for long enough, we should see that it is another monotonicity fact. More accurately, we need what is called a *congruence* result for -->*. In this form, it's not quite right for easy proof. Let's go away and prove RTCredn_ap_monotonic separately. (Another exercise!) Our new theorem should state

```
val RTCredn_ap_congruence = store_thm(                                     43
  "RTCredn_ap_congruence",
  ``!x y. x -->* y ==> !z. x # z -->* y # z /\ z # x -->* z # y``,
  ...);
```

Now that we have this, our sub-goal is almost immediately provable. Using it, we know that

$x\ x' \rightarrow^* y\ x'$
$y\ x' \rightarrow^* y\ y'$

All we need to do is "stitch together" the two transitions above and go from $x\ x'$ to $y\ y'$. We can do this by appealing to our earlier RTC_RTC result.

```
e (PROVE_TAC [RTC_RTC, RTCredn_ap_congruence]);                            44
OK..
Meson search level: .......

Goal proved. [...]
> val it =
    Initial goal proved.
    |- ∀x y. x -||-> y ⇒ x -->* y : goalstack
```

But given that we can finish off what we thought was an awkward branch with just another application of PROVE_TAC, we don't need to use our fancy TRY-footwork at the stage before. Instead, we can just merge the theorem lists passed to both invocations, dispense with the REPEAT CONJ_TAC and have a very short tactic proof indeed:

```
val predn_RTCredn = store_thm(                                        45
  "predn_RTCredn",
  ``!x y. x -||-> y  ==>  x -->* y``,
  Induct_on `$-||->` THEN
  PROVE_TAC [RTC_rules, redn_rules, RTC_RTC, RTCredn_ap_congruence]);
```

· · · ◇ · · ·

Now it's time to prove that if a number of parallel reduction steps are chained together, then we can mirror this with some number of steps using the original reduction relation. Our goal:

```
- g `!x y. x -||->* y  ==> x -->* y`;                                 46
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ∀x y. x -||->* y ⇒ x -->* y
```

We use the appropriate induction principle to get to:

```
- e (Induct_on `RTC`);                                                47
OK..
1 subgoal:
> val it =
    (∀x. x -->* x) ∧
    ∀x x' y. x -||-> x' ∧ x' -||-> y* ∧ x' -->* y ⇒ x -->* z
```

This we can finish off in one step. The first conjunct is obvious, and in the second the x -||-> y and our last result combine to tell us that x -->* y. Then this can be chained together with the other assumption in the second conjunct and we're done.

```
- e (PROVE_TAC [RTC_rules, predn_RTCredn, RTC_RTC]);                  48
OK..
Meson search level: .......

Goal proved.[...]
> val it =
    Initial goal proved.
    |- ∀x y. x -||->* y ⇒ x -->* y : proof
```

Packaged up, this proof is:

49
```
val RTCpredn_RTCredn = store_thm(
  "RTCpredn_RTCredn",
  ``!x y. x -||->* y   ==>  x -->* y``,
  Induct_on `RTC` THEN PROVE_TAC [predn_RTCredn, RTC_RTC, RTC_rules]);
```

· · · ◇ · · ·

Our final act is to use what we have so far to conclude that $\rightarrow^*$ and $\dashv\!|\!\mapsto^*$ are equal. We state our goal:

50
```
- g `$-||->* = $-->*`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         $-||->* = $-->*
```

We want to now appeal to extensionality. The simplest way to do this is to rewrite with the theorem FUN_EQ_THM:

51
```
- FUN_EQ_THM;
> val it = |- ∀f g. (f = g) ⇔ ∀x. f x = g x : thm
```

So, we rewrite:

52
```
- e (SIMP_TAC std_ss [FUN_EQ_THM]);
OK..
1 subgoal:
> val it =
    ∀x x'. x -||->* x' = x -->* x'
```

This goal is an easy consequence of our two earlier implications.

53
```
- e (PROVE_TAC [RTCpredn_RTCredn, RTCredn_RTCpredn]);
OK..
Meson search level: ......

Goal proved. [...]
> val it =
    Initial goal proved.
    |- $-||->* = $-->* : goalstack
```

Packaged, the proof is:

54
```
val RTCpredn_EQ_RTCredn = store_thm(
  "RTCpredn_EQ_RTCredn",
  ``$-||->* = $-->*``,
  SIMP_TAC std_ss [FUN_EQ_THM] THEN
  PROVE_TAC [RTCpredn_RTCredn, RTCredn_RTCpredn]);
```

8.5.4 Proving a diamond property for parallel reduction

Now we just have one substantial proof to go. Before we can even begin, there are a number of minor lemmas we will need to prove first. These are basically specialisations of the theorem predn_cases. We want exhaustive characterisations of the possibilities when the following terms undergo a parallel reduction: x y, K, S, K x, S x, K x y, S x y and S x y z.

To do this, we will write a little function that derives characterisations automatically:

```
- fun characterise t = SIMP_RULE (srw_ss()) [] (SPEC t predn_cases);     55
> val characterise = fn : term -> thm
```

The characterise function specialises the theorem predn_cases with the input term, and then simplifies. The srw_ss() simpset includes information about the injectivity and disjointness of constructors and eliminates obvious impossibilities. For example,

```
- val K_predn = characterise ``K``;                                      56
<<HOL message: more than one resolution of overloading was possible>>
> val K_predn = |- ∀a1. K -||-> a1 = (a1 = K) : thm

- val S_predn = characterise ``S``;
<<HOL message: more than one resolution of overloading was possible>>
> val S_predn = |- ∀a1. S -||-> a1 = (a1 = S) : thm
```

Unfortunately, what we get back from other inputs is not so good:

```
- val Sx_predn0 = characterise ``S # x``;                                57
> val Sx_predn0 =
    |- ∀a1.
         S # x -||-> a1 =
         (a1 = S # x) ∨
         ∃y v. (a1 = y # v) ∧ S -||-> y ∧ x -||-> v : thm
```

That first disjunct is redundant, as the following demonstrates:

```
val Sx_predn = prove(                                                    58
  ``!x y. S # x -||-> y = ?z. (y = S # z) /\ (x -||-> z)``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [Sx_predn0, predn_rules, S_predn]);
```

Our characterise function will just have to help us in the proofs that follow.

```
val Kx_predn = prove(                                                    59
  ``!x y. K # x -||-> y = ?z. (y = K # z) /\ (x -||-> z)``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ``K # x``, predn_rules, K_predn]);
```

What of K x y? A little thought demonstrates that there really must be two cases this time.

```
val Kxy_predn = prove(                                                 60
  ``!x y z.
       K # x # y -||-> z =
       (?u v. (z = K # u # v) /\ (x -||-> u) /\ (y -||-> v)) \/
       (z = x)``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ``K # x # y``, predn_rules,
                 Kx_predn]);
```

By way of contrast, there is only one case for S x y because it is not yet a "redex" at the top-level.

```
val Sxy_predn = prove(                                                 61
  ``!x y z. S # x # y -||-> z =
            ?u v. (z = S # u # v) /\ (x -||-> u) /\ (y -||-> v)``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ``S # x # y``, predn_rules,
                 Sx_predn]);
```

Next, the characterisation for S x y z:

```
val Sxyz_predn = prove(                                                62
  ``!w x y z. S # w # x # y -||-> z =
              (?p q r. (z = S # p # q # r) /\
                       w -||-> p /\ x -||-> q /\ y -||-> r) \/
              (z = (w # y) # (x # y))``,
  REPEAT GEN_TAC THEN EQ_TAC THEN
  RW_TAC std_ss [characterise ``S # w # x # y``, predn_rules,
                 Sxy_predn]);
```

Last of all, we want a characterisation for x y. What `characterise` gives us this time can't be improved upon, for all that we might look upon the four disjunctions and despair.

```
- val x_ap_y_predn = characterise ``x # y``;                           63
> val x_ap_y_predn =
    |- ∀a1.
         x # y -||-> a1 =
         (a1 = x # y) ∨
         (∃y' v. (a1 = y' # v) ∧ x -||-> y' ∧ y -||-> v) ∨
         (x = K # a1) ∨
         ∃f g. (x = S # f # g) ∧ (a1 = f # y # (g # y)) : thm
```

· · · ◇ · · ·

Now we are ready to prove the final goal. It is

```
- g ‘!x y. x -||-> y ==>                                          64
          !z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u‘;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ∀x y. x -||-> y ⇒ ∀z. x -||-> z ⇒ ∃u. y -||-> u ∧ z -||-> u
```

We now induct and split the goal into its individual conjuncts:

```
- e (Induct_on ‘$-||->‘ THEN REPEAT CONJ_TAC);                    65
OK..
4 subgoals:
> val it =
    ∀f g x z. S # f # g # x -||-> z ⇒
              ∃u. f # x # (g # x) -||-> u ∧ z -||-> u


    ∀x y z. K # x # y -||-> z ⇒ ∃u. x -||-> u ∧ z -||-> u


    ∀x y u v.
      x -||-> y ∧
      (∀z. x -||-> z ⇒ ∃u. y -||-> u ∧ z -||-> u) ∧
      u -||-> v ∧
      (∀z. u -||-> z ⇒ ∃u. v -||-> u ∧ z -||-> u) ⇒
      ∀z. x # u -||-> z ⇒ ∃u. y # v -||-> u ∧ z -||-> u


    ∀x z. x -||-> z ⇒ ∃u. x -||-> u ∧ z -||-> u
```

The first goal is easily disposed of. The witness we would provide for this case is simply z, but PROVE_TAC will do the work for us:

```
- e (PROVE_TAC [predn_rules]);                                    66
OK..
Meson search level: ...

Goal proved. [...]
```

The next goal includes two instances of terms of the form `x # y -||-> z`. We can use our `x_ap_y_predn` theorem here. However, if we rewrite indiscriminately with it, we will really confuse the goal. We want to rewrite just the assumption, not the instance underneath the existential quantifier. Starting everything by repeatedly stripping can't lead us too far astray.

```
- e (REPEAT STRIP_TAC);                                              67
OK..
1 subgoal:
> val it =
    ∃u. y # v -||-> u ∧ z -||-> u
    ------------------------------------
      0.  x -||-> y
      1.  ∀z. x -||-> z ⇒ ∃u. y -||-> u ∧ z -||-> u
      2.  u -||-> v
      3.  ∀z. u -||-> z ⇒ ∃u. v -||-> u ∧ z -||-> u
      4.  x # u -||-> z
```

We need to split up assumption 4. We can get it out of the assumption list using the `Q.PAT_ASSUM` theorem-tactical. We will write

```
    Q.PAT_ASSUM ‘x # y -||-> z‘
      (STRIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn])
```

The quotation specifies the pattern that we want to match. The second argument specifies how we are going to transform the theorem. Reading the compositions from right to left, first we will simplify with the `x_ap_y_predn` theorem and then we will assume the result back into the assumptions, stripping disjunctions and existentials as we go.[4]

We already know that doing this is going to produce four new sub-goals (there were four disjuncts in the `x_ap_y_predn` theorem). At least one of these should be trivial because it will correspond to the case when the parallel reduction is just a "do nothing" step. Let's try eliminating the simple cases with a "speculative" call to `PROVE_TAC` wrapped inside a `TRY`. And before doing that, we should do some rewriting to make sure that equalities in the assumptions are eliminated.

So:

[4]An alternative to using `PAT_ASSUM` is to use `by` instead: you would have to state the four-way disjunction yourself, but the proof would be more "declarative" in style, and though wordier, might be more maintainable.

68

```
- e (Q.PAT_ASSUM ‘x # y -||-> z‘
       (STRIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn]) THEN
     RW_TAC std_ss [] THEN
     TRY (PROVE_TAC [predn_rules]));
OK..
Meson search level: ..............................
Meson search level: ..............................
Meson search level: .................
Meson search level: .....
2 subgoals:
> val it =
    ∃u’. y # v -||-> u’ ∧ f # u # (g # u) -||-> u’
    ------------------------------------
      0.  S # f # g -||-> y
      1.  ∀z. S # f # g -||-> z ⇒ ∃u. y -||-> u ∧ z -||-> u
      2.  u -||-> v
      3.  ∀z. u -||-> z ⇒ ∃u. v -||-> u ∧ z -||-> u

    ∃u. y # v -||-> u ∧ z -||-> u
    ------------------------------------
      0.  K # z -||-> y
      1.  ∀z’. K # z -||-> z’ ⇒ ∃u. y -||-> u ∧ z’ -||-> u
      2.  u -||-> v
      3.  ∀z. u -||-> z ⇒ ∃u. v -||-> u ∧ z -||-> u
```

Brilliant! We’ve eliminated two of the four disjuncts already. Now our next goal features a term `K # z -||-> y` in the assumptions. We have a theorem that pertains to just this situation. But before applying it willy-nilly, let us try to figure out exactly what the situation is. A diagram of the current situation might look like

$$\begin{tikzcd}
\boxed{K \# z} \# \boxed{u} \arrow[r] \arrow[d] & z \arrow[d, dotted] \\
\boxed{y} \# \boxed{v} \arrow[r, dotted] & ?u?
\end{tikzcd}$$

Our theorem tells us that `y` must actually be of the form `K # w` for some `w`, and that there must be an arrow between `z` and `w`. Thus:

```
- e (‘?w. (y = K # w) /\ (z -||-> w)‘ by PROVE_TAC [Kx_predn]);    69
OK..
Meson search level: ......
1 subgoal:
> val it =
    ∃u. y # v -||-> u ∧ z -||-> u
    ------------------------------------
      0.  K # z -||-> y
      1.  ∀z’. K # z -||-> z’ ⇒ ∃u. y -||-> u ∧ z’ -||-> u
      2.  u -||-> v
      3.  ∀z. u -||-> z ⇒ ∃u. v -||-> u ∧ z -||-> u
      4.  y = K # w
      5.  z -||-> w
```

On inspection, it becomes clear that the `u` must be `w`. The first conjunct requires `K # w # v -||-> w`, which we have because this is what Ks do, and the second conjunct is already in the assumption list. Rewriting (eliminating that equality in the assumption list first will make `PROVE_TAC`’s job that much easier), and then first order reasoning will solve this goal:

```
- e (RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]);                70
OK..
Meson search level: ...

Goal proved. [...]
Remaining subgoals:
> val it =
    ∃u’. y # v -||-> u’ ∧ f # u # (g # u) -||-> u’
    ------------------------------------
      0.  S # f # g -||-> y
      1.  ∀z. S # f # g -||-> z ⇒ ∃u. y -||-> u ∧ z -||-> u
      2.  u -||-> v
      3.  ∀z. u -||-> z ⇒ ∃u. v -||-> u ∧ z -||-> u
```

This case involving S is analogous. Here’s the tactic to apply:

```
- e (‘?p q. (y = S # p # q) /\ (f -||-> p) /\ (g -||-> q)‘         71
        by PROVE_TAC [Sxy_predn] THEN
     RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]);
OK..
Meson search level: ........
Meson search level: ..........

Goal proved.[...]
Remaining subgoals:
> val it =
    ∀f g x z. S # f # g # x -||-> z ⇒
              ∃u. f # x # (g # x) -||-> u ∧ z -||-> u


    ∀x y z. K # x # y -||-> z ⇒ ∃u. x -||-> u ∧ z -||-> u
```

This next goal features a `K # x # y -||-> z` term that we have a theorem for already. And again, let's speculatively use a call to `PROVE_TAC` to eliminate the simple cases immediately (`Kxy_predn` is a disjunct so we'll get two sub-goals if we don't eliminate anything).

```
- e (RW_TAC std_ss [Kxy_predn] THEN                          72
     TRY (PROVE_TAC [predn_rules]));
OK..
Meson search level: ..
Meson search level: ...

Goal proved. [...]
Remaining subgoals:
> val it =
    ∀f g x z. S # f # g # x -||-> z ⇒
              ∃u. f # x # (g # x) -||-> u ∧ z -||-> u
```

Better yet! We got both cases immediately, and have moved onto the last case. We can try the same strategy.

```
- e (RW_TAC std_ss [Sxyz_predn] THEN PROVE_TAC [predn_rules]);   73
OK..
Meson search level: ..
Meson search level: ..........

Goal proved.[...]
> val it =
    Initial goal proved.
    |- ∀x y. x -||-> y ⇒ ∀z. x -||-> z ⇒
             ∃u. y -||-> u ∧ z -||-> u : goalstack
```

The final goal proof can be packaged into:

```
val predn_diamond_lemma = prove(                                          74
  ``!x y. x -||-> y ==>
        !z. x -||-> z ==> ?u. y -||-> u /\ z -||-> u``,
  Induct_on `$-||->` THEN REPEAT CONJ_TAC THENL [
    PROVE_TAC [predn_rules],
    REPEAT STRIP_TAC THEN
    Q.PAT_ASSUM `x # y -||-> z`
      (STRIP_ASSUME_TAC o SIMP_RULE std_ss [x_ap_y_predn]) THEN
    RW_TAC std_ss [] THEN
    TRY (PROVE_TAC [predn_rules]) THENL [
      `?w. (y = K # w) /\ (z -||-> w)` by PROVE_TAC [Kx_predn] THEN
      RW_TAC std_ss [] THEN PROVE_TAC [predn_rules],
      `?p q. (y = S # p # q) /\ (f -||-> p) /\ (g -||-> q)` by
         PROVE_TAC [Sxy_predn] THEN
      RW_TAC std_ss [] THEN PROVE_TAC [predn_rules]
    ],
    RW_TAC std_ss [Kxy_predn] THEN PROVE_TAC [predn_rules],
    RW_TAC std_ss [Sxyz_predn] THEN PROVE_TAC [predn_rules]
  ]);
```

· · · ◇ · · ·

We are on the home straight. The lemma can be turned into a statement involving the diamond constant directly:

```
val predn_diamond = store_thm(                                            75
  "predn_diamond",
  ``diamond $-||->``,
  PROVE_TAC [diamond_def, predn_diamond_lemma]);
```

And now we can prove that our original relation is confluent in similar fashion:

```
val confluent_redn = store_thm(                                           76
  "confluent_redn",
  ``confluent $-->``,
  PROVE_TAC [predn_diamond, confluent_diamond_RTC,
             RTCpredn_EQ_RTCredn, diamond_RTC]);
```

8.6 Exercises

If necessary, answers to the first three exercises can be found by examining the source file in `examples/ind_def/clScript.sml`.

1. Prove that

$$\mathsf{RTC}\ R\ x\ y\ \wedge\ \mathsf{RTC}\ R\ y\ z\ \ \supset\ \ \mathsf{RTC}\ R\ x\ z$$

You will need to prove the goal by induction, and will probably need to massage it slightly first to get it to match the appropriate induction principle. Store the theorem under the name `RTC_RTC`.

2. Another induction. Show that

$$(\forall x\, y.\ R_1\ x\ y \supset R_2\ x\ y) \supset (\forall x\, y.\ \mathsf{RTC}\ R_1\ x\ y \supset \mathsf{RTC}\ R_2\ x\ y)$$

Call the resulting theorem `RTC_monotone`.

3. Yet another RTC induction, but where R is no longer abstract, and is instead the original reduction relation. Prove

$$x \rightarrow^* y \quad \supset \quad \forall z.\ \ x\ z \rightarrow^* y\ z \wedge z\ x \rightarrow^* z\ y$$

Call it `RTCredn_ap_congruence`.

4. Come up with a counter-example for the following property:

$$\begin{array}{c} \left(\begin{array}{ll} \forall x\, y\, z. & R\ x\ y \ \ \wedge \ \ R\ x\ z \quad \supset \\ & \exists u.\ \mathsf{RTC}\ R\ y\ u \ \ \wedge \ \ \mathsf{RTC}\ R\ z\ u \end{array}\right) \\ \supset \\ \mathsf{diamond}\ (\mathsf{RTC}\ R) \end{array}$$

Chapter 9

Proof Tools: Propositional Logic

Users of HOL can create their own theorem proving tools by combining predefined rules and tactics. The ML type-discipline ensures that only logically sound methods can be used to create values of type `thm`. In this chapter, a real example is described.

Two implementations of the tool are given to illustrate various styles of proof programming. The first implementation is the obvious one, but is inefficient because of the 'brute force' method used. The second implementation attempts to be a great deal more intelligent. Extensions to the tools to allow more general applicability are also discussed.

The problem to be solved is that of deciding the truth of a closed formula of propositional logic. Such a formula has the general form

$$\begin{array}{rcl} \varphi & ::= & v \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \varphi \Rightarrow \varphi \mid \varphi = \varphi \\ \mathit{formula} & ::= & \forall \vec{v}.\ \varphi \end{array}$$

where the variables v are all of boolean type, and where the universal quantification at the outermost level captures all of the free variables.

9.1 Method 1: Truth Tables

The first method to be implemented is the brute force method of trying all possible boolean combinations. This approach's only real virtue is that it is exceptionally easy to implement. First we will prove the motivating theorem:

```
val FORALL_BOOL = prove(
  ``(!v. P v) = P T /\ P F``,
  SRW_TAC [][EQ_IMP_THM] THEN Cases_on `v` THEN SRW_TAC [][]);
```

The proof proceeds by splitting the goal into two halves, showing

$$(\forall v.\ P(v)) \Rightarrow P(\top) \wedge P(\bot)$$

(which goal is automatically shown by the simplifier), and

$$P(\top) \wedge P(\bot) \Rightarrow P(v)$$

for an arbitrary boolean variable v. After case-splitting on v, the assumptions are then enough to show the goal. (This theorem is actually already proved in the theory `bool`.)

The next, and final, step is to rewrite with this theorem:

```
val tautDP = SIMP_CONV bool_ss [FORALL_BOOL]
```

This enables the following

```
- tautDP ``!p q. p /\ q /\ ~p``;                                        1
> val it = |- (!p q. p /\ q /\ ~p) = F : thm

- tautDP ``!p. p \/ ~p``
> val it = |- (!p. p \/ ~p) = T : thm
```

and even the marginally more intimidating

```
- time tautDP                                                           2
     ``!p q c a. ~(((~a \/ p /\ ~q \/ ~p /\ q) /\
                    (~(p /\ ~q \/ ~p /\ q) \/ a)) /\
                   (~c \/ p /\ q) /\ (~(p /\ q) \/ c)) \/
                 ~(p /\ q) \/ c /\ ~a``;
runtime: 0.147s,    gctime: 0.012s,     systime: 0.000s.
> val it =
    |- (!p q c a.
          ~(((~a \/ p /\ ~q \/ ~p /\ q) /\ (~(p /\ ~q \/ ~p /\ q) \/ a)) /\
            (~c \/ p /\ q) /\ (~(p /\ q) \/ c)) \/ ~(p /\ q) \/ c /\ ~a) =
       T : thm
```

This is a dreadful algorithm for solving this problem. The system's built-in function, `tautLib.TAUT_CONV`, solves the problem above much faster. The only real merit in this solution is that it took one line to write. This is a general illustration of the truth that HOL's high-level tools, particularly the simplifier, can provide fast prototypes for a variety of proof tasks.

9.2 Method 2: the DPLL Algorithm

The Davis-Putnam-Loveland-Logemann method [4] for deciding the satisfiability of propositional formulas in CNF (Conjunctive Normal Form) is a powerful technique, still used in state-of-the-art solvers today. If we strip the universal quantifiers from our input formulas, our task can be seen as determining the validity of a propositional formula. Testing the negation of such a formula for satisfiability is a test for validity: if the formula's negation is satisfiable, then it is not valid (the satisfying assignment will make the original false); if the formula's negation is unsatisfiable, then the formula is valid (no assignment can make it false).

(The source code for this example is available in the file `examples/dpll.sml`.)

Preliminaries

To begin, assume that we have code already to convert arbitrary formulas into CNF, and to then decide the satisfiability of these formulas. Assume further that if the input to the latter procedure is unsatisfiable, then it will return with a theorem of the form

$$\vdash \varphi = \texttt{F}$$

or if it is satisfiable, then it will return a satisfying assignment, a map from variables to booleans. This map will be a function from HOL variables to one of the HOL terms `T` or `F`. Thus, we will assume

```
datatype result = Unsat of thm | Sat of term -> term
val toCNF : term -> thm
val DPLL : term -> result
```

(The theorem returned by `toCNF` will equate the input term to another in CNF.)
Before looking into implementing these functions, we will need to consider

- how to transform our inputs to suit the function; and
- how to use the outputs from the functions to produce our desired results

We are assuming our input is a universally quantified formula. Both the CNF and DPLL procedures expect formulas without quantifiers. We also want to pass these procedures the negation of the original formula. Both of the required term manipulations required can be done by functions found in the structure `boolSyntax`. (In general, important theories (such as `bool`) are accompanied by `Syntax` modules containing functions for manipulating the term-forms associated with that theory.)

In this case we need the functions

```
strip_forall : term -> term list * term
mk_neg       : term -> term
```

The function `strip_forall` strips a term of all its outermost universal quantifications, returning the list of variables stripped and the body of the quantification. The function `mk_neg` takes a term of type `bool` and returns the term corresponding to its negation.

Using these functions, it is easy to see how we will be able to take $\forall \vec{v}.\ \varphi$ as input, and pass the term $\neg\varphi$ to the function `toCNF`. A more significant question is how to use the results of these calls. The call to `toCNF` will return a theorem

$$\vdash \neg\varphi = \varphi'$$

The formula φ' is what will then be passed to `DPLL`. (We can extract it by using the `concl` and `rhs` functions.) If `DPLL` returns the theorem $\vdash \varphi' = \texttt{F}$, an application of `TRANS` to this and the theorem displayed above will derive the formula $\vdash \neg\varphi = F$. In order to derive the final result, we will need to turn this into $\vdash \varphi$. This is best done by proving a bespoke theorem embodying the equality (there isn't one such already in the system):

```
val NEG_EQ_F = prove(``(~p = F) = p``, REWRITE_TAC []);
```

To turn $\vdash \varphi$ into $\vdash (\forall \vec{v}.\ \varphi) = \mathtt{T}$, we will perform the following proof:

$$\dfrac{\dfrac{\vdash \varphi}{\vdash \forall \vec{v}.\ \varphi}\ \texttt{GENL}(\vec{v})}{\vdash (\forall \vec{v}.\ \varphi) = \mathtt{T}}\ \texttt{EQT\_INTRO}$$

The other possibility is that DPLL will return a satisfying assignment demonstrating that φ' is satisfiable. If this is the case, we want to show that $\forall \vec{v}.\ \varphi$ is false. We can do this by assuming this formula, and then specialising the universally quantified variables in line with the provided map. In this way, it will be possible to produce the theorem

$$\forall \vec{v}.\ \varphi \vdash \varphi[\vec{v} := \textit{satisfying assignment}]$$

Because there are no free variables in $\forall \vec{v}.\ \varphi$, the substitution will produce a completely ground boolean formula. This will straightforwardly rewrite to F (if the assignment makes $\neg\varphi$ true, it must make φ false). Turning $\phi \vdash \mathtt{F}$ into $\vdash \phi = \mathtt{F}$ is a matter of calling DISCH and then rewriting with the built-in theorem IMP_F_EQ_F:

$$\vdash \forall t.\ t \Rightarrow \mathtt{F} = (t = \mathtt{F})$$

Putting all of the above together, we can write our wrapper function, which we will call DPLL_UNIV, with the UNIV suffix reminding us that the input must be universally quantified.

```
fun DPLL_UNIV t = let
  val (vs, phi) = strip_forall t
  val cnf_eqn = toCNF (mk_neg phi)
  val phi' = rhs (concl cnf_eqn)
in
  case DPLL phi' of
    Unsat phi'_eq_F => let
      val negphi_eq_F = TRANS cnf_eqn phi'_eq_F
      val phi_thm = CONV_RULE (REWR_CONV NEG_EQ_F) negphi_eq_F
    in
      EQT_INTRO (GENL vs phi_thm)
    end
  | Sat f => let
      val t_assumed = ASSUME t
      fun spec th =
          spec (SPEC (f (#1 (dest_forall (concl th)))) th)
          handle HOL_ERR _ => REWRITE_RULE [] th
    in
      CONV_RULE (REWR_CONV IMP_F_EQ_F) (DISCH t (spec t_assumed))
    end
end
```

The auxiliary function `spec` that is used in the second case relies on the fact that `dest_forall` will raise a `HOL_ERR` exception if the term it is applied to is not universally quantified. When `spec`'s argument is not universally quantified, this means that the recursion has bottomed out, and all of the original formula's universal variables have been specialised. Then the resulting formula can be rewritten to false (`REWRITE_RULE`'s built-in rewrites will handle all of the necessary cases).

The `DPLL_UNIV` function also uses `REWR_CONV` in two places. The `REWR_CONV` function applies a single (first-order) rewrite at the top of a term. These uses of `REWR_CONV` are done within calls to the `CONV_RULE` function. This lifts a conversion c (a function taking a term t and producing a theorem $\vdash t = t'$), so that `CONV_RULE` c takes the theorem $\vdash t$ to $\vdash t'$.

9.2.1 Conversion to Conjunctive Normal Form

A formula in Conjunctive Normal Form is a conjunction of disjunctions of literals (either variables, or negated variables). It is possible to convert formulas of the form we are expecting into CNF by simply rewriting with the following theorems

$$\begin{array}{rcl}
\neg(\phi \wedge \psi) & = & \neg\phi \vee \neg\psi \\
\neg(\phi \vee \psi) & = & \neg\phi \wedge \neg\psi \\
\phi \vee (\psi \wedge \xi) & = & (\phi \vee \psi) \wedge (\phi \vee \xi) \\
(\psi \wedge \xi) \vee \phi & = & (\phi \vee \psi) \wedge (\phi \vee \xi) \\
\\
\phi \Rightarrow \psi & = & \neg\phi \vee \psi \\
(\phi = \psi) & = & (\phi \Rightarrow \psi) \wedge (\psi \Rightarrow \phi)
\end{array}$$

Unfortunately, using these theorems as rewrites can result in an exponential increase in the size of a formula. (Consider using them to convert an input in Disjunctive Normal Form, a disjunction of conjunctions of literals, into CNF.)

A better approach is to convert to what is known as "definitional CNF". HOL includes functions to do this in the structure `defCNF`. Unfortunately, this approach adds extra, existential, quantifiers to the formula. For example

```
- defCNF.DEF_CNF_CONV ``p \/ (q /\ r)``;                              3
> val it =
    |- p \/ q /\ r =
       ?x. (x \/ ~q \/ ~r) /\ (r \/ ~x) /\ (q \/ ~x) /\ (p \/ x) : thm
```

Under the existentially-bound `x`, the code has produced a formula in CNF. With an example this small, the formula is actually bigger than that produced by the naïve translation, but with more realistic examples, the difference quickly becomes significant. The last example used with `tautDP` is 20 times bigger when translated naïvely than when using `defCNF`, and the translation takes 150 times longer to perform.

But what of these extra existentially quantified variables? In fact, we can ignore the quantification when calling the core DPLL procedure. If we pass the unquantified body to DPLL, we will either get back an unsatisfiable verdict of the form $\vdash \varphi' = \mathtt{F}$, or a satisfying assignment for all of the free variables. If the latter occurs, the same satisfying assignment will also satisfy the original. If the former, we will perform the following proof

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdash \varphi' = \mathtt{F}}{\vdash \varphi' \Rightarrow \mathtt{F}}}{\vdash \forall \vec{x}.\ \varphi' \Rightarrow \mathtt{F}}}{\vdash (\exists \vec{x}.\ \varphi') \Rightarrow \mathtt{F}}}{\vdash (\exists \vec{x}.\ \varphi') = \mathtt{F}}$$

producing a theorem of the form expected by our `wrapper` function.

In fact, there is an alternative function in the `defCNF` API that we will use in preference to `DEF_CNF_CONV`. The problem with `DEF_CNF_CONV` is that it can produce a big quantification, involving lots of variables. We will rather use `DEF_CNF_VECTOR_CONV`. Instead of output of the form

$$\vdash \varphi = (\exists \vec{x}.\ \varphi')$$

this second function produces

$$\vdash \varphi = (\exists (v : \mathsf{num} \to \mathsf{bool}).\ \varphi')$$

where the individual variables x_i of the first formula are replaced by calls to the v function $v(i)$, and there is just one quantified variable, v. This variation will not affect the operation of the proof sketched above. And as long as we don't require literals to be variables or their negations, but also allow them to be terms of the form $v(i)$ and $\neg v(i)$ as well, then the action of the DPLL procedure on the formula φ' won't be affected either.

Unfortunately for uniformity, in simple cases, the definitional CNF conversion functions may not result in any existential quantifications at all. This makes our implementation of `DPLL` somewhat more complicated. We calculate a `body` variable that will be passed onto the `CoreDPLL` function, as well as a `transform` function that will transform an unsatisfiability result into something of the desired form. If the result of conversion to CNF produces an existential quantification, we use the proof sketched above. Otherwise, the transformation can be the identity function, `I`:

```
fun DPLL t = let
  val (transform, body) = let
    val (vector, body) = dest_exists t
    fun transform body_eq_F = let
      val body_imp_F = CONV_RULE (REWR_CONV (GSYM IMP_F_EQ_F)) body_eq_F
      val fa_body_imp_F = GEN vector body_imp_F
      val ex_body_imp_F = CONV_RULE FORALL_IMP_CONV fa_body_imp_F
    in
      CONV_RULE (REWR_CONV IMP_F_EQ_F) ex_body_imp_F
    end
  in
    (transform, body)
  end handle HOL_ERR _ => (I, t)
in
  case CoreDPLL body of
    Unsat body_eq_F => Unsat (transform body_eq_F)
  | x => x
end
```

where we have still to implement the core DPLL procedure (called CoreDPLL above). The above code uses REWR_CONV with the IMP_F_EQ_F theorem to affect two of the proof's transformations. The GSYM function is used to flip the orientation a theorem's top-level equalities. Finally, the FORALL_IMP_CONV conversion takes a term of the form

$$\forall x.\ P(x) \Rightarrow Q$$

and returns the theorem

$$\vdash (\forall x.\ P(x) \Rightarrow Q) = ((\exists x.\ P(x)) \Rightarrow Q)$$

9.2.2 The Core DPLL Procedure

The DPLL procedure can be seen as a slight variation on the basic "truth table" technique we have already seen. As with that procedure, the core operation is a case-split on a boolean variable. There are two significant differences though: DPLL can be seen as a search for a satisfying assignment, so that if picking a variable to have a particular value results in a satisfying assignment, we do not need to also check what happens if the same variable is given the opposite truth-value. Secondly, DPLL takes some care to pick good variables to split on. In particular, *unit propagation* is used to eliminate variables that will not cause branching in the search-space.

Our implementation of the core DPLL procedure is a function that takes a term and returns a value of type result: either a theorem equating the original term to false, or a satisfying assignment (in the form of a function from terms to terms). As the DPLL search for a satisfying assignment proceeds, an assignment is incrementally constructed. This suggests that the recursive core of our function will need to take a term (the current formula) and a context (the current assignment) as parameters. The assignment can be naturally represented as a set of equations, where each equation is either $v = \texttt{T}$ or $v = \texttt{F}$.

This suggests that a natural representation for our program state is a theorem: the hypotheses will represent the assignment, and the conclusion can be the current formula. Of course, HOL theorems can't just be wished into existence. In this case, we can make everything sound by also assuming the initial formula. Thus, when we begin our initial state will be $\phi \vdash \phi$. After splitting on variable v, we will generate two new states $\phi, (v{=}\texttt{T}) \vdash \phi_1$, and $\phi, (v{=}\texttt{F}) \vdash \phi_2$, where the ϕ_i are the result of simplifying ϕ under the additional assumption constraining v.

The easiest way to add an assumption to a theorem is to use the rule ADD_ASSUM. But in this situation, we also want to simplify the conclusion of the theorem with the same assumption. This means that it will be enough to rewrite with the theorem $\psi \vdash \psi$, where ψ is the new assumption. The action of rewriting with such a theorem will cause the new assumption to appear among the assumptions of the result.

The `casesplit` function is thus:

```
fun casesplit v th = let
  val eqT = ASSUME (mk_eq(v, boolSyntax.T))
  val eqF = ASSUME (mk_eq(v, boolSyntax.F))
in
  (REWRITE_RULE [eqT] th, REWRITE_RULE [eqF] th)
end
```

A case-split can result in a formula that has been rewritten all the way to true or false. These are the recursion's base cases. If the formula has been rewritten to true, then we have found a satisfying assignment, one that is now stored for us in the hypotheses of the theorem itself. The following function, `mk_satmap`, extracts those hypotheses into a finite-map, and then returns the lookup function for that finite-map:

```
fun mk_satmap th = let
  val hyps = hypset th
  fun foldthis (t,acc) = let
    val (l,r) = dest_eq t
  in
    Binarymap.insert(acc,l,r)
  end handle HOL_ERR _ => acc
  val fmap = HOLset.foldl foldthis (Binarymap.mkDict Term.compare) hyps
in
  Sat (fn v => Binarymap.find(fmap,v)
               handle Binarymap.NotFound => boolSyntax.T)
end
```

The `foldthis` function above adds the equations that are stored as hypotheses into the finite-map. The exception handler in `foldthis` is necessary because one of the hypotheses will be the original formula. The exception handler in the function that looks up variable bindings is necessary because a formula may be reduced to true without every variable being assigned a value at all. In this case, it is irrelevant what value we give to the variable, so we arbitrarily map such variables to `T`.

If the formula has been rewritten to false, then we can just return this theorem directly. Such a theorem is not quite in the right form for the external caller, which is expecting an equation, so if the final result is of the form $\phi \vdash \mathtt{F}$, we will have to transform this to $\vdash \phi = \mathtt{F}$.

The next question to address is what to do with the results of recursive calls. If a case-split returns a satisfying assignment this can be returned unchanged. But if a recursive call returns a theorem equating the input to false, more needs to be done. If this is the first call, then the other branch needs to be checked. If this also returns that the theorem is unsatisfiable, we will have two theorems:

$$\phi_0, \Delta, (v = \mathtt{T}) \vdash \mathtt{F} \qquad \phi_0, \Delta, (v = \mathtt{F}) \vdash \mathtt{F}$$

where ϕ_0 is the original formula, Δ is the rest of the current assignment, and v is the variable on which a split has just been performed. To turn these two theorems into the desired

$$\phi_0, \Delta \vdash \mathtt{F}$$

we will use the rule of inference `DISJ_CASES`:

$$\frac{\Gamma \vdash \psi \vee \xi \quad \Delta_1 \cup \{\psi\} \vdash \phi \quad \Delta_2 \cup \{\xi\} \vdash \phi}{\Gamma \cup \Delta_1 \cup \Delta_2 \vdash \phi}$$

and the theorem `BOOL_CASES_AX`:

$$\vdash \forall t.\ (t = \mathtt{T}) \vee (t = \mathtt{F})$$

We can put these fragments together and write the top-level `CoreDPLL` function, in Figure 9.1.

All that remains to be done is to figure out which variable to case-split on. The most important variables to split on are those that appear in what are called "unit clauses", a clause containing just one literal. If there is a unit clause in a formula then it is of the form

$$\phi \wedge v \wedge \phi'$$

or

$$\phi \wedge \neg v \wedge \phi'$$

In either situation, splitting on v will always result in a branch that evaluates directly to false. We thus eliminate a variable without increasing the size of the problem. The

```
fun CoreDPLL form = let
  val initial_th = ASSUME form
  fun recurse th = let
    val c = concl th
  in
    if c = boolSyntax.T then
      mk_satmap th
    else if c = boolSyntax.F then
      Unsat th
    else let
        val v = find_splitting_var c
        val (l,r) = casesplit v th
      in
        case recurse l of
          Unsat l_false => let
          in
            case recurse r of
              Unsat r_false =>
                Unsat (DISJ_CASES (SPEC v BOOL_CASES_AX) l_false r_false)
            | x => x
          end
        | x => x
      end
  end
in
  case (recurse initial_th) of
    Unsat th => Unsat (CONV_RULE (REWR_CONV IMP_F_EQ_F) (DISCH form th))
  | x => x
end
```

Figure 9.1: The core of the DPLL function

process of eliminating unit clauses is usually called "unit propagation". Unit propagation is not usually thought of as a case-splitting operation, but doing it this way makes our code simpler.

If a formula does not include a unit clause, then choice of the next variable to split on is much more of a black art. Here we will implement a very simple choice: to split on the variable that occurs most often. Our function `find_splitting_var` takes a formula and returns the variable to split on.

```
fun find_splitting_var phi = let
  fun recurse acc [] = getBiggest acc
    | recurse acc (c::cs) = let
        val ds = strip_disj c
      in
        case ds of
          [lit] => (dest_neg lit handle HOL_ERR _ => lit)
        | _ => recurse (count_vars ds acc) cs
      end
in
  recurse (Binarymap.mkDict Term.compare) (strip_conj phi)
end
```

This function works by handing a list of clauses to the inner `recurse` function. This strips each clause apart in turn. If a clause has only one disjunct it is a unit-clause and the variable can be returned directly. Otherwise, the variables in the clause are counted and added to the accumulating map by `count_vars`, and the recursion can continue.

The `count_vars` function has the following implementation:

```
fun count_vars ds acc =
  case ds of
    [] => acc
  | lit::lits => let
      val v = dest_neg lit handle HOL_ERR _ => lit
    in
      case Binarymap.peek (acc, v) of
        NONE => count_vars lits (Binarymap.insert(acc,v,1))
      | SOME n => count_vars lits (Binarymap.insert(acc,v,n + 1))
    end
```

The use of a binary tree to store variable data makes it efficient to update the data as it is being collected. Extracting the variable with the largest count is then a linear scan of the tree, which we can do with the `foldl` function:

```
fun getBiggest acc =
  #1 (Binarymap.foldl(fn (v,cnt,a as (bestv,bestcnt)) =>
                         if cnt > bestcnt then (v,cnt) else a)
                     (boolSyntax.T, 0)
                     acc
```

9.2.3 Performance

Once inputs get even a little beyond the clearly trivial, the function we have written (at the top-level, `DPLL_UNIV`) performs considerably better than the truth table implementation. For example, the generalisation of the following term, with 29 variables, takes `wrapper` two and a half minutes to demonstrate as a tautology:

```
(s0_0 = (x_0 = ~y_0)) /\ (c0_1 = x_0 /\ y_0) /\
(s0_1 = ((x_1 = ~y_1) = ~c0_1)) /\
(c0_2 = x_1 /\ y_1 \/ (x_1 \/ y_1) /\ c0_1) /\
(s0_2 = ((x_2 = ~y_2) = ~c0_2)) /\
(c0_3 = x_2 /\ y_2 \/ (x_2 \/ y_2) /\ c0_2) /\
(s1_0 = ~(x_0 = ~y_0)) /\ (c1_1 = x_0 /\ y_0 \/ x_0 \/ y_0) /\
(s1_1 = ((x_1 = ~y_1) = ~c1_1)) /\
(c1_2 = x_1 /\ y_1 \/ (x_1 \/ y_1) /\ c1_1) /\
(s1_2 = ((x_2 = ~y_2) = ~c1_2)) /\
(c1_3 = x_2 /\ y_2 \/ (x_2 \/ y_2) /\ c1_2) /\
(c_3 = ~c_0 /\ c0_3 \/ c_0 /\ c1_3) /\
(s_0 = ~c_0 /\ s0_0 \/ c_0 /\ s1_0) /\
(s_1 = ~c_0 /\ s0_1 \/ c_0 /\ s1_1) /\
(s_2 = ~c_0 /\ s0_2 \/ c_0 /\ s1_2) /\ ~c_0 /\
(s2_0 = (x_0 = ~y_0)) /\ (c2_1 = x_0 /\ y_0) /\
(s2_1 = ((x_1 = ~y_1) = ~c2_1)) /\
(c2_2 = x_1 /\ y_1 \/ (x_1 \/ y_1) /\ c2_1) /\
(s2_2 = ((x_2 = ~y_2) = ~c2_2)) /\
(c2_3 = x_2 /\ y_2 \/ (x_2 \/ y_2) /\ c2_2) ==>
(c_3 = c2_3) /\ (s_0 = s2_0) /\ (s_1 = s2_1) /\ (s_2 = s2_2)
```

(if you want real speed, the built-in function `tautLib.TAUT_PROVE` does the above in less than a second, by using an external tool to generate the proof of unsatisfiability, and then translating that proof back into HOL).

9.3 Extending our Procedure's Applicability

The function `DPLL_UNIV` requires its input to be universally quantified, with all free variables bound, and for each literal to be a variable or the negation of a variable. This makes `DPLL_UNIV` a little unfriendly when it comes to using it as part of the proof of a goal. In this section, we will write one further "wrapper" layer to wrap around `DPLL_UNIV`, producing a tool that can be applied to many more goals.

Relaxing the Quantification Requirement The first step is to allow formulas that are not closed. In order to hand on a formula that *is* closed to `DPLL_UNIV`, we can simply generalise over the formula's free variables. If `DPLL_UNIV` then says that the new, ground formula is true, then so too will be the original. On the other hand, if `DPLL_UNIV` says that the ground formula is false, then we can't conclude anything further and will have to raise an exception.

Code implementing this is shown below:

```
fun nonuniv_wrap t = let
  val fvs = free_vars t
  val gen_t = list_mk_forall(fvs, t)
  val gen_t_eq = DPLL_UNIV gen_t
in
  if rhs (concl gen_t_eq) = boolSyntax.T then let
      val gen_th = EQT_ELIM gen_t_eq
    in
      EQT_INTRO (SPECL fvs gen_th)
    end
  else
    raise mk_HOL_ERR "dpll" "nonuniv_wrap" "No conclusion"
end
```

Allowing Non-Literal Leaves We can do better than `nonuniv_wrap`: rather than quantifying over just the free variables (which we have conveniently assumed will only be boolean), we can turn any leaf part of the term that is not a variable or a negated variable into a fresh variable. We first extract those boolean-valued leaves that are not the constants true or false.

```
fun var_leaves acc t = let
  val (l,r) = dest_conj t handle HOL_ERR _ =>
              dest_disj t handle HOL_ERR _ =>
              dest_imp t handle HOL_ERR _ =>
              dest_bool_eq t
in
  var_leaves (var_leaves acc l) r
end handle HOL_ERR _ =>
  if type_of t <> bool then
    raise mk_HOL_ERR "dpll" "var_leaves" "Term not boolean"
  else if t = boolSyntax.T then acc
  else if t = boolSyntax.F then acc
  else HOLset.add(acc, t)
```

Note that we haven't explicitly attempted to pull apart boolean negations (which one might do with `dest_neg`). This is because `dest_imp` also destructs terms `~p`, returning `p` and `F` as the antecedent and conclusion. We have also used a function `dest_bool_eq` designed to pull apart only those equalities which are over boolean values. Its definition is

```
fun dest_bool_eq t = let
  val (l,r) = dest_eq t
  val _ = type_of l = bool orelse
          raise mk_HOL_ERR "dpll" "dest_bool_eq" "Eq not on bools"
in
  (l,r)
end
```

Now we can finally write our final DPLL_TAUT function:

```
fun DPLL_TAUT tm =
  let val (univs,tm') = strip_forall tm
      val insts = HOLset.listItems (var_leaves empty_tmset tm')
      val vars = map (fn t => genvar bool) insts
      val theta = map2 (curry (op |->)) insts vars
      val tm'' = list_mk_forall (vars,subst theta tm')
  in
      EQT_INTRO (GENL univs
                   (SPECL insts (EQT_ELIM (DPLL_UNIV tm''))))
  end
```

Note how this code first pulls off all external universal quantifications (with `strip_forall`), and then re-generalises (with `list_mk_forall`). The calls to `GENL` and `SPECL` undo these manipulations, but at the level of theorems. This produces a theorem equating the original input to true. (If the input term is not an instance of a valid propositional formula, the call to `EQT_ELIM` will raise an exception.)

Exercises

1. Extend the procedure so that it handles conditional expressions (both arms of the terms must be of boolean type).

Chapter 10

Example: Abstract Data Types

This chapter consists of the specification of an abstract data type in the HOL-Omega logic, as a worked example. The goals of this chapter are:

(i) To present how new term and type constants are introduced into the logic,

(ii) To show how abstract data types can be created and used to hide information,

(iii) To show how an abstract algebra can be realized in the HOL-Omega logic.

The notion of abstract data types comes from the field of software engineering, when one is concerned with creating a model of a system that expresses just what is necessary, nothing more. This brevity is intended to focus on just the essential aspects of the system, without the encumberance of unnecessary detail. This is appropriate when laying down requirements for what a system should do, or when drawing up an initial design of a system. It is important to not over-specify the system, in order to allow the eventual implementors of the design to experiment with different algorithms or data structures in order to find the choices that work best. Those choices should be left until later, when the implementor has had the time to consider alternative possibilities. The initial design should decide only *what* it should do, not *how* it should do it. The specific *how* should be hidden from the rest of the system that uses this part, as an irrelevant detail, simplifying that portion's construction.

Such information hiding is useful for both data structure design and for algorithm design. In this chapter, we shall address just information hiding of data structures. We would like to specify the essential contents of some data structure while leaving all inessential details unspecified.

Then the construction of other parts of the system can begin, using the partially specified data structure, while simultaneously deeper work can commence on selecting the exact best implementation of the data structure. This implementation can even be changed completely while the rest of the system is half-done, without causing any ripple effects on the rest of the project, so long as both sides conform to the original partial specification of the data structure. This kind of modularity is absolutely essential for good software engineering and the practical maintenance of large systems.

Abstract data types are crucially important in the modeling of abstract algebras. Here we consider an algebra to be a collection of types and contants, where some of the types are only named and given kinds, but not defined. Likewise, the constants may also have types but lack definitions. Instead, along with the signature of the algebra is attached a collection of properties (boolean expressions) that relate the different constants in the algebra, and which serve as the axioms of that algebra.

Abstract algebras are very useful in the modeling of hardware and software designs, where many of the details may be yet undefined, where we desire for the moment to leave certain aspects unspecified and abstract. As an example of an abstract algebra, consider the following signature:

type	vect	:	**ty**
op	VT	:	vect
op	VF	:	vect
op	VCONCAT	:	vect → vect → vect
axiom	VCONCAT x (VCONCAT y z) = VCONCAT (VCONCAT x y) z		

The above mentions one type, vect, two constant vectors, VT and VF, and one operator for combining them, VCONCAT, with the proviso that VCONCAT is associative. This specification has many models, for example it has as a model, non-empty bit strings. But of all these models, there are some which are "best" in the sense that they are *initial*. Initiality is a concept from category theory, defined as follows.

In the category of algebras, the arrows are homomorphisms from one algebra to another. Another algebra of the same signature might look like

type	vect'	:	**ty**
op	VT'	:	vect'
op	VF'	:	vect'
op	VCONCAT'	:	vect' → vect' → vect'
axiom	VCONCAT' x (VCONCAT' y z) = VCONCAT' (VCONCAT' x y) z		

In the category of algebras of this signature, the arrow from the first algebra to this other algebra is a homomorphism, that is, a function ϕ from vect to vect' such that

$$\boxed{\begin{array}{rcl} \phi\,\text{VT} & = & \text{VT'} \\ \phi\,\text{VF} & = & \text{VF'} \\ \phi\,(\text{VCONCAT}\ x\ \text{y}) & = & \text{VCONCAT'}\ (\phi\ x)\ (\phi\ \text{y}) \end{array}}$$

In category theory, an object 0 is called *initial* if for every object A in the category, there exists exactly one arrow from 0 to A.

It is a standard result of universal algebra[1] that for each signature there is a uniquely determined algebra which is initial, and which is therefore called the *initial algebra*. The initial algebra is in some sense the "best" interpretation of the signature, as it

[1]George Gratzer. *Universal Algebra*. Van Nostrand, 1968.

incorporates the most information and detail; all other algebras of that signature are in that sense inferior.

This means that it is would be highly valuable to be able to define a new type and its constants by describing it as the initial algebra of its signature. And that is what we will do in this chapter. But before we begin, we will first lay the necessary groundwork by discussing the different foundational principles for introducing new type and term constants into the HOL-Omega logic.

10.1 New term and type constants

One of the key features of the HOL-Omega logic is that it is extensible, that is, that new constants can be created and added to the logic by the user. Both term constants and type constants may be added. This is very important, as it is the primary way that users create models of existing real-world applications within the logic: one creates new types to represent the special kinds of data being manipulated by the application, and then one creates new term constants, based on those new types, to represent the particular operations and activities performed by the application.

HOL-Omega contains some very powerful, highly automated tools for easily defining new types and new term constants, and these are the tools that are most often used to build a model inside HOL-Omega. Despite their complexity and power, all of these tools are eventually based on a small number of basic definitional principles in the core of the logic. These principles give the primitive, essential tools for extending the logic by adding new type and term names. The high-level tools just provide ways to leverage these fundamental principles in a more automated and user-friendly way; they do not add any real new power.

In the following, we will discuss these fundamental principles. The first three of these are taken almost unchanged from HOL; the fourth is unique to HOL-Omega, and is the key new feature supporting abstract data types and abstract algebras.

10.1.1 New term constant definition

New term constants may be introduced by the *new term constant definition* principle, implemented as the ML function `new_definition`:

```
new_definition : (string * term) -> thm
```

Evaluating `new_definition("`*name*`", ‘‘`$c\ x_1 \ldots x_n$ `=` t`‘‘)`

- where c is the name of the constant to be created,
- $x_1 \ldots x_n$ are zero or more distinct variables, as formal arguments to c,

- t is a term that may contain the $x_1 \ldots x_n$, but has no other free term variables, and
- all the free type and kind variables of $x_1 \ldots x_n$ and t are also free type or kind variables of the type of c,

defines c in the logic as a new constant with the value $\lambda x_1 \ldots x_n.\ t$. It also creates and returns the theorem `|-` $c\ x_1 \ldots x_n$ `=` t, and in addition saves this theorem as a definition in the current theory under the name *name*.

Note that `new_definition` cannot be used to create a recursive function; t cannot refer to the new c being created.

If the above side conditions are met, this is always a valid operation to do, as the new name c is really just an abbreviation for a term that could already have been constructed in the logic. In principle all instances of such new term constants could be replaced by the terms that they abbreviate, so it is impossible to introduce unsoundness by this definitional principle.

10.1.2 New term constant specification

New term constants may also be introduced by the *new term constant specification* principle, implemented as the ML function `new_specification`:

```
new_specification : string * string list * thm -> thm
```

Evaluating `new_specification("`*name*`", ["`c_1`",...,"`c_n`"], |- ?`$x_1 \ldots x_n$`.`t`)`

- $c_1 \ldots c_n$ are the names of the n constants to be created, all distinct,
- $x_1 \ldots x_n$ are n distinct variables,
- t is a term that may contain the $x_1 \ldots x_n$, but has no other free term variables, and
- all the free type and kind variables of $x_1 \ldots x_n$ and t are also free type or kind variables of the type of each c_i for $1 \leq i \leq n$,

defines each c_i in the logic as a new constant, for $1 \leq i \leq n$, with some fixed values such that

`|-` $t[c_1, \ldots, c_n/x_1, \ldots, x_n]$.

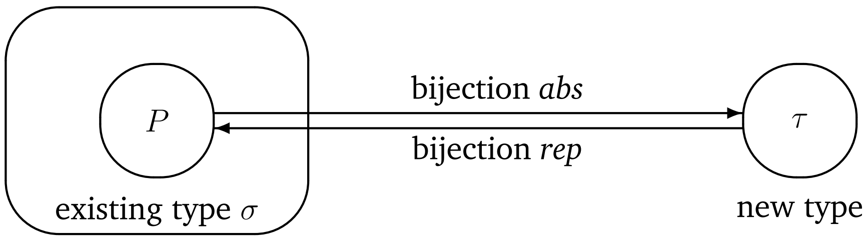
It also returns the above theorem, and in addition saves it as a definition in the current theory under the name *name*.

Note that `new_specification` cannot be used to create recursive functions; t cannot refer to the new c_i being created.

Since the theorem `|- ?`$x_1 \ldots x_n$`.`t is true, there exist values for the constants $c_1, \ldots, c_n$ such that the definition theorem above is true. However, this is all that is known about the new constants; in general we do not know their exact values, just that they together satisfy the property $t[c_1, \ldots, c_n/x_1, \ldots, x_n]$. That is why this is called a *specification* rather than a definition. Even if specific witnesses were used to prove `|- ?`$x_1 \ldots x_n\,.\,t$ originally, the values of the new constants need not be the same as those witnesses. This means that the values of the constants might not be defined entirely, but only in part, and this partiality is quite useful in not over-specifying a design.

10.1.3 New type constant definition

New type constants, including new type operator constants, may be introduced by the *new type constant definition* principle. The idea is that a new type may be defined as being isomorphic to a non-empty subset of a pre-existing type. Say that σ is the existing type, and that P is the non-empty subset of σ. Then the new type τ can be described as



Here *abs* and *rep* are functions which are bijections (one-to-one and onto) between the subset P of the existing type σ and the newly created type τ. In HOL, this is the *only* primitive means provided to create new types.

The new type constant definition principle is implemented as the ML function

```
new_type_definition : (string * thm) -> thm
```

If P is a term of type σ `->` `bool` for some type σ, containing n distinct type variables (of any kinds), then evaluating `new_type_definition("`*name*`", |- ?`x:σ. P x`)` results in *name* being declared as a new n-ary type constant in the current theory. Note that σ must have kind `ty:`r for some rank r. The n type arguments to *name* occur in the order given by an alphabetic ordering of the names of the corresponding type variables. If the type variables have kinds $k_1, \ldots, k_n$, respectively, then the kind of the new type constant *name* will be k_1 `=> ... =>` k_n `=> ty:`r. The theorem returned by `new_type_definition` will be of the form `|- ?rep:('a,...,'n)`*name* `->` σ`. TYPE_DEFINITION P rep`, and this theorem will also be stored in the current theory under the automatically-generated name *name*`_TY_DEF`. `TYPE_DEFINITION` is a constant defined by:

```
|- TYPE_DEFINITION (P:'a->bool) (rep:'b->'a) =
     (!x' x''. (rep x' = rep x'') ==> (x' = x'')) /\
     (!x. P x = (?x'. x = rep x'))
```

Thus `|- ?rep. TYPE_DEFINITION P rep` asserts that there is a bijection between the newly defined type `('a,...,'n)`*name* and the set of values of type σ that satisfy P.

The use of `new_type_definition` and the definition of new term constants involving this type are more fully explained in *DESCRIPTION*.

10.1.4 New type constant specification

In HOL-Omega, new type constants may also be introduced by the *new type constant specification* principle, where apart from any particular existing types, new types may be introduced according to a general property that describes them. This definitional principle is not present in HOL, and it is the sole new definitional principle in HOL-Omega. This feature is the fundamental basis of abstract data types. As in the corresponding definitional principle for specifying term constants, a theorem must be provided that states that some types exist that satisfy the general property. This definitional principle is implemented as the ML function `new_type_specification`:

```
new_type_specification : string * string list * thm -> thm
```

Evaluating `new_specification("`*name*`", ["`t_1`",...,"`t_n`"], |- ?:`$\alpha_1 \ldots \alpha_n . q$`)`, where

- $t_1 \ldots t_n$ are the names of the n type constants to be created, all distinct,
- $\alpha_1 \ldots \alpha_n$ are n distinct type variables,
- q is a term of type `bool` that contains no free term variables,
- q may contain the $\alpha_1 \ldots \alpha_n$, but has no other free type variables, and
- all the free kind variables of $\alpha_1 \ldots \alpha_n$ and q are also free kind variables of the type of each t_i for $1 \leq i \leq n$,

defines each t_i in the logic as a new type constant with the same kind as α_i, for $1 \leq i \leq n$, with some fixed type values such that

`|-` $q[t_1, \ldots, t_n / \alpha_1, \ldots, \alpha_n]$.

It also returns the above theorem, and in addition saves it as a definition in the current theory under the automatically-generated name *name*`_TY_SPEC`.

We will see this definitional principle demonstrated in this chapter.

10.2 Bit Vectors

The following example is taken from Tom Melham's prescient 1994 paper, "The HOL Logic Extended with Quantification over Type Variables."

Consider the type of non-empty bit vectors. We can characterize these algebraically as an HOL-Omega type `vect`, with two constants `t` and `f` and an associative binary operator `c` which concatenates two bit vectors together.

In addition, we wish for the type of bit vectors to be "initial," in the sense that an initial algebra is initial in category theory. That is, we want the type we come up with to be the "most general" or "best possible" type that satisfies what we ask, and nothing we didn't ask. Being initial means that for any other type that also exhibits the same structure of having two constants and an associative binary operator, there must be one and only one homomorphism from the bit vector type to the other type.

Such a homomorphism is a function ϕ that maintains the computational structure of the bit vector type with its operators after mapping them into the other type. For example, ϕ must map `c` to some binary operation `c'` on the other type that is associative, and also maintains the pattern of computation of `c` when mapped from the bit vector type to the other type:

$$\forall (x : \texttt{vect})(y : \texttt{vect}).\ \phi\ (\texttt{c}\ x\ y) = \texttt{c'}\ (\phi\ x)\ (\phi\ y).$$

This means that we get the same answer if we first combine x and y using `c` and then map the result using ϕ, or instead we first map x to $\phi\ x$ and y to $\phi\ y$, and then combine those values using `c'`; and this should work for any possible values of x and y.

To create this new type of bit vectors, our strategy will be to first prove the existance of a type with the properties mentioned above, and then use the new type constant specification principle described earlier to actually introduce the type as a new type constant in the logic. So we need to prove the following theorem in HOL-Omega:

`vect_exists:`

$$\begin{array}{l}
\quad \vdash \exists{:}\alpha. \\
\qquad\quad \exists (\texttt{t} : \alpha)\ (\texttt{f} : \alpha)\ (\texttt{c} : \alpha \to \alpha \to \alpha). \\
\qquad\qquad (\forall x\ y\ z.\ \texttt{c}\ x\ (\texttt{c}\ y\ z) = \texttt{c}\ (\texttt{c}\ x\ y)\ z) \wedge \\
\qquad\qquad (\forall{:}\beta. \\
\qquad\qquad\quad \forall (\texttt{t'} : \beta)\ (\texttt{f'} : \beta)\ (\texttt{c'} : \beta \to \beta \to \beta). \\
\qquad\qquad\qquad (\forall x\ y\ z.\ \texttt{c'}\ x\ (\texttt{c'}\ y\ z) = \texttt{c'}\ (\texttt{c'}\ x\ y)\ z) \ \Rightarrow \\
\qquad\qquad\qquad (\exists! \phi : \alpha \to \beta. \\
\qquad\qquad\qquad\quad (\phi\ \texttt{t} = \texttt{t'})\ \wedge\ (\phi\ \texttt{f} = \texttt{f'}) \wedge \\
\qquad\qquad\qquad\quad (\forall (x : \alpha)(y : \alpha).\ \phi\ (\texttt{c}\ x\ y) = \texttt{c'}\ (\phi\ x)\ (\phi\ y))))
\end{array}$$

In this theorem, α is a type variable which stands for what will be the new bit vector type, and β is a type variable which stands for all "other" types that also exhibit the same algebraic signature, as described above.

The theorem states that there exists a type α (see line 1) and also values `t`, `f`, and `c` (see line 2) such that `c` is associative (line 3), and for all possible other types β (line 4) and all possible values `t'`, `f'`, and `c'` (line 5) such that `c'` is associative (line 6), there must be one and only one homomorphism ϕ of type $\alpha \to \beta$ (line 7) for which `t` maps to `t'`, `f` maps to `f'` (line 8), and `c'` maintains the pattern of computation of `c` (line 9).

It is important to realize that the universal quantification of the type β on line 4 is necessary. If this were left out, then β would be a free type variable of the theorem, and this would violate a condition required by the new type constant specification principle.

If this requirement were not present, the principle would become unsound. For example, consider a theorem of the form $\vdash \exists{:}\alpha.P$ where a type variable β different from α appears free in P. This theorem implicitly considers the type variable β to be universally quantified, so that is meaning is that for any type σ which could be substituted for β, there exists a type α for which $P[\sigma/\beta]$ is true. But there is no way for α to depend on β, as these are two different type variables.

Take for example the theorem $\vdash \exists{:}\alpha.\ (\forall(x{:}\alpha)(y{:}\alpha).\ x = y) = (\forall(x{:}\beta)(y{:}\beta).\ x = y)$, which can be proven in HOL-Omega by taking $\alpha = \beta$. If we could use new type constant specification with this theorem, then we could create a new type, say `atype`, with the specification theorem $\vdash\ (\forall(x{:}\texttt{atype})(y{:}\texttt{atype}).\ x = y) = (\forall(x{:}\beta)(y{:}\beta).\ x = y)$. We can then substitute in this theorem for β either the type `unit` or the type `bool`, and obtain

$$\vdash\ (\forall(x{:}\texttt{atype})(y{:}\texttt{atype}).\ x = y) = (\forall(x{:}\texttt{unit})(y{:}\texttt{unit}).\ x = y) \quad \text{and}$$
$$\vdash\ (\forall(x{:}\texttt{atype})(y{:}\texttt{atype}).\ x = y) = (\forall(x{:}\texttt{bool})(y{:}\texttt{bool}).\ x = y),$$

from which follows $\vdash\ (\forall(x{:}\texttt{unit})(y{:}\texttt{unit}).\ x = y) = (\forall(x{:}\texttt{bool})(y{:}\texttt{bool}).\ x = y)$. But the type `unit` has exactly one element, whereas the type `bool` has two, so this simplifies to $\vdash\ \texttt{T} = \texttt{F}$, a false theorem.

So to use the theorem `vect_exists`, the type variable β must be universally quantified.

If we can prove the theorem `vect_exists`, we can then use the type and term constant specification principles to establish the new type `vect` and associated constants `VT`, `VF`, and `VCONCAT` such that

```
vect_consts_spec:
    ⊢ (∀x y z. VCONCAT x (VCONCAT y z) = VCONCAT (VCONCAT x y) z) ∧      1
      (∀:β.                                                               2
        ∀(t' : β) (f' : β) (c' : β → β → β).                              3
          (∀x y z. c' x (c' y z) = c' (c' x y) z)  ⇒                      4
          (∃!φ : α → β.                                                   5
             (φ VT = t')  ∧  (φ VF = f') ∧                                6
             (∀(x : α)(y : α). φ (VCONCAT x y) = c' (φ x) (φ y))))        7
```

To prove the theorem `vect_exists`, we could first prove an instance of it for some particular type substituted for α as a witness throughout the body of the theorem (lines 2-9), and then use the forward inference rule `TY_EXISTS` to derive `vect_exists`.

In fact, we can go further, and first prove an instance for particular values substituted for `t`, `f`, and `c`. This means that we would have to establish some new type, say “`bits`”, and then create term constants `t0:bits`, `f0:bits`, and `c0:bits` $\rightarrow$ `bits` $\rightarrow$ `bits` in the logic, for which we can prove the following theorem:

$$\begin{array}{l}
\texttt{bits\_is\_initial:} \\
\quad\quad (\forall x\ y\ z.\ \texttt{c0}\ x\ (\texttt{c0}\ y\ z) = \texttt{c0}\ (\texttt{c0}\ x\ y)\ z) \wedge \\
\quad\quad (\forall{:}\beta. \\
\quad\quad\quad \forall(\texttt{t'} : \beta)\ (\texttt{f'} : \beta)\ (\texttt{c'} : \beta \rightarrow \beta \rightarrow \beta). \\
\quad\quad\quad\quad (\forall x\ y\ z.\ \texttt{c'}\ x\ (\texttt{c'}\ y\ z) = \texttt{c'}\ (\texttt{c'}\ x\ y)\ z)\ \Rightarrow \\
\quad\quad\quad\quad (\exists!\phi : \texttt{bits} \rightarrow \beta. \\
\quad\quad\quad\quad\quad (\phi\ \texttt{t0} = \texttt{t'})\ \wedge\ (\phi\ \texttt{f0} = \texttt{f'}) \wedge \\
\quad\quad\quad\quad\quad (\forall(x : \texttt{bits})(y : \texttt{bits}).\ \phi\ (\texttt{c0}\ x\ y) = \texttt{c'}\ (\phi\ x)\ (\phi\ y))))
\end{array}$$

So we need to establish some type `bits` such that the theorem `bits_is_initial` is true. We can almost use lists of booleans (`bool list`) as this type; we could set `t0 = [T]`, `f0 = [F]`, and `c0 = APPEND`. There’s just one problem; we are trying to represent the type of *non-empty* bit vectors, and `bool list` includes the empty list `[]`.

The answer is to create `bits` as a new type isomorphic to a *subset* of `bool list`, that omits the element `[]`. This can be done in HOL-Omega using the new type constant definition principle described earlier.

10.2.1 Defining a new type

The first step in forming a subset type is to determine the subset predicate; in this example, for a list of booleans l, we choose the predicate λl : `bool list`. $l \neq$ `[]`.

```
- set_trace "Unicode" 0;
> val it = () : unit
- new_theory "bit_vector";
<<HOL message: Created theory "bit_vector">>
> val it = () : unit

- val P = ``\l:bool list. ~(l = [])``;
> val P =
    ``\l. l <> []`` :
  term
```

We then need to prove that the predicate is inhabited, that there is at least one element that satisfies the predicate.

```
- val NOT_CONS_NIL = listTheory.NOT_CONS_NIL;                          2
> val NOT_CONS_NIL =
    |- !a1 a0. a0::a1 <> []
     : thm

- val bits_inhab = TAC_PROOF(([],
   ``?l. ^P l``),
   EXISTS_TAC ``[T]``
   THEN BETA_TAC
   THEN REWRITE_TAC[NOT_CONS_NIL]
  );
> val bits_inhab =
    |- ?l. (\l. l <> []) l
     : thm
```

Using this theorem we can create the new type bits as that subset of bool list.

```
- val bits_def = new_type_definition ("bits", bits_inhab);             3
> val bits_def =
    |- ?rep. TYPE_DEFINITION (\l. l <> []) rep
     : thm
```

10.2.2 Abstraction and representation bijections

The theorem we get from the call to new_type_definition is not very useful as is. The first step is to define two new functions in the logic that map between the new type and the subset of the original type. The new type is considered the "abstract" type, and the prior type is considered the "representation" type. Therefore the two functions to be defined are bits_ABS : bool list -> bits and bits_REP : bits -> bool list. These functions are bijections between the old and new types.

```
- val bits_bijs = define_new_type_bijections                           4
                  {name="bits_bijs",
                   ABS ="bits_ABS",
                   REP ="bits_REP",
                   tyax=bits_def};
> val bits_bijs =
    |- (!a. bits_ABS (bits_REP a) = a) /\
       !r. (\l. l <> []) r <=> (bits_REP (bits_ABS r) = r)
     : thm
```

To further assist the user, HOL provides four ML functions that take a bijections theorem such as bits_bijs, and automatically prove the one-to-one and onto properties of both the abstraction and the representation bijections.

```
- val bits_REP_one_one = BETA_RULE (prove_rep_fn_one_one bits_bijs);      5
> val bits_REP_one_one =
    |- !a a'. (bits_REP a = bits_REP a') <=> (a = a')
     : thm

- val bits_REP_onto    = BETA_RULE (prove_rep_fn_onto    bits_bijs);
> val bits_REP_onto =
    |- !r. r <> [] <=> ?a. r = bits_REP a
     : thm

- val bits_ABS_one_one = BETA_RULE (prove_abs_fn_one_one bits_bijs);
> val bits_ABS_one_one =
    |- !r r'.
         r <> [] ==> r' <> [] ==> ((bits_ABS r = bits_ABS r') <=> (r = r'))
     : thm

- val bits_ABS_onto    = BETA_RULE (prove_abs_fn_onto    bits_bijs);
> val bits_ABS_onto =
    |- !a. ?r. (a = bits_ABS r) /\ r <> []
     : thm
```

The bijection theorem itself can be broken into two useful theorems.

```
- val (bits_ABS_REP,bits_EQ_REP_ABS) = CONJ_PAIR (BETA_RULE bits_bijs);   6
> val bits_ABS_REP =
    |- !a. bits_ABS (bits_REP a) = a
     : thm
  val bits_EQ_REP_ABS =
    |- !r. r <> [] <=> (bits_REP (bits_ABS r) = r)
     : thm
```

The last theorem is actually more useful as an implication, instead of an equality, especially when using the automatic simplification tactics, so we prove that version.

```
- val bits_REP_ABS = store_thm(                                            7
   "bits_REP_ABS",
   ``!r. ~(r = []) ==> (bits_REP (bits_ABS r) = r)``,
   REWRITE_TAC [bits_EQ_REP_ABS]
  );
> val bits_REP_ABS =
    |- !r. r <> [] ==> (bits_REP (bits_ABS r) = r)
     : thm
```

Finally, considering the one-to-one and onto properties of bits_REP, every representing value that bits_REP maps onto will satisfy the original subset predicate.

```
- val bits_REP_NOT_NULL = store_thm(                                    8
   "bits_REP_NOT_NULL",
   ``!a. ~(bits_REP a = [])``,
   SIMP_TAC list_ss [bits_REP_onto,bits_REP_one_one]
  );
> val bits_REP_NOT_NULL =
    |- !a. bits_REP a <> [] : thm
```

10.2.3 Defining new term constants of the new type

Now we are ready to create the constants we need in the type bits. Each definition is based on a value or function on bool lists, and makes use of the bits_ABS and bits_REP bijections to move arguments and function results back and forth between the two types.

```
- val t0_def = Define `t0 = bits_ABS [T]`;                              9
Definition has been stored under "t0_def"
> val t0_def = |- t0 = bits_ABS [T] : thm

- val f0_def = Define `f0 = bits_ABS [F]`;
Definition has been stored under "f0_def"
> val f0_def = |- f0 = bits_ABS [F] : thm

- val c0_def = Define `c0 x y = bits_ABS (bits_REP x ++ bits_REP y)`;
Definition has been stored under "c0_def"
> val c0_def =
    |- !x y. c0 x y = bits_ABS (bits_REP x ++ bits_REP y)
     : thm
```

Based on these definitions, we can now prove properties about these operators. The associativity of c0 follows easily from the associativity of the append operation on lists.

```
- val c0_assoc = store_thm(                                             10
   "c0_assoc",
   ``!x y z:bits. c0 x (c0 y z) = c0 (c0 x y) z``,
   SIMP_TAC list_ss [c0_def,bits_REP_ABS,bits_REP_NOT_NULL]
  );
> val c0_assoc =
    |- !x y z. c0 x (c0 y z) = c0 (c0 x y) z
     : thm
```

10.2.4 Induction on the new type

That was fairly easy, but to prove more difficult theorems about the type bits we will need to fashion more sophisticated proof machinery, such as an induction principle. The induction principle states that for any property P:bits -> bool, the property will hold of all values in the type bits if the property is true of both t0 and f0, and when the property is true of any two elements of `bits`, then it must be true of their concatenation using `c0`.

First we prove a helpful lemma.

```
- val bits_CONS_EQ_REP_ABS_APPEND = store_thm(                              11
   "bits_CONS_EQ_REP_ABS_APPEND",
   ``!x y ys. x :: y :: ys =
              bits_REP (bits_ABS [x]) ++ bits_REP (bits_ABS (y::ys))``,
   SIMP_TAC list_ss [bits_REP_ABS]
  );
> val bits_CONS_EQ_REP_ABS_APPEND =
    |- !x y ys.
         x::y::ys = bits_REP (bits_ABS [x]) ++ bits_REP (bits_ABS (y::ys))
     : thm
```

Now we can state and prove an induction principle for `bits`.

```
- val bits_induct = store_thm(                                              12
   "bits_induct",
   ``!P:bits -> bool.
       (P t0) /\
       (P f0) /\
       (!x y. P x /\ P y ==> P (c0 x y)) ==>
       (!b. P b)``,
   REWRITE_TAC [t0_def,f0_def,c0_def]
   THEN GEN_TAC THEN STRIP_TAC
   THEN ONCE_REWRITE_TAC [GSYM bits_ABS_REP]
   THEN GEN_TAC
   THEN MP_TAC (SPEC ``b:bits`` bits_REP_NOT_NULL)
   THEN SPEC_TAC (``bits_REP b``,``l:bool list``)
   THEN measureInduct_on `LENGTH l`
   THEN Cases_on `l` (* two subgoals *)
   THEN REWRITE_TAC[NOT_CONS_NIL] (* eliminates one subgoal *)
   THEN Cases_on `t` (* two subgoals *)
   THENL
     [ Cases_on `h` (* two subgoals *)
       THEN ASM_REWRITE_TAC [],

       REWRITE_TAC [bits_CONS_EQ_REP_ABS_APPEND]
       THEN ASM_SIMP_TAC list_ss []
     ]
  );
> val bits_induct =
    |- !P. P t0 /\ P f0 /\ (!x y. P x /\ P y ==> P (c0 x y)) ==> !b. P b
     : thm
```

10.2.5 Combinator for recursive functions

The next step is to define a combinator to simplify fashioning recursive functions on `bits`. This is similar to the way that the combinator `FOLDR` can be used to fashion recursive functions on lists without actually making any new recursive definitions. The new combinator will be named "`bits_fold`," with type `'b -> 'b -> ('b -> 'b -> 'b) -> bits -> 'b`. The first three arguments will specify how the resulting function behaves in the three cases where the fourth argument (of type `bits`) is of the form `t0`, `f0`, or `c0 a b` for some `a` and `b`. If `bits_fold` x y op is called on the argument `t0`, it should return x; if it is called on the argument `f0`, it should return y; and if it is called on the argument `c0` a b, it should return $op($`bits_fold` x y op $a)($`bits_fold` x y op $b)$.

We define this function `bits_fold` using two auxilliary functions, `bit_fold1` on booleans and `bits_fold1` on lists of booleans:

```
bit_fold1   (x:'b)  (y:'b)       :  bool -> 'b
bits_fold1  (x:'b)  (y:'b)  op   :  bool list -> 'b
bits_fold   (x:'b)  (y:'b)  op   :  bits -> 'b
```

```
- val bit_fold1_def = Define                                          13
  `(bit_fold1 x y T = x:'b) /\
   (bit_fold1 x y F = y)`;
Definition has been stored under "bit_fold1_def"
> val bit_fold1_def =
    |- (!x y. bit_fold1 x y T = x) /\ !x y. bit_fold1 x y F = y
     : thm

- val bits_fold1_def = Define
   `bits_fold1 (x:'b) (y:'b) (op:'b -> 'b -> 'b) (z :: zs) =
      if zs = []
        then bit_fold1 x y z
        else op (bit_fold1 x y z) (bits_fold1 x y op zs)`;
Definition has been stored under "bits_fold1_def"
> val bits_fold1_def =
    |- !x y op z zs.
         bits_fold1 x y op (z::zs) =
         if zs = [] then
           bit_fold1 x y z
         else
           op (bit_fold1 x y z) (bits_fold1 x y op zs)
     : thm

- val bits_fold_def = Define
   `bits_fold (x:'b) y op z = bits_fold1 x y op (bits_REP z)`;
Definition has been stored under "bits_fold_def"
> val bits_fold_def =
    |- !x y op z. bits_fold x y op z = bits_fold1 x y op (bits_REP z)
     : thm
```

Notice that the function `bits_fold1` is actually undefined on empty boolean lists.

Now we need to prove that `bits_fold` as defined yields the proper answers for each of the three cases of its `bits` argument. The first two cases, on `t0` and `f0`, are easy.

```
- val bits_fold_scalars = store_thm(                                      14
    "bits_fold_scalars",
    ``!(x :'b) (y:'b) (op:'b -> 'b -> 'b).
        (bits_fold x y op t0 = x) /\
        (bits_fold x y op f0 = y)``,
    SIMP_TAC list_ss
      [bits_fold_def,t0_def,f0_def,bits_REP_ABS,bits_fold1_def,bit_fold1_def]
  );
> val bits_fold_scalars =
    |- !x y op. (bits_fold x y op t0 = x) /\ (bits_fold x y op f0 = y)
     : thm
```

The third case for `bits_fold`, on `c0 a b`, is just the statement that `bits_fold` $x\ y\ op$ is a homomorphism. Since `bits_fold` is defined in terms of `bits_fold1`, we will first prove that `bits_fold1` $x\ y\ op$ is a homomorphism.

For clarity, let's abbreviate `bits_fold1` $x\ y\ op$ by f. Then f `: bool list -> 'b` is a homomorphism if and only if $f(a \texttt{ ++ } b) = op(f\ a)(f\ b)$ for all a, b that are non-empty lists of booleans. This equation could be expressed in category theory notation as the following commuting diagram:

$$\begin{tikzcd}
\texttt{list \# list} \arrow[r, "f\ \texttt{\#\#}\ f"] \arrow[d, "\texttt{++}"'] & \texttt{'b \# 'b} \arrow[d, "op"] \\
\texttt{list} \arrow[r, "f"'] & \texttt{'b}
\end{tikzcd}$$

First, because we want to quantify over just those boolean lists that are non-empty, we wish to use the restricted quantifier library, `res_quanLib`. This allows us to say "`!(a:bool list) (b:bool list) :: ( \v.~(v = [])). ...`" The idea is that the universally quantified variables, `a` and `b`, range not over all values of their type, but only over the restricted subset that satisfies the predicate given after the double colon (`::`).

```
- load "res_quanLib";                                                      15
> val it = () : unit
- open res_quanLib;
> ...
```

Then the proof that `bits_fold1` is a homomorphism uses a simple induction on the boolean list `a`, with case splits for the cases in the definition of `bits_fold1`.

16

```
- val bits_fold1_is_homo = store_thm(
    "bits_fold1_is_homo",
    ``!(x :'b) (y:'b) (op:'b -> 'b -> 'b).
        (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
        !(a:bool list) (b:bool list) :: (\v.~(v = [])).
            bits_fold1 x y op (a ++ b) =
            op (bits_fold1 x y op a) (bits_fold1 x y op b)``,
    REPEAT GEN_TAC
    THEN DISCH_TAC
    THEN SIMP_TAC (bool_ss ++ resq_SS) [pred_setTheory.IN_ABS]
    THEN Induct (* two subgoals *)
    THEN REWRITE_TAC [NOT_CONS_NIL] (* eliminates one subgoal *)
    THEN REWRITE_TAC [listTheory.APPEND,bits_fold1_def]
    THEN Cases_on `a` (* two subgoals *)
    THEN ASM_SIMP_TAC list_ss []
  );
> val bits_fold1_is_homo =
    |- !x y op.
         (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
         !a b::(\v. v <> []).
           bits_fold1 x y op (a ++ b) =
           op (bits_fold1 x y op a) (bits_fold1 x y op b)
     : thm
```

Given that `bits_fold1` is a homomorphism, it is easy to prove that `bits_fold` is a homomorphism as well.

For clarity, let's abbreviate `bits_fold` $x\ y\ op$ by f. Then f : `bits -> 'b` is a homomorphism if and only if $f(\texttt{c0}\ a\ b) = op(f\ a)(f\ b)$ for all a, b in `bits`. This equation could be expressed in category theory notation as the following commuting diagram:

$$\begin{tikzcd}
\texttt{bits \# bits} \arrow[r, "f\ \texttt{\#\#}\ f"] \arrow[d, "\texttt{c0}"'] & \texttt{'b \# 'b} \arrow[d, "op"] \\
\texttt{bits} \arrow[r, "f"'] & \texttt{'b}
\end{tikzcd}$$

The proof is accomplished just by simplification, using the theorem `bits_fold1_is_homo`.

```
- val bits_fold_is_homo = store_thm(                                        17
   "bits_fold_is_homo",
   ``!(x :'b) (y:'b) (op:'b -> 'b -> 'b).
       (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
       !a b.
           bits_fold x y op (c0 a b) =
           op (bits_fold x y op a) (bits_fold x y op b)``,
   SIMP_TAC (list_ss ++ resq_SS)
        [bits_fold_def,c0_def,bits_REP_ABS,bits_REP_NOT_NULL,
         pred_setTheory.IN_ABS,bits_fold1_is_homo]
  );
> val bits_fold_is_homo =
    |- !x y op.
         (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
         !a b.
           bits_fold x y op (c0 a b) =
           op (bits_fold x y op a) (bits_fold x y op b)
     : thm
```

This completes the proof that the recursive function formed by `bits_fold` $x\ y\ op$ has the correct behavior on each of the three cases of how a value of the type `bits` was formed, whether by `t0`, `f0`, or by `c0` $a\ b$.

10.2.6 Proof of initiality

Now we can prove that `bits` with `t0`, `f0`, `c0` is initial; that is, there is exactly one homomorphism from `bits` to any other type `'b` with t': `'b`, f': `'b`, and c': `'b -> 'b -> 'b`, where c' is associative.

`bits_is_initial:`

$$\vdash \forall{:}\beta.\ \forall(\texttt{t'} : \beta)\ (\texttt{f'} : \beta)\ (\texttt{c'} : \beta \to \beta \to \beta). \qquad 1$$
$$(\forall x\ y\ z.\ \texttt{c'}\ x\ (\texttt{c'}\ y\ z) = \texttt{c'}\ (\texttt{c'}\ x\ y)\ z)\ \Rightarrow \qquad 2$$
$$(\exists!\phi : \texttt{bits} \to \beta. \qquad 3$$
$$(\phi\ \texttt{t0} = \texttt{t'})\ \wedge\ (\phi\ \texttt{f0} = \texttt{f'}) \wedge \qquad 4$$
$$(\forall (x:\alpha)(y:\alpha).\ \phi\ (\texttt{c0}\ x\ y) = \texttt{c'}\ (\phi\ x)\ (\phi\ y))) \qquad 5$$

This proof is interesting, and we will trace it in some detail here. We begin by establishing the goal to be proved.

```
- g `!:'b. !(x:'b) (y:'b) (op:'b -> 'b -> 'b).                           18
             (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
             ?!(fn:bits -> 'b).
                       (fn  t0          = x) /\
                       (fn  f0          = y) /\
                 (!a b. fn (c0 a b) = op (fn a) (fn b))`;
> val it =
    Proof manager status: 1 proof.
    1. Incomplete goalstack:
         Initial goal:

         !:'b.
           !x y op.
             (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
             ?!fn.
               (fn t0 = x) /\ (fn f0 = y) /\
               !a b. fn (c0 a b) = op (fn a) (fn b)


     : proofs
```

First we remove the universal quantifications and move the antecedent of the implication to the assumption list.

```
- e (REPEAT STRIP_TAC);                                                  19
OK..
1 subgoal:
> val it =

    ?!fn. (fn t0 = x) /\ (fn f0 = y) /\ !a b. fn (c0 a b) = op (fn a) (fn b)
    ------------------------------------
      !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
     : proof
```

Next we transform the unique existence quantification into a conjunction of two clauses, one of which states the existence, and the other the uniqueness.

```
- e (SIMP_TAC bool_ss [EXISTS_UNIQUE_DEF]);                              20
OK..
1 subgoal:
> val it =

    (?fn.
       (fn t0 = x) /\ (fn f0 = y) /\
       !a b. fn (c0 a b) = op (fn a) (fn b)) /\
    !x' y'.
      ((x' t0 = x) /\ (x' f0 = y) /\
       !a b. x' (c0 a b) = op (x' a) (x' b)) /\ (y' t0 = x) /\
      (y' f0 = y) /\ (!a b. y' (c0 a b) = op (y' a) (y' b)) ==>
      (x' = y')
    ------------------------------------
      !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
     : proof
```

Now we split this conjunction into two subgoals, and deferring uniqueness until later, we begin to work on the first subgoal, the existence property.

```
- e (CONJ_TAC);                                                          21
OK..
2 subgoals:
> val it =

    !x' y'.
      ((x' t0 = x) /\ (x' f0 = y) /\
       !a b. x' (c0 a b) = op (x' a) (x' b)) /\ (y' t0 = x) /\
      (y' f0 = y) /\ (!a b. y' (c0 a b) = op (y' a) (y' b)) ==>
      (x' = y')
    ------------------------------------
      !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3


    ?fn. (fn t0 = x) /\ (fn f0 = y) /\ !a b. fn (c0 a b) = op (fn a) (fn b)
    ------------------------------------
      !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
     : proof
```

Now is when we make use of our `bits_fold` combinator to supply the right function on `bits`, given the values `x:'b`, `y:'b`, and `op:'b -> 'b -> 'b`.

```
- e (EXISTS_TAC ``bits_fold (x:'b) y op``);                               22
OK..
1 subgoal:
> val it =

    (bits_fold x y op t0 = x) /\ (bits_fold x y op f0 = y) /\
    !a b.
      bits_fold x y op (c0 a b) =
      op (bits_fold x y op a) (bits_fold x y op b)
    ------------------------------------
      !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
     : proof
```

The clauses on `t0` and `f0` are easily solved using the theorem `bits_fold_scalars`.

```
- e (REWRITE_TAC[bits_fold_scalars]);                                     23
OK..
1 subgoal:
> val it =

    !a b.
      bits_fold x y op (c0 a b) =
      op (bits_fold x y op a) (bits_fold x y op b)
    ------------------------------------
      !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
     : proof
```

The rest of the goal is almost exactly an instance of `bits_fold_is_homo`.

```
- e (POP_ASSUM MP_TAC THEN REWRITE_TAC[bits_fold_is_homo]);              24
OK..

Goal proved.
 [.]
|- !a b.
     bits_fold x y op (c0 a b) =
     op (bits_fold x y op a) (bits_fold x y op b)

Goal proved.
 [.]
|- (bits_fold x y op t0 = x) /\ (bits_fold x y op f0 = y) /\
   !a b.
     bits_fold x y op (c0 a b) =
     op (bits_fold x y op a) (bits_fold x y op b)

Goal proved.
 [.]
|- ?fn.
     (fn t0 = x) /\ (fn f0 = y) /\ !a b. fn (c0 a b) = op (fn a) (fn b)
```

This completes the proof of the existence half of this theorem, and the HOL-Omega goalstack returns us to the pending uniqueness goal.

```
Remaining subgoals:                                                         25
> val it =

    !x' y'.
      ((x' t0 = x) /\ (x' f0 = y) /\
       !a b. x' (c0 a b) = op (x' a) (x' b)) /\ (y' t0 = x) /\
      (y' f0 = y) /\ (!a b. y' (c0 a b) = op (y' a) (y' b)) ==>
      (x' = y')
    ------------------------------------
      !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
     : proof
```

We can clean this up by renaming the two universally quantified functions x' and y' as the more descriptive f and g, respectively, when we remove the quantifications.

```
- e (Q.X_GEN_TAC `f` THEN Q.X_GEN_TAC `g`);                                 26
OK..
1 subgoal:
> val it =

    ((f t0 = x) /\ (f f0 = y) /\ !a b. f (c0 a b) = op (f a) (f b)) /\
    (g t0 = x) /\ (g f0 = y) /\ (!a b. g (c0 a b) = op (g a) (g b)) ==>
    (f = g)
    ------------------------------------
      !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
     : proof
```

The next step is to move the antecedent to the hypotheses, breaking up the conjunction into its clauses.

```
- e (STRIP_TAC);                                                            27
OK..
1 subgoal:
> val it =

    f = g
    ------------------------------------
      0.  !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
      1.  f t0 = x
      2.  f f0 = y
      3.  !a b. f (c0 a b) = op (f a) (f b)
      4.  g t0 = x
      5.  g f0 = y
      6.  !a b. g (c0 a b) = op (g a) (g b)
     : proof
```

To prove two functions are equal, we use the principle of extensionality of functions in the logic.

```
- e (CONV_TAC FUN_EQ_CONV);                                          28
OK..
1 subgoal:
> val it =

    !b. f b = g b
    ------------------------------------
      0.  !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
      1.  f t0 = x
      2.  f f0 = y
      3.  !a b. f (c0 a b) = op (f a) (f b)
      4.  g t0 = x
      5.  g f0 = y
      6.  !a b. g (c0 a b) = op (g a) (g b)
     : proof
```

This goal is a property of all values `b` of type `bits`. We can prove this using our previously proven induction principle, `bits_induct`. We will use higher order matching to have the expression `P b` in the consequent of the induction principle be matched to the full body of the current goal.

```
- bits_induct;                                                       29
> val it =
    |- !P. P t0 /\ P f0 /\ (!x y. P x /\ P y ==> P (c0 x y)) ==> !b. P b
     : thm

- e (HO_MATCH_MP_TAC bits_induct);
OK..
1 subgoal:
> val it =

    (f t0 = g t0) /\ (f f0 = g f0) /\
    !b b'. (f b = g b) /\ (f b' = g b') ==> (f (c0 b b') = g (c0 b b'))
    ------------------------------------
      0.  !b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3
      1.  f t0 = x
      2.  f f0 = y
      3.  !a b. f (c0 a b) = op (f a) (f b)
      4.  g t0 = x
      5.  g f0 = y
      6.  !a b. g (c0 a b) = op (g a) (g b)
     : proof
```

That was exactly the induction we needed. Now the rest of this goal can be easily solved by a single, simple step, moving all the antecedents to the assumptions, and then rewriting the goal with all the assumptions.

30

```
- e (REPEAT STRIP_TAC THEN ASM_REWRITE_TAC []);
OK..

Goal proved.
 [......]
|- (f t0 = g t0) /\ (f f0 = g f0) /\
   !b b'. (f b = g b) /\ (f b' = g b') ==> (f (c0 b b') = g (c0 b b'))

Goal proved.
 [......] |- !b. f b = g b

Goal proved.
 [......] |- f = g

Goal proved.
|- ((f t0 = x) /\ (f f0 = y) /\ !a b. f (c0 a b) = op (f a) (f b)) /\
   (g t0 = x) /\ (g f0 = y) /\ (!a b. g (c0 a b) = op (g a) (g b)) ==>
   (f = g)

Goal proved.
|- !x' y'.
     ((x' t0 = x) /\ (x' f0 = y) /\
      !a b. x' (c0 a b) = op (x' a) (x' b)) /\ (y' t0 = x) /\
     (y' f0 = y) /\ (!a b. y' (c0 a b) = op (y' a) (y' b)) ==>
     (x' = y')

Goal proved.
 [.]
|- (?fn.
      (fn t0 = x) /\ (fn f0 = y) /\
      !a b. fn (c0 a b) = op (fn a) (fn b)) /\
   !x' y'.
     ((x' t0 = x) /\ (x' f0 = y) /\
      !a b. x' (c0 a b) = op (x' a) (x' b)) /\ (y' t0 = x) /\
     (y' f0 = y) /\ (!a b. y' (c0 a b) = op (y' a) (y' b)) ==>
     (x' = y')

Goal proved.
 [.]
|- ?!fn.
     (fn t0 = x) /\ (fn f0 = y) /\ !a b. fn (c0 a b) = op (fn a) (fn b)
> val it =
    Initial goal proved.
    |- !:'b.
         !x y op.
           (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
           ?!fn.
             (fn t0 = x) /\ (fn f0 = y) /\
             !a b. fn (c0 a b) = op (fn a) (fn b)
     : proof
```

We can package up this proof into a single tactic in the following proof script.

31

```
- val bits_is_initial = store_thm(
   "bits_is_initial",
   ``!:'b. !(x:'b) (y:'b) (op:'b -> 'b -> 'b).
              (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
              ?!(fn:bits -> 'b).
                        (fn  t0           = x) /\
                        (fn  f0           = y) /\
                  (!a b. fn (c0 a b) = op (fn a) (fn b))``,
   REPEAT STRIP_TAC
   THEN SIMP_TAC bool_ss [EXISTS_UNIQUE_DEF]
   THEN CONJ_TAC
   THENL
     [ EXISTS_TAC ``bits_fold (x:'b) y op``
       THEN REWRITE_TAC[bits_fold_scalars]
       THEN POP_ASSUM MP_TAC
       THEN REWRITE_TAC[bits_fold_is_homo],

       Q.X_GEN_TAC `f`
       THEN Q.X_GEN_TAC `g`
       THEN STRIP_TAC
       THEN CONV_TAC FUN_EQ_CONV
       THEN HO_MATCH_MP_TAC bits_induct (* induct on b *)
       THEN REPEAT STRIP_TAC (* 3 subgoals *)
       THEN ASM_REWRITE_TAC []
     ]
  );
> val bits_is_initial =
    |- !:'b.
         !x y op.
           (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
           ?!fn.
             (fn t0 = x) /\ (fn f0 = y) /\
             !a b. fn (c0 a b) = op (fn a) (fn b)
     : thm
```

10.2.7 Existence and creation of bit vector type

Now that we've proven `bits_is_initial`, it follows immediately that there exists some type `'a` with `t:'a`, `f:'a`, and `c:'a -> 'a -> 'a`, with `c` associative, where the type and operators are initial.

```
- val vect_exists = store_thm(                                          32
   "vect_exists",
   ``?:'a.
       ?(t:'a) (f:'a) (c:'a -> 'a -> 'a).
         (!a1 a2 a3. c a1 (c a2 a3) = c (c a1 a2) a3) /\
         (!:'b.
            !(x:'b) (y:'b) (op:'b -> 'b -> 'b).
              (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
              ?!fn:'a -> 'b.         (fn t         = x) /\
                                     (fn f         = y) /\
                            (!a1 a2. fn (c a1 a2) = op (fn a1) (fn a2)))``,
   TY_EXISTS_TAC ``:bits``
   THEN EXISTS_TAC ``t0``
   THEN EXISTS_TAC ``f0``
   THEN EXISTS_TAC ``c0``
   THEN REWRITE_TAC [c0_assoc,bits_is_initial]
  );
> val vect_exists =
    |- ?:'a.
         ?t f c.
           (!a1 a2 a3. c a1 (c a2 a3) = c (c a1 a2) a3) /\
           !:'b.
             !x y op.
               (!b1 b2 b3. op b1 (op b2 b3) = op (op b1 b2) b3) ==>
               ?!fn.
                 (fn t = x) /\ (fn f = y) /\
                 !a1 a2. fn (c a1 a2) = op (fn a1) (fn a2)
     : thm
```

This achieves our goal of proving `vect_exists`. Now we can use this theorem with the type specification principle, as previously described, to actually introduce a new type name into the HOL-Omega logic, about which the only things known are associativity and initiality. We turn on the display of types to highlight the presence of the new type.

```
- show_types := true;                                                   33
> val it = () : unit

- val vect_TY_SPEC = new_type_specification("vect",["vect"],vect_exists);
> val vect_TY_SPEC =
    |- ?(t :vect) (f :vect) (c :vect -> vect -> vect).
         (!(a1 :vect) (a2 :vect) (a3 :vect).
            c a1 (c a2 a3) = c (c a1 a2) a3) /\
         !:'b.
           !(x :'b) (y :'b) (op :'b -> 'b -> 'b).
             (!(b1 :'b) (b2 :'b) (b3 :'b).
                op b1 (op b2 b3) = op (op b1 b2) b3) ==>
             ?!(fn :vect -> 'b).
               (fn t = x) /\ (fn f = y) /\
               !(a1 :vect) (a2 :vect). fn (c a1 a2) = op (fn a1) (fn a2)
     : thm
```

10.2.8 Creation of bit vector constants

Now that we have the theorem `vect_TY_SPEC` which states the existence of values `t`, `f`, and `c`, we can use the definitional principle for term constant specification to introduce actual term constants of the new type into the logic.

```
- val vect_consts_spec =                                                   34
    new_specification ("vect_consts_spec",
                       [ "VT", "VF", "VCONCAT" ],
                       vect_TY_SPEC);
> val vect_consts_spec =
    |- (!(a1 :vect) (a2 :vect) (a3 :vect).
          VCONCAT a1 (VCONCAT a2 a3) = VCONCAT (VCONCAT a1 a2) a3) /\
       !:'b.
         !(x :'b) (y :'b) (op :'b -> 'b -> 'b).
           (!(b1 :'b) (b2 :'b) (b3 :'b).
              op b1 (op b2 b3) = op (op b1 b2) b3) ==>
           ?!(fn :vect -> 'b).
             (fn VT = x) /\ (fn VF = y) /\
             !(a1 :vect) (a2 :vect). fn (VCONCAT a1 a2) = op (fn a1) (fn a2)
     : thm

- type_of ``VT``;
> val it = ``:vect`` : hol_type
- type_of ``VF``;
> val it = ``:vect`` : hol_type
- type_of ``VCONCAT``;
> val it =
    ``:vect -> vect -> vect``
     : hol_type
```

For convenience, we can split the term constants property into its conjuncts.

```
- val (VCONCAT_ASSOC, vect_is_initial) = CONJ_PAIR vect_consts_spec;       35
> val VCONCAT_ASSOC =
    |- !(a1 :vect) (a2 :vect) (a3 :vect).
         VCONCAT a1 (VCONCAT a2 a3) = VCONCAT (VCONCAT a1 a2) a3
     : thm
  val vect_is_initial =
    |- !:'b.
         !(x :'b) (y :'b) (op :'b -> 'b -> 'b).
           (!(b1 :'b) (b2 :'b) (b3 :'b).
              op b1 (op b2 b3) = op (op b1 b2) b3) ==>
           ?!(fn :vect -> 'b).
             (fn VT = x) /\ (fn VF = y) /\
             !(a1 :vect) (a2 :vect). fn (VCONCAT a1 a2) = op (fn a1) (fn a2)
     : thm
```

10.2.9 Bit vectors as an abstract data type

The key thing here is that these two properties, `VCONCAT_ASSOC` and `vect_is_initial`, are literally all that is known about values of type `vect`. For the type `bits` that we created earlier, if we wanted to, we could reach back into the representation type of `bits` to derive new properties about `bits` itself. In contrast to this, no such underlying structure is available for `vect`. In particular, there is no theorem relating values of `vect` and `bits`, as there were relating values of `bits` and `bool list`. While the `bits` type was necessary to demonstrate the existence of `vect`, that does not mean that `vect` has all the properties of `bits`, but only those properties we designed `vect` to have. In this case, since initiality is such a powerful property, many of the properties of `bits` could now be derived for `vect`, but this approach allows us to limit the provable properties of the specified type to the particular subset we choose. We have truly achieved here an abstract data type, having just the properties we wished, and nothing more.

Chapter 11

Example: The Category of Types

This chapter exhibits the new features of HOL-Omega in an example of real use. It reveals some of the potential and power provided, and what new logical developments are possible beyond those of HOL. This chapter studies a simplified version of category theory, modeled as a shallow embedding in the HOL-Omega logic, as a worked example. The goals of this chapter are:

(i) To demonstrate the flexibility and consistency of the new logic's type system because of the kind system,

(ii) To show type abbreviations and term definitions involving the new forms, and

(iii) To show proofs of propositions involving the new forms, using both new tactics and variations of old ones.

Category theory is a fascinating and deep mathematical theory that has great generality and applicability to many fields of mathematics. It is very simple in its foundations, and yet has surprisingly profound results that are widely adaptable in different contexts.

For beginners in category theory, a good book is *Basic Category Theory for Computer Scientists,* by Benjamin C. Pierce, MIT Press, 1991.

11.1 Categories

A category consists of *objects* and *arrows,* where each arrow points from a *source* object to a *target* object. We can represent that an arrow f points from object a to object b by the notation $f : a \to b$. Sometimes we will write the name of the arrow above it, as in $a \xrightarrow{f} b$. The category must contain, for each object a, a corresponding *identity arrow* $id_a : a \to a$, pointing from the object to itself. Also, for each pair of arrows $f : a \to b$ and $g : b \to c$, where the target of the first arrow is the source of the second, there must exist an arrow which is the *composition* of f and g, written $g \circ f$. In addition, the composition operation must satisfy the following two laws:

1. **(identity)** $f \circ id_a = f$ and $id_b \circ f = f$ for all $f : a \to b$,

2. **(associativity)** $h \circ (g \circ f) = (h \circ g) \circ f$ for all $f : a \to b$, $g : b \to c$, and $h : c \to d$.

In this chapter, we shall examine one particular category, which we call **Type**. In **Type**, the objects are the types of the HOL-Omega logic that have kind **ty**:0, and the arrows are the term functions in the logic between such types. Then the identity arrow for a type α is the identity function `I : `α` -> `α, defined as `I = \x:`α`.x`. The composition of two arrows is simply functional composition. For functions `f : `α` -> `β and `g : `β` -> `γ, their composition is defined in HOL-Omega as `g o f = \x. g (f x)`, where `o` is an infix binary operator. With these definitions, it is easy to see that the above laws hold.

In any category, when two arrows are composed, one must check that the target of one arrow is the source of the other. In the category **Type**, this customary check is performed fully automatically as part of the type-checking of the logic. Thus this check is discharged immediately, simply due to the fact that the HOL-Omega logic is strongly typed. This is very convenient, as the user never needs to worry about performing this check themselves; it is all handled silently by the type-checker.

11.2 Functors

Along with natural transformations, functors are one of the major ideas of category theory. A functor is a pair of two maps, one from objects to objects, and one from arrows to arrows, that satisfy certain properties.

For our purposes we focus on just functors from the category **Type** to itself. These functors have a map from types to types, and a map from functions to functions. The first map we express as a type operator in the HOL-Omega logic, of kind **ty**:0 ⇒ **ty**:0. The second map we express as a higher-order function in the logic which takes a function as an argument and returns a function as its result. If the object map is a type operator $'F$: **ty**:0 ⇒ **ty**:0, then the arrow map is a term function of the type $'F$ functor, where functor is the type abbreviation

$$\textsf{functor} = \lambda' F.\ \forall \alpha\ \beta.\ (\alpha \to \beta) \to (\alpha\ 'F \to \beta\ 'F).$$

A type abbreviation does not actually introduce a new type constant into the logic, but it provides a name that can be expanded by the type parser, to allow inputs that are more pleasant and readable. The same type abbreviation is also used when printing types, shrinking the output to make the printed versions of types more readable as well. Such a type abbreviation is introduced by the ML function `type_abbrev`.

```
- val _ = new_theory "functor";                                        1
<<HOL message: Created theory "functor">>
- set_trace "Unicode" 0;
> val it = () : unit

- type_abbrev ("functor",
               ``: \'F. !'a 'b. ('a -> 'b) -> ('a 'F -> 'b 'F)``);
> val it = () : unit
```

We can see how types are condensed by using this type abbreviation.

```
- ``:!'a 'b. ('a -> 'b) -> ('a list -> 'b list)``;                        2
> val it = ``:list functor`` : hol_type
- ``:!'a 'b. ('a -> 'b) -> (('a -> 'a) -> ('b -> 'b))``;
> val it = ``:(\'b. 'b -> 'b) functor`` : hol_type
```

The type abbreviations that are presently defined may be displayed by printing the current type grammar. After the grammar rules is a list of the current type abbreviations.

```
- type_grammar();                                                          3
> val it =
    Rules:
      (20)   TY  ::=  "!" <..binders..> "." TY | "?" <..binders..> "." TY |
                      "\" <..binders..> "." TY
      (50)   TY  ::=  TY o TY [comp] | TY -> TY [fun] (R-associative)
      (60)   TY  ::=  TY + TY [sum] (R-associative)
      (70)   TY  ::=  TY # TY [prod] (R-associative)
      (80)   TY  ::=  TY TY (type application)
      (90)   TY  ::=  TY : KIND | TY :<= RANK (kind or rank cast of type)
      (100)  TY  ::=  functor | TY list | TY set | TY recspace |
                             num | (TY, TY)prod | TY option | unit | one |
                             (TY, TY)sum | label | ('k => TY)itself |
                             ('k => TY)kind_itself | comp | W | C | B | A |
                             S | K | I | (TY, TY)fun | ind | bool

             TY  ::=  TY[TY] (array type)
    Type abbreviations:
      A = \'f :'k => 'l. 'f     (not printed)
      B = \'f :'l => 'm 'g :'k => 'l 'a :'k. 'a 'g 'f     (not printed)
      C = \'f :'l => 'k => 'm 'a :'k 'b :'l. ('b, 'a) 'f     (not printed)
      I = \'a :'k. 'a
      K = \'a :'k 'b :'l. 'a     (not printed)
      S =
        \'a :'k => 'l => 'm 'b :'k => 'l 'c :'k. ('c, 'c 'b) 'a
        (not printed)
      W = \'f :'k => 'k => 'l 'a :'k. ('a, 'a) 'f     (not printed)
      comp = \'g :'l => 'm 'f :'k => 'l 'a :'k. 'a 'f 'g
      functor = \'F :ty => ty. !'a 'b. ('a -> 'b) -> 'a 'F -> 'b 'F
      'a set = 'a -> bool     (not printed)
      unit = one : grammar
```

The list of type abbreviations includes the functor abbreviation that was just defined, but also a number of other useful abbreviations, including type operator versions of the combinators A, B, C, I, K, S, and W (not to be confused with the *term* versions of these combinators). Of particular value are the abbreviations for the identity type operator `I = \('a:'k).'a` and the constant type operator `K = \('a:'k)('b:'l).'a`. Also, for composing two type operators, there is an infix type abbreviation `o`, which is defined as a postfix binary type operator `comp`, but whenever possible prints as the infix `o`.

```
- ``:comp``;                                                          4
> val it = ``:\'g :ty => ty 'f :ty => ty 'a. 'a 'f 'g`` : hol_type
- ``:('a,'b)comp``;
> val it = ``:('a :ty => ty) o ('b :ty => ty)`` : hol_type
```

Given the type abbreviation functor, we can now define the *functor* predicate in the logic, which decides if a given function F is in fact a functor by testing whether it satisfies two conditions. A functor maps identity arrows to identity arrows, and it maps the composition of two arrows to the composition of the maps of each arrow individually.

$$\begin{array}{ll} \textit{functor}\ (F : {}'F\ \text{functor}) = & \\ \quad (\forall{:}\alpha.\ F\,(\mathtt{I} : \alpha \to \alpha) = \mathtt{I})\ \wedge & \textit{Identity} \\ \quad (\forall{:}\alpha\ \beta\ \gamma.\ \forall(f : \alpha \to \beta)(g : \beta \to \gamma).\ F\,(g \circ f) = F\,g \circ F\,f\,) & \textit{Composition} \end{array}$$

```
- val functor_def = new_definition("functor_def", Term                 5
   `functor (F': 'F functor) =
      (* Identity *)
          (!:'a. F' (I:'a->'a) = I) /\
      (* Composition *)
          (!:'a 'b 'c. !(f:'a -> 'b) (g:'b -> 'c).
                 F' (g o f) = F' g o F' f)
     `);
> val functor_def =
    |- !F'.
         functor F' <=>
         (!:'a. F' I = I) /\ !:'a 'b 'c. !f g. F' (g o f) = F' g o F' f
     : thm
```

By the way, in this definition we use the variable name `F'` for the functor because the name `F` is already reserved for the truth value false.

By default, the types involved in the definition are not printed, including the type arguments in type application terms. Turning on the printing of types shows the full structure of the definition, including type arguments which were not originally present in the user's input but inferred and inserted during type-checking.

```
- show_types := true;                                                  6
> val it = () : unit

- functor_def;
> val it =
    |- !(F' :('F :ty => ty) functor).
         functor F' <=>
         (!:'a. F' [:'a, 'a:] (I :'a -> 'a) = (I :'a 'F -> 'a 'F)) /\
         !:'a 'b 'c.
           !(f :'a -> 'b) (g :'b -> 'c).
             F' [:'a, 'c:] (g o f) = F' [:'b, 'c:] g o F' [:'a, 'b:] f
     : thm
```

The last line of the definition shows three different type instantiations of the variable `F'`, all within the same expression. This would have been impossible in HOL, as there a variable can only have one type within a single expression. The use of universal types in the type of `F'` permit one to "manually" specify the particular type instantiation needed for each occurrence of `F'`. This is in contrast to the "automatic" instantiations of term constants for different types within the same expression that have long been a standard part of higher order logic. Despite having these two different means for forming type instances of terms within the same logic, everything works cleanly, without confusion.

In the following, these type application arguments will normally be omitted for clarity.

11.2.1 Examples of functors

11.2.1.1 Identity functor

The identity function in HOL is the constant `I`, defined as `I = \x:'a.x`. When wrapped by two type abstractions as `\:'a 'b. I: ('a -> 'b) -> ('a -> 'b)`, it becomes a functor of type I functor. This is the identity functor, which maps each object to itself, and each arrow to itself. We can prove that this is a functor, using the simplification set for combinators to help simplify expressions with `I` and `o`.

```
- show_types := false;                                                    7
> val it = () : unit

- open combinTheory combinSimps;
> . . .

- val combin_ss = bool_ss ++ COMBIN_ss;
> val combin_ss = . . .

- val identity_functor = store_thm
  ("identity_functor",
   ``functor ((\:'a 'b. I) : I functor)``,
   SIMP_TAC combin_ss [functor_def]
  );
> val identity_functor = |- functor (\:'a 'b. I) : thm
```

11.2.1.2 Constant functor

One of the combinators in HOL is the constant `K`, defined as `K = \(x:'a)(y:'b).x`. Similarly, the type abbreviation K is defined as `\('a:'k)('b:'l).'a`. Then `\:'b 'c. K I : ('b -> 'c) -> ('a -> 'a)` is a functor of type (`'a` K) functor. This constant functor maps every object to a particular object `'a`, and every arrow to the identity arrow on `'a`.

```
- val constant_functor = store_thm                                   8
  ("constant_functor",
   ``!:'a. functor ((\:'b 'c. K I) : ('a K) functor)``,
   SIMP_TAC combin_ss [functor_def]
  );
> val constant_functor =
    |- !:'a. functor (\:'b 'c. K I) : thm
```

11.2.1.3 List map functor

In HOL, the map function on lists is defined as

```
- listTheory.MAP;                                                    9
> val it =
    |- (!f. MAP f [] = []) /\ !f h t. MAP f (h::t) = f h::MAP f t
     : thm
- type_of ``MAP``;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it =
    ``:('a -> 'b) -> 'a list -> 'b list``
     : hol_type
```

When wrapped with two type abstractions as `\:'a 'b. MAP`, it becomes a functor of type list functor. This functor maps each object `'a` to `'a list`, and each arow `f : 'a -> 'b` to the arrow `MAP f : 'a list -> 'b list`. To prove this is a functor, we bring in two preproven theorems about `MAP` from other parts of HOL-Omega.

```
- load "quotientLib";                                                10
> val it = () : unit
- val MAP_I = quotient_listTheory.LIST_MAP_I;
> val MAP_I = |- MAP I = I : thm
- val MAP_o = rich_listTheory.MAP_o;
> val MAP_o =
    |- !f g. MAP (f o g) = MAP f o MAP g
     : thm

- val map_functor = store_thm
  ("map_functor",
   ``functor ((\:'a 'b. MAP) : list functor)``,
   SIMP_TAC bool_ss [functor_def,MAP_I,MAP_o]
  );
> val map_functor = |- functor (\:'a 'b. MAP) : thm
```

11.2.1.4 Diagonal functor

In HOL, two types `'a` and `'b` may be combined into a *pair type* using the infix binary type operator `#`, as `'a # 'b`. Values of this type are written as `(x:'a, y:'b)`. Then two functions `f :'a -> 'c` and `g :'b -> 'd` may be combined using the infix binary operator `##` as `(f ## g) : 'a # 'b -> 'c # 'd`. The operator `##` is defined as

```
- pairTheory.PAIR_MAP_THM;                                              11
> val it =
    |- !f g x y. (f ## g) (x,y) = (f x,g y)
     : thm
```

The diagonal functor maps objects `'a` to pair types `'a # 'a`, and arrows `f :'a -> 'b` to arrows `(f ## f) :'a # 'a -> 'b # 'b`.

```
- load "quotient_pairTheory";                                           12
> val it = () : unit
- val PAIR_I = quotient_pairTheory.PAIR_MAP_I;
> val PAIR_I = |- I ## I = I : thm
- val PAIR_o = quotient_pairTheory.PAIR_MAP_o;
> val PAIR_o =
    |- !f1 g1 f2 g2. g1 o f1 ## g2 o f2 = (g1 ## g2) o (f1 ## f2)
     : thm

- val diagonal_functor = store_thm
  ("diagonal_functor",
   ``functor ((\:'b 'c. \f. f ## f) : (\'a. 'a # 'a) functor)``,
   SIMP_TAC bool_ss [functor_def,PAIR_I,PAIR_o]
  );
> val diagonal_functor =
    |- functor (\:'b 'c. (\f. f ## f))
     : thm
```

11.2.1.5 Power set functor

The HOL-Omega type function $\lambda\alpha.\ \alpha \to$ `bool` takes a type `'a` to the type `'a -> bool`, the type of predicates on the type `'a`. Since a predicate is the same as a subset of the values of the type, this type is the same as the type of sets of elements from the type `'a`. Each of these sets is a subset of the complete set of values of the type `'a`, so this type function $\lambda\alpha.\ \alpha \to$ `bool` is actually the power set operation on types.

In HOL, the image function on sets is defined as

```
- pred_setTheory.IMAGE_DEF;                                             13
> val it =
    |- !(f :'a -> 'b) (s :'a -> bool). IMAGE f s = {f x | x IN s}
     : thm
- type_of ``IMAGE``;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it =
    ``:('a -> 'b) -> ('a -> bool) -> 'b -> bool``
     : hol_type
```

When wrapped with two type abstractions as `\:'a 'b. IMAGE`, it becomes a functor of type ($\lambda\alpha.\ \alpha \rightarrow$ `bool`) functor. This functor maps each object `'a` to `'a -> bool`, and each arrow `f : 'a -> 'b` to the arrow `IMAGE f : ('a -> bool) -> ('b -> bool)`.

To prove this is a functor, we first prove some theorems about `IMAGE` applied to identity arrows and composition, making use of preproven theorems in the theory `pred_set`.

```
- val I_EQ = store_thm                                              14
  ("I_EQ",
   ``I = \x:'a. x``,
   SIMP_TAC combin_ss [FUN_EQ_THM]
  );
> val I_EQ = |- I = (\x. x) : thm

- pred_setTheory.IMAGE_ID;
> val it =
    |- !s. IMAGE (\x. x) s = s
     : thm
- val IMAGE_I = store_thm
  ("IMAGE_I",
   ``IMAGE (I:'a -> 'a) = I``,
   SIMP_TAC combin_ss [FUN_EQ_THM,I_EQ,pred_setTheory.IMAGE_ID]
  );
> val IMAGE_I = |- IMAGE I = I : thm

- pred_setTheory.IMAGE_COMPOSE;
> val it =
    |- !f g s. IMAGE (f o g) s = IMAGE f (IMAGE g s)
     : thm
- val IMAGE_o = store_thm
  ("IMAGE_o",
   ``!(f :'a -> 'b) (g :'b -> 'c).
        IMAGE (g o f) = IMAGE g o IMAGE f``,
   SIMP_TAC combin_ss [FUN_EQ_THM,pred_setTheory.IMAGE_COMPOSE]
  );
> val IMAGE_o =
    |- !f g. IMAGE (g o f) = IMAGE g o IMAGE f
     : thm
```

Now we can prove that `\:'a 'b.IMAGE` is a functor of type `(\'a.'a -> bool)functor`.

```
- val powerset_functor = store_thm                                  15
  ("powerset_functor",
   ``functor ((\:'a 'b. IMAGE) : (\'a. 'a -> bool) functor)``,
   SIMP_TAC bool_ss [functor_def,IMAGE_I,IMAGE_o]
  );
> val powerset_functor =
    |- functor (\:'a 'b. IMAGE) :
  thm
```

11.2.1.6 List of lists map functor

We earlier saw how the MAP function is made into a functor. Similarly, we can compose the MAP function with itself, to create a function that maps a function across a list of lists. The term `MAP o MAP` has type `('a -> 'b) -> 'a list list -> 'b list list`, so the term `\:'a 'b. MAP o MAP` has type `!'a 'b. ('a -> 'b) -> 'a list list -> 'b list list`, which is the same as the type `(list o list) functor`. Notice how neatly the composition of the MAP function with itself matches the composition of the type operator `list` with itself in the type of the functor `\:'a 'b. MAP o MAP`. In fact, this is an example of a more general principle, which we shall explore next.

11.2.2 Composition of functors

Inspired by the above, we can prove that for any two functors `F'` and `G`, the functional composition of the arrow maps of the two functors may itself be made into a functor.

```
- show_types := true;                                                    16
> val it = () : unit

- val functor_o = store_thm
  ("functor_o",
   ``!(F': 'F functor) (G: 'G functor).
      functor F' /\ functor G ==>
      functor (\:'a 'b. G o F')``,
   SIMP_TAC combin_ss [functor_def]
  );
> val functor_o =
    |- !(F' :('F :ty => ty) functor) (G :('G :ty => ty) functor).
         functor F' /\ functor G ==>
         (functor
            (\:'a 'b.
               ((G [:'a 'F, 'b 'F:] o F' [:'a, 'b:])
                  :('a -> 'b) -> 'a 'F 'G -> 'b 'F 'G)) :bool)
      : thm
```

Note that here the two functors have type arguments applied to each, helpfully inserted by the parser, so as to transform the functors from values of universal type to values of functional type. Then they can be composed using the normal functional composition operator `o`, with the result then being wrapped up inside the type abstractions `\:'a 'b` to make a functor. This approach is entirely valid, and yet there is a simpler approach, where we define the composition of two functors directly.

Given two functors F and G, the composition of the two functors is defined as a new functor, whose map on objects is the composition of the object maps of the two functors, and whose map on arrows is the composition of the two arrow maps of the two functors. This can be easily defined in HOL-Omegaas follows.

```
- val oo_def = Define                                                       17
    `$oo (G: 'G functor) (F': 'F functor) = \:'a 'b. G o F' [:'a,'b:]`;
Definition has been stored under "oo_def"
> val oo_def =
    |- !(G :('G :ty => ty) functor) (F' :('F :ty => ty) functor).
         oo G F' = (\:'a 'b. G [:'a 'F, 'b 'F:] o F' [:'a, 'b:])
     : thm
```

Again, the parser has helpfully inserted the needed type arguments. We can improve the presentation of this functor composition operation by making `oo` an infix operator, and by overloading the infix `o` operator to refer to `oo`:

```
- add_infix("oo", 800, HOLgrammars.RIGHT);                                 18
> val it = () : unit
- overload_on ("o", ``$oo : 'G functor -> 'F functor -> ('G o 'F) functor``);
> val it = () : unit
```

Now the definition of function composition will display using the infix `o` operator.

```
- oo_def;                                                                   19
> val it =
    |- !(G :('G :ty => ty) functor) (F' :('F :ty => ty) functor).
         ((G o F') :('G o 'F) functor) =
         (\:'a 'b.
            ((G [:'a 'F, 'b 'F:] o F' [:'a, 'b:])
               :('a -> 'b) -> 'a 'F 'G -> 'b 'F 'G))
     : thm
```

This shows that the value returned by `G o F'` has a functor type. Nevertheless, to prove that this value truly is a functor, we must prove that it satisfies the conditions of the `functor` predicate.

```
- val functor_oo = store_thm                                                20
  ("functor_oo",
   ``!(F': 'F functor) (G: 'G functor).
      functor F' /\ functor G ==>
      functor (G o F')``,
   SIMP_TAC combin_ss [functor_def,oo_def]
  );
> val functor_oo =
    |- !(F' :('F :ty => ty) functor) (G :('G :ty => ty) functor).
         functor F' /\ functor G ==>
         (functor ((G o F') :('G o 'F) functor) :bool)
     : thm
```

We can immediately derive that `(\:'a 'b. MAP) o (\:'a 'b. MAP)` is a functor, as a simple corollary of this theorem.

```
- show_types := false;                                                21
> val it = () : unit

- val map_oo_map_functor = save_thm
  ("map_oo_map_functor",
   MATCH_MP functor_oo (CONJ map_functor map_functor)
  );
> val map_oo_map_functor =
    |- functor ((\:'a 'b. MAP) o (\:'a 'b. MAP))
     : thm
```

Similarly, we can show that `\:'a 'b. MAP o MAP` is a functor.

```
- val functor_o = store_thm                                           22
  ("functor_o",
   ``!(F': 'F functor) (G: 'G functor).
      functor F' /\ functor G ==>
      functor (\:'a 'b. G o F')``,
   SIMP_TAC combin_ss [functor_def]
  );
> val functor_o =
    |- !F' G. functor F' /\ functor G ==> functor (\:'a 'b. G o F')
     : thm

- val map_o_map_functor = save_thm
  ("map_o_map_functor",
   (TY_BETA_RULE o MATCH_MP functor_o) (CONJ map_functor map_functor)
  );
> val map_o_map_functor =
    |- functor (\:'a 'b. MAP o MAP)
     : thm
```

The use of `TY_BETA_RULE` above simplifies the theorem. This can be seen if we turn on the display of types.

```
- show_types := true;                                                 23
> val it = () : unit

- map_o_map_functor;
> val it =
    |- (functor
          (\:'a 'b.
             (((MAP :('a list -> 'b list) -> 'a list list -> 'b list list) o
               (MAP :('a -> 'b) -> 'a list -> 'b list))
                :('a -> 'b) -> 'a list list -> 'b list list)) :bool)
     : thm
```

whereas if we omit the use of `TY_BETA_RULE`, we obtain

```
- MATCH_MP functor_o (CONJ map_functor map_functor);                    24
> val it =
    |- (functor
          (\:'a 'b.
             (((\:'a 'b. (MAP :('a -> 'b) -> 'a list -> 'b list))
                 [:'a list, 'b list:] o
               (\:'a 'b. (MAP :('a -> 'b) -> 'a list -> 'b list))
                 [:'a, 'b:])
                :('a -> 'b) -> 'a list list -> 'b list list)) :bool)
     : thm
```

Logically the two are equivalent, but the second version has unreduced type beta-redexes, whereas the first version is reduced, and so simpler and easier to read. In essence, this shows how defining an operator like `oo` to directly compose two functors is really simpler than just relying on functional composition to accomplish the task.

11.3 Natural Transformations

Alongside functors, natural transformations are the other major idea of category theory. A natural transformation is essentially a mapping from one functor to another, that preserves the structure of the image of the first functor into the structure of the image of the second functor. A natural transformation $\eta : F \dot{\to} G$ from the functor F: `'F` functor to the functor G: `'G` functor is actually a function from objects to arrows; it maps an object α to an arrow $\eta[\alpha]\ :\ \alpha$ `'F` $\to$ α `'G`. In addition, the function η must preserve the structure of the image of F into the image of G; this means that it must make the following diagram commute for all possible arrows $h : \alpha \to \beta$:

$$\begin{tikzcd}
\alpha\ \texttt{'F} \arrow[r, "\eta[\alpha]"] \arrow[d, "F(h)"'] & \alpha\ \texttt{'G} \arrow[d, "G(h)"] \\
\beta\ \texttt{'F} \arrow[r, "\eta[\beta]"] & \beta\ \texttt{'G}
\end{tikzcd}$$

The first step is to define a type abbreviation for the type of natural transformations, based on the type operators associated with the two functors on which it is based.

```
- val _ = type_abbrev ("nattransf", Type `: \'F 'G. !'a. 'a 'F -> 'a 'G`); 25

- ``: !'a. 'a list -> 'a set``;
> val it =
    ``:(list, \'a. 'a -> bool) nattransf``
     : hol_type
```

Now we can define a natural transformation as such a function ϕ, along with two associated functors F and G, which makes the above diagram commute. This is the same as saying that the arrow $\phi[\alpha]$ composed with $G\ h$ is the same as the arrow $F\ h$ composed with $\phi[\beta]$, for all arrows $h : \alpha \to \beta$.

```
- val nattransf_def = new_definition("nattransf_def", Term                    26
   ‘nattransf (phi : ('F,'G) nattransf)
              ( F' : 'F functor )
              ( G  : 'G functor ) =
         !:'a 'b. !(h:'a -> 'b).
                G h o phi = phi o F' h‘);
> val nattransf_def =
    |- !(phi :('F :ty => ty, 'G :ty => ty) nattransf) (F' :'F functor)
          (G :'G functor).
         nattransf phi F' G <=>
         !:'a 'b.
           !(h :'a -> 'b).
             ((G [:'a, 'b:] h o phi [:'a:]) :'a 'F -> 'b 'G) =
             ((phi [:'b:] o F' [:'a, 'b:] h) :'a 'F -> 'b 'G)
     : thm
```

Here we can see how the parser has inserted type arguments to `phi`, `F'`, and `G`, where the specific types in the type arguments are determined through type inference. In particular, `phi` is given different type arguments for its two instances in the body, and so this definition could not have been expressed within classic HOL, in which a variable can only have one single type.

11.3.1 Examples of Natural Transformations

11.3.1.1 Identity natural transformation

One simple example of a natural transformation is the identity natural transformation. This natural transformation maps each functor to itself. Therefore, there is an instance of the identity natural transformation for each different functor F: `'F` functor. As a function, this natural transformation maps each object `'a` to the identity arrow on `'a 'F`, that is, `I:'a 'F -> 'a 'F`.

```
- val identity_nattransf = store_thm                                          27
  ("identity_nattransf",
   ‘‘!:'F. !F' : 'F functor.
        nattransf (\:'a. I) F' F'‘‘,
   SIMP_TAC combin_ss [nattransf_def]
  );
> val identity_nattransf =
    |- !:'F :ty => ty.
         !(F' :'F functor). nattransf (\:'a. (I :'a 'F -> 'a 'F)) F' F'
     : thm
```

11.3.1.2 Reverse natural transformation

Another simple natural transformation is REVERSE, which reverses the elements of a list:

```
- listTheory.REVERSE_DEF;                                                    28
> val it =
    |- (REVERSE ([] :'a list) = ([] :'a list)) /\
       !(h :'a) (t :'a list). REVERSE (h::t) = ((REVERSE t ++ [h]) :'a list)
     : thm

- val nattransf_REVERSE = store_thm
  ("nattransf_REVERSE",
   ``nattransf (\:'a. REVERSE)
               (\:'a 'b. MAP)
               (\:'a 'b. MAP)``,
   SIMP_TAC bool_ss [nattransf_def]
   THEN REPEAT STRIP_TAC
   THEN REWRITE_TAC[FUN_EQ_THM]
   THEN Induct
   THEN FULL_SIMP_TAC list_ss []
  );
> val nattransf_REVERSE =
    |- nattransf (\:'a. (REVERSE :'a list -> 'a list))
         (\:'a 'b. (MAP :('a -> 'b) -> 'a list -> 'b list))
         (\:'a 'b. (MAP :('a -> 'b) -> 'a list -> 'b list))
     : thm
```

11.3.1.3 Initial prefixes natural transformation

A more interesting natural transformation is the function INITS, which takes a list l and returns a list of all prefixes of l. This natural transformation is from the functor $\lambda\alpha\ \beta.$ MAP to the functor $\lambda\alpha\ \beta.$(MAP o MAP). INITS is defined recursively as follows.

```
- val INITS_def = Define                                                     29
  `(INITS [] = [[]]) /\
   (INITS ((x:'a)::xs) = [] :: MAP (CONS x) (INITS xs))`;
Definition has been stored under "INITS_def"
> val INITS_def =
    |- (INITS ([] :'a list) = [([] :'a list)]) /\
       !(x :'a) (xs :'a list).
         INITS (x::xs) = ([] :'a list)::MAP (CONS x) (INITS xs)
     : thm
```

To prove that INITS is a natural transformation, we need to first prove a couple of lemmas. The first lemma shows how MAP and CONS commute. The second lemma describes how MAP and INITS commute. Finally, the second lemma implies that INITS is a natural transformation. Each of these proofs are by induction on lists, and then uses the simplifier with the lists simplification set to finish off the different cases of the induction.

```
- val MAP_o_CONS = store_thm                                          30
  ("MAP_o_CONS",
   ``!(f:'a -> 'b) x. MAP f o CONS x = CONS (f x) o MAP f``,
   REPEAT GEN_TAC
   THEN REWRITE_TAC[FUN_EQ_THM]
   THEN Induct
   THEN ASM_SIMP_TAC list_ss []
  );
> val MAP_o_CONS =
    |- !(f :'a -> 'b) (x :'a).
         ((MAP f o CONS x) :'a list -> 'b list) =
         ((CONS (f x) o MAP f) :'a list -> 'b list)
     : thm

- val MAP_INITS = store_thm
  ("MAP_INITS",
   ``!l f:'a -> 'b. (MAP o MAP) f (INITS l) = INITS (MAP f l)``,
   Induct
   THEN FULL_SIMP_TAC list_ss [INITS_def,rich_listTheory.MAP_MAP_o,MAP_o_CONS]
   THEN ASM_REWRITE_TAC [GSYM rich_listTheory.MAP_MAP_o]
  );
> val MAP_INITS =
    |- !(l :'a list) (f :'a -> 'b).
         (((MAP :('a list -> 'b list) -> 'a list list -> 'b list list) o
           (MAP :('a -> 'b) -> 'a list -> 'b list)) f (INITS l) :
            'b list list) =
         INITS (MAP f l) :
  thm

- val nattransf_INITS = store_thm
  ("nattransf_INITS",
   ``nattransf (\:'a. INITS)
               (\:'a 'b. MAP)
               (\:'a 'b. MAP o MAP)``,
   SIMP_TAC bool_ss [nattransf_def]
   THEN REPEAT STRIP_TAC
   THEN REWRITE_TAC[FUN_EQ_THM]
   THEN Induct
   THEN FULL_SIMP_TAC list_ss [MAP_INITS]
  );
> val nattransf_INITS =
    |- (nattransf (\:'a. (INITS :'a list -> 'a list list))
          (\:'a 'b. (MAP :('a -> 'b) -> 'a list -> 'b list))
          (\:'a 'b.
             (((MAP :('a list -> 'b list) -> 'a list list -> 'b list list) o
               (MAP :('a -> 'b) -> 'a list -> 'b list))
                :('a -> 'b) -> 'a list list -> 'b list list)) :bool)
     : thm
```

11.3.2 Vertical composition of natural transformations

Two natural transformations may be composed together in either of two ways, vertically or horizontally. We shall examine horizontal composition in a moment, but for now, here is the definition of vertical composition of natural transformations.

```
- val ooo_def = Define `$ooo (phi2: ('G,'H)nattransf)                    31
                             (phi1: ('F,'G)nattransf) =
                      \:'a. phi2 o (phi1[:'a:])`;
Definition has been stored under "ooo_def"
> val ooo_def =
    |- !(phi2 :('G :ty => ty, 'H :ty => ty) nattransf)
          (phi1 :('F :ty => ty, 'G) nattransf).
         ooo phi2 phi1 =
         (\:'a. ((phi2 [:'a:] o phi1 [:'a:]) :'a 'F -> 'a 'H))
     : thm
```

We can improve the presentation of this composition operation by making it an infix operator and by overloading the o symbol to refer to this operation.

```
- val _ = add_infix("ooo", 800, HOLgrammars.RIGHT);                       32
- val _ = overload_on ("o", Term `$ooo : ('G,'H)nattransf ->
                                        ('F,'G)nattransf ->
                                        ('F,'H)nattransf`);
- ooo_def;
> val it =
    |- !(phi2 :('G :ty => ty, 'H :ty => ty) nattransf)
          (phi1 :('F :ty => ty, 'G) nattransf).
         ((phi2 o phi1) :('F, 'H) nattransf) =
         (\:'a. ((phi2 [:'a:] o phi1 [:'a:]) :'a 'F -> 'a 'H))
     : thm
```

Now that we can prove that the vertical composition of two natural transformations, as defined above, in fact yields a natural transformation.

```
- val nattransf_comp = store_thm                                          33
  ("nattransf_comp",
   ``nattransf (     phi1     : ('F,'G)nattransf) F' G /\
     nattransf (     phi2     : ('G,'H)nattransf) G  H ==>
     nattransf ((phi2 o phi1) : ('F,'H)nattransf) F' H``,
   SIMP_TAC bool_ss [nattransf_def,ooo_def]
   THEN REPEAT STRIP_TAC
   THEN ASM_REWRITE_TAC[o_ASSOC]
   THEN ASM_REWRITE_TAC[GSYM o_ASSOC]
  );
> val nattransf_comp =
    |- nattransf (phi1 :('F :ty => ty, 'G :ty => ty) nattransf)
         (F' :'F functor) (G :'G functor) /\
       nattransf (phi2 :('G, 'H :ty => ty) nattransf) G (H :'H functor) ==>
       nattransf ((phi2 o phi1) :('F, 'H) nattransf) F' H
     : thm
```

11.3.3 Horizontal composition of natural transformations

We can also define the horizontal composition of two natural transformations.

```
- val hcomp_def = Define ‘hcomp (phi2: ('F2,'G2)nattransf)                 34
                                (F2:'F2 functor)
                                (phi1: ('F1,'G1)nattransf) =
                           \:'a. phi2 o F2 (phi1[:'a:])‘;
Definition has been stored under "hcomp_def"
> val hcomp_def =
    |- !(phi2 :('F2 :ty => ty, 'G2 :ty => ty) nattransf) (F2 :'F2 functor)
          (phi1 :('F1 :ty => ty, 'G1 :ty => ty) nattransf).
         hcomp phi2 F2 phi1 =
         (\:'a.
            ((phi2 [:'a 'G1:] o F2 [:'a 'F1, 'a 'G1:] (phi1 [:'a:]))
               :'a 'F1 'F2 -> 'a 'G1 'G2))
      : thm
```

Unfortunately, the definition requires that we use one of the functors associated with the second natural transformation, so this operation cannot have the nice binary style of the vertical composition operator. We need to include the F2 functor argument in the definition of hcomp, and this makes it a 3-ary operation, not binary.

Having defined the horizontal composition of two natural transformations, we can prove that the composition is also a natural transformation itself.

```
- show_types := false;                                                     35
> val it = () : unit

- val nattransf_hcomp = store_thm
  ("nattransf_hcomp",
   ``nattransf (phi1 : ('F1,'G1) nattransf) F1 G1 /\
     nattransf (phi2 : ('F2,'G2) nattransf) F2 G2 /\
     functor F2 ==>
     nattransf (hcomp phi2 F2 phi1)
               (F2 o F1)
               (G2 o G1)``,
   SIMP_TAC bool_ss [nattransf_def,functor_def,hcomp_def,oo_def]
   THEN REPEAT STRIP_TAC
   THEN ASM_SIMP_TAC bool_ss [o_THM,o_ASSOC]
   THEN POP_ASSUM (fn th => REWRITE_TAC[GSYM o_ASSOC,GSYM th])
   THEN ASM_REWRITE_TAC[]
  );
> val nattransf_hcomp =
    |- nattransf phi1 F1 G1 /\ nattransf phi2 F2 G2 /\ functor F2 ==>
       nattransf (hcomp phi2 F2 phi1) (F2 o F1) (G2 o G1)
      : thm
```

Alternatively, we could have just as easily defined horizontal composition using the G2 functor instead of F2.

```
- val hcomp'_def = Define `hcomp' (phi2: ('F2,'G2)nattransf)                36
                                (G2:'G2 functor)
                                (phi1: ('F1,'G1)nattransf) =
                          \:'a. G2 phi1 o (phi2[:'a 'F1:])`;
Definition has been stored under "hcomp'_def"
> val hcomp'_def =
    |- !phi2 G2 phi1. hcomp' phi2 G2 phi1 = (\:'a. G2 phi1 o phi2)
     : thm
```

Using this different definition of the horizontal composition of two natural transformations, we can again prove that the composition is also a natural transformation.

```
- val nattransf_hcomp' = store_thm                                         37
  ("nattransf_hcomp'",
   ``nattransf (phi1 : ('F1,'G1) nattransf) F1 G1 /\
     nattransf (phi2 : ('F2,'G2) nattransf) F2 G2 /\
     functor F2 ==>
     nattransf (hcomp' phi2 G2 phi1)
               (F2 o F1)
               (G2 o G1)``,
   SIMP_TAC bool_ss [nattransf_def,functor_def,hcomp'_def,oo_def]
   THEN REPEAT STRIP_TAC
   THEN ASM_SIMP_TAC bool_ss [o_THM,o_ASSOC]
   THEN POP_ASSUM (fn th => REWRITE_TAC[GSYM o_ASSOC,GSYM th])
   THEN ASM_REWRITE_TAC[]
  );
> val nattransf_hcomp' =
    |- nattransf phi1 F1 G1 /\ nattransf phi2 F2 G2 /\ functor F2 ==>
       nattransf (hcomp' phi2 G2 phi1) (F2 o F1) (G2 o G1)
     : thm
```

In fact, the two definitions of horizontal composition are equivalent, given that phi2 is a natural transformation.

```
- val nattransf_hcomp_hcomp' = store_thm                                   38
  ("nattransf_hcomp_hcomp'",
   ``!(phi1 : ('F1,'G1) nattransf) (phi2 : ('F2,'G2) nattransf) F2 G2.
      nattransf phi2 F2 G2 ==>
      (hcomp phi2 F2 phi1 = hcomp' phi2 G2 phi1)``,
   SIMP_TAC bool_ss [nattransf_def,hcomp_def,hcomp'_def]
  );
> val nattransf_hcomp_hcomp' =
    |- !phi1 phi2 F2 G2.
         nattransf phi2 F2 G2 ==> (hcomp phi2 F2 phi1 = hcomp' phi2 G2 phi1)
     : thm
```

This is a result of the fact that two natural transformations commute, in the following fashion.

```
- val nattransf_commute = store_thm                                   39
  ("nattransf_commute",
   ``nattransf (phi1 : ('F1,'G1) nattransf) F1 G1 /\
     nattransf (phi2 : ('F2,'G2) nattransf) F2 G2  ==>
     (phi2 o F2 (phi1[:'a:]) = G2 phi1 o phi2)``,
   SIMP_TAC bool_ss [nattransf_def]
  );
> val nattransf_commute =
    |- nattransf phi1 F1 G1 /\ nattransf phi2 F2 G2 ==>
       (phi2 o F2 phi1 = G2 phi1 o phi2)
     : thm
```

11.3.4 Composition of natural transformations with functors

In addition to composing natural transformations with themselves, we can compose them with functors. There are two ways that a functor may be combined with a natural transformation, where either the functor is applied first and then the natural transformation, or the reverse. We define operations for both of these, positioning the functor on the right for the version where the functor is applied first, and on the left for the version where the functor is applied last.

To begin, here is the composition of a natural transformation with a functor, where the functor is applied first.

```
- show_types := true;                                                 40
> val it = () : unit

- val oof_def = Define `$oof (phi: ('F,'G) nattransf) (H': 'H functor) =
                      \:'a. phi [:'a 'H:]`;
Definition has been stored under "oof_def"
> val oof_def =
    |- !(phi :('F :ty => ty, 'G :ty => ty) nattransf)
          (H' :('H :ty => ty) functor). oof phi H' = (\:'a. phi [:'a 'H:])
     : thm
```

As before, we can improve the presentation by making this an infix operator and overloading the o symbol.

```
- val _ = add_infix("oof", 750, HOLgrammars.LEFT);                    41
- val _ = overload_on ("o", Term `$oof : ('F,'G) nattransf ->
                                        'H functor ->
                                        ('F o 'H,'G o 'H) nattransf`);

- (dest_const o fst o strip_comb) ``(phi:('F,'G)nattransf) o (H:'H functor)``;
> val it =
    ("oof",
     ``:('F :ty => ty, 'G :ty => ty) nattransf ->
        ('H :ty => ty) functor -> (!'a. 'a 'H 'F -> 'a 'H 'G)``)
     : string * hol_type
```

Now we can prove that this composition of a natural transformation with a functor is itself a natural transformation.

```
- show_types := false;                                                    42
> val it = () : unit

- val nattransf_functor_comp = store_thm
  ("nattransf_functor_comp",
   ``nattransf (phi : ('F,'G)nattransf) F' G /\
     functor (H : 'H functor) ==>
     nattransf (phi o H)     (* phi oof H *)
               ( F' o H)     (*  F' oo  H *)
               ( G  o H)``,  (*  G  oo  H *)
   SIMP_TAC combin_ss [nattransf_def,functor_def,oo_def,oof_def]
  );
> val nattransf_functor_comp =
    |- nattransf phi F' G /\ functor H ==>
       nattransf (phi o H) (F' o H) (G o H)
     : thm
```

Here is the other composition of a functor with a natural transformation, where the functor is applied last, as previously mentioned.

```
- show_types := true;                                                     43
> val it = () : unit

- val foo_def = Define `$foo (H: 'H functor) (phi: ('F,'G)nattransf) =
                        \:'a. H (phi[:'a:])`;
Definition has been stored under "foo_def"
> val foo_def =
    |- !(H :('H :ty => ty) functor)
          (phi :('F :ty => ty, 'G :ty => ty) nattransf).
         foo H phi = (\:'a. H [:'a 'F, 'a 'G:] (phi [:'a:]))
     : thm
```

Again, we can improve the presentation by declaring the operation as a binary infix, and overloading the o operator. Note that the HOL-Omega system can distinguish between compositions of natural transformations with themselves or with functors on the left or on the right, simply by looking at the types of the arguments to o.

```
- val _ = add_infix("foo", 750, HOLgrammars.LEFT);                        44
- val _ = overload_on ("o", Term `$foo : 'H functor ->
                                         ('F,'G) nattransf ->
                                         ('H o 'F,'H o 'G) nattransf`);
- (dest_const o fst o strip_comb) ``(H:'H functor) o (phi:('F,'G)nattransf)``;
> val it =
    ("foo",
     ``:('H :ty => ty) functor ->
        ('F :ty => ty, 'G :ty => ty) nattransf ->
        (!'a. 'a 'F 'H -> 'a 'G 'H)``)
     : string * hol_type
```

In this case also, the composition of a functor with a natural transformation, where the functor is applied last, is itself a natural transformation.

```
- show_types := false;                                                    45
> val it = () : unit

- val functor_nattransf_comp = store_thm
  ("functor_nattransf_comp",
   ``nattransf (phi : ('F,'G)nattransf) F' G  /\
     functor (H : 'H functor) ==>
     nattransf (H o phi)      (* H foo phi *)
               (H o F')       (* H oo F'   *)
               (H o G)``,     (* H oo G    *)
   SIMP_TAC combin_ss [nattransf_def,functor_def,oo_def,foo_def]
   THEN REPEAT STRIP_TAC
   THEN POP_ASSUM (fn th => REWRITE_TAC[GSYM th])
   THEN ASM_REWRITE_TAC[]
  );
> val functor_nattransf_comp =
    |- nattransf phi F' G /\ functor H ==>
       nattransf (H o phi) (H o F') (H o G)
     : thm
```

The above examples of manipulating functors and natural transformations are intended to show how easily and fluidly the HOL-Omega system supports such reasoning about this simple version of category theory. Generally the concepts are not hard to express, taking only brief phrases in the logic. Once the right definitions are made, the different concepts combine very neatly and smoothly, just as one would hope for a subject as beautiful as category theory.

11.4 Algebras and Initial Algebras

Algebras, also called *F*-algebras, are an important part of category theory. They have a direct application as abstract data types in computer programming, and thus have a beneficial effect on program clarity, modularity, and maintenance. Despite this utility, algebras arise as a beautifully elegant use of the idea of functors.

From this section on, the development of this chapter is taken almost directly from the book *Algebra of Programming* by Richard Bird and Oege de Moor, Prentice Hall, 1997. Some of the descriptions of the ideas presented are very similar to the text of Bird and de Moor, and they should be given full credit for these.

Given a functor F of type $'F$ functor, an *algebra* is defined as an arrow of type $\alpha\,'F \to \alpha$, where the object α is called the *carrier* of the algebra.

This definition is very abstract, so it may help to have an example. Consider the natural numbers, along with the zero value and the successor function. In the HOL logic these are represented as the type `num` and the constants `0:num` and `SUC:num -> num`. These two constants allow one to construct any value of the natural numbers.

This can be thought of as an algebra with one type and two constants, generated by a functor F where the map of F on objects is $\alpha\,'F = \texttt{unit + }\alpha$ and the map of F on arrows is $F'\ h = \texttt{I ++ }h$. Here the infix binary operator ++ is defined by the two cases $(f \texttt{ ++ } g)(\texttt{INL}\ a) = \texttt{INL}(f\ a)$ and $(f \texttt{ ++ } g)(\texttt{INR}\ b) = \texttt{INR}(g\ b)$.

Clearly F is a functor, as it maps the identity arrow $\texttt{I} = \lambda x{:}\alpha.\ x$ to the identity arrow $F'\ \texttt{I} = \lambda x{:}(\texttt{unit + }\alpha).\ x$, and it maps the composition $g \circ f$ to $F'(g \circ f) = \texttt{I ++ }(g \circ f) = (\texttt{I} \circ \texttt{I})\texttt{ ++ }(g \circ f) = (\texttt{I ++ }g) \circ (\texttt{I ++ }f) = F'\ g \circ F'\ f$.

Then the natural numbers could be described as an F-algebra by taking the carrier `num` and the arrow `nat_alg` of type `unit + num` $\rightarrow$ `num`, defined as

$$\begin{array}{lll} \texttt{Nat\_alg} = \lambda m.\ \textbf{case}\ m\ \textbf{of} & \texttt{INL ()} & \Rightarrow \texttt{0} \\ & \mid\ \texttt{INR}\ n & \Rightarrow \texttt{SUC}\ n\ . \end{array}$$

How does `Nat_alg` describe the natural numbers? This function encodes all of the information about both of the constants that construct natural numbers, `0:num` and `SUC:num -> num`. Depending on the input to `nat_alg`, we can select either of these two functions. So this combines all of the constructors for natural numbers in one function.

The notion of an algebra can be realized in HOL-Omega as a type abbreviation.

```
- val _ = type_abbrev ("algebra", Type `: \'F 'a. 'a 'F -> 'a`);      46
- ``:(\'a. unit + 'a) algebra``;
> val it =
    ``:\'a. unit + 'a -> 'a``
     : hol_type
```

We define the functor F above for natural numbers in HOL-Omega as `Nat_fun`.

```
- val Nat_fun = new_definition("Nat_fun", Term                        47
     `Nat_fun = (\:'a 'b. \h. I ++ h) : (\'a. unit + 'a) functor`);
> val Nat_fun =
    |- Nat_fun = (\:'a 'b. (\h. I ++ h))
     : thm
```

We can show that `Nat_fun` is a functor, and so suitable to form an algebra.

```
- val SUM_MAP_I = sumTheory.SUM_MAP_I;                                48
> val SUM_MAP_I = |- I ++ I = I : thm
- val SUM_MAP_o = functorTheory.SUM_o;
> val SUM_MAP_o =
    |- !f1 g1 f2 g2. g1 o f1 ++ g2 o f2 = (g1 ++ g2) o (f1 ++ f2)
     : thm

- val Nat_functor = store_thm
  ("Nat_functor",
   ``functor Nat_fun``,
   SIMP_TAC combin_ss [functor_def,Nat_fun,SUM_MAP_I,GSYM SUM_MAP_o]
  );
> val Nat_functor = |- functor Nat_fun : thm
```

We can specify the natural numbers as the algebra described above.

```
- val Nat_alg = new_definition("Nat_alg",                          49
    ``Nat_alg =
      (\m. case m of INL () => 0
                   | INR n => SUC n)
       : (\'a. unit + 'a, num) algebra``);
> val Nat_alg =
    |- Nat_alg = (\m. case m of INL () => 0 | INR n => SUC n)
     : thm
```

But the same functor F could in principle specify any F-algebra with one type, call it α, and two constants Z and S with the types $Z : \alpha$ and $S : \alpha \to \alpha$. For example this functor would also generate an algebra with the type `bool` and $Z =$ `F` and $S = \lambda t.$`T`, with the type `num` with $Z =$ `0` and $S = \lambda n.((n+1)$ `MOD 7`$)$, or with the type `unit list` and $Z =$ `[]` and $S =$ `CONS()`.

```
- val Bool_alg = new_definition("Bool_alg",                        50
    ``Bool_alg =
      (\b. case b of INL () => F
                   | INR b' => T)
       : (\'a. unit + 'a, bool) algebra``);
> val Bool_alg =
    |- Bool_alg = (\b. case b of INL () => F | INR b' => T)
     : thm
```

So a single functor can give rise to many algebras with the same functor.

Given two algebras $f : \alpha\ 'F \to \alpha$ and $g : \beta\ 'F \to \beta$ based on the same functor $F' : 'F$ functor, an *F-homomorphism* is a mapping $h : \alpha \to \beta$ such that $h \circ f = g \circ F'\ h$.

```
- val homo_def = new_definition("homo_def", Term                   51
   `homo (F': 'F functor) f g (h:'a -> 'b) = (h o f = g o F' h)`);
> val homo_def =
    |- !F' f g h. homo F' f g h <=> (h o f = g o F' h)
     : thm
```

There is an F-homomorphism from `Nat_alg` to `Bool_alg`.

```
- val Nat_Bool_homo = store_thm                                    52
  ("Nat_Bool_homo",
   ``homo (\:'a 'b. \h. I ++ h) Nat_alg Bool_alg
          (\n. n <> 0)``,
   RW_TAC bool_ss [homo_def,Nat_alg,Bool_alg]
   THEN CONV_TAC FUN_EQ_CONV
   THEN BETA_TY_TAC
   THEN Cases
   THEN RW_TAC std_ss [oneTheory.one_case_rw]
  );
> val Nat_Bool_homo =
    |- homo (\:'a 'b. (\h. I ++ h)) Nat_alg Bool_alg (\n. n <> 0)
     : thm
```

However, there are no F-homomorphisms from Bool_alg to Nat_alg.

```
- val no_Bool_Nat_homo = store_thm                                  53
  ("no_Bool_Nat_homo",
   ``!phi. ~(homo (\:'a 'b. \h. I ++ h) Bool_alg Nat_alg phi)``,
   RW_TAC bool_ss [homo_def,Nat_alg,Bool_alg]
   THEN DISCH_THEN (MP_TAC o CONV_RULE FUN_EQ_CONV)
   THEN DISCH_THEN (MP_TAC o SPEC ``INR T : unit + bool``)
   THEN RW_TAC arith_ss []
  );
> val no_Bool_Nat_homo =
    |- !phi. ~homo (\:'a 'b. (\h. I ++ h)) Bool_alg Nat_alg phi
     : thm
```

The identity arrow is an F-homomorphism for any algebra to itself.

```
- val identity_homo = store_thm                                     54
  ("identity_homo",
   ``!(F':'F functor) f.
       functor F' ==>
       homo F' f f (I:'a -> 'a)``,
   SIMP_TAC combin_ss [homo_def,functor_def]
  );
> val identity_homo =
    |- !F' f. functor F' ==> homo F' f f I
     : thm
```

The composition of two F-homomorphisms is also an F-homomorphism.

```
- val homo_comp = store_thm                                         55
  ("homo_comp",
   ``!(F':'F functor) f1 f3 (h1:'a -> 'b) (h2:'b -> 'c).
       (?f2. homo F' f1 f2 h1 /\ homo F' f2 f3 h2) /\
       functor F' ==>
       homo F' f1 f3 (h2 o h1)``,
   RW_TAC bool_ss [homo_def,functor_def]
   THEN ASM_REWRITE_TAC[GSYM o_ASSOC]
   THEN ASM_REWRITE_TAC[o_ASSOC]
  );
> val homo_comp =
    |- !F' f1 f3 h1 h2.
         (?f2. homo F' f1 f2 h1 /\ homo F' f2 f3 h2) /\ functor F' ==>
         homo F' f1 f3 (h2 o h1)
     : thm
```

With all of these possible algebras generated by one functor, the question naturally arises, are any of these particularly superior to the others? It turns out there are some which stand out as more completely representing the functor.

Since there is an identity F-homomorphism from each algebra to itself, and since each composition of two F-homomorphisms is another F-homomorphism, the set of all algebras of a given functor F form a category **Alg**(F), where the objects are F-algebras and the arrows are F-homomorphisms. Then an algebra f is an *initial algebra* if it is an initial object in that category. That means that for every algebra g of that functor, there is exactly one F-homomorphism from the initial algebra f to the other algebra g.

```
- val ialg_def = new_definition("ialg_def", Term                        56
   `ialg (  F'  : 'F functor)
         (alpha : ('F,'t)algebra) =
      !:'a. !(f : ('F,'a)algebra). ?!h. homo F' alpha f h`);
> val ialg_def =
    |- !F' alpha. ialg F' alpha <=> !:'a. !f. ?!h. homo F' alpha f h
     : thm
```

The idea of an initial algebra is surprisingly powerful for such a terse definition. If α is an initial algebra, then this implies a recursion principle for proofs by recursion, and also a function definition principle, so that new recursive functions may be defined according to the constructors of the type. The definition of `ialg` above is a very condensed representation of all this information, which would have been impossible without the ability in HOL-Omega to quantify over type variables, as in `!:'a. ...` above.

The algebra `Nat_alg` is an example of such an initial algebra, which we prove next.

```
- val SIMP_REC_THM = prim_recTheory.SIMP_REC_THM;                       57
> val SIMP_REC_THM =
    |- !x f.
         (SIMP_REC x f 0 = x) /\
         !m. SIMP_REC x f (SUC m) = f (SIMP_REC x f m)
     : thm

- val SIMP_REC_cata_lemma = store_thm
  ("SIMP_REC_cata_lemma",
  ``((h: num -> 'a) o Nat_alg = f o Nat_fun h)
    = (h = SIMP_REC (f(INL ())) (f o INR))``,
   SIMP_TAC std_ss [Nat_alg,Nat_fun,o_DEF,FUN_EQ_THM,
                    sumTheory.FORALL_SUM,oneTheory.one_case_rw]
   THEN EQ_TAC THEN STRIP_TAC
   THENL [ Induct, ALL_TAC ]
   THEN ASM_SIMP_TAC std_ss [SIMP_REC_THM,oneTheory.one]
  );
> val SIMP_REC_cata_lemma =
    |- (h o Nat_alg = f o Nat_fun h) <=>
       (h = SIMP_REC (f (INL ())) (f o INR))
     : thm

- val Nat_ialg = store_thm
  ("Nat_ialg",
  ``ialg Nat_fun Nat_alg``,
   SIMP_TAC bool_ss [ialg_def,homo_def,SIMP_REC_cata_lemma]
  );
> val Nat_ialg = |- ialg Nat_fun Nat_alg : thm
```

This is a deeper proof that the ones we have encountered previously, but its structure is typical for such proofs of initial algebras. Proving that there is exactly one homomorphism implies that we have to prove 1) that there exists such a homomorphism, and 2) the homomorphism is unique. To prove the existence, we need to exhibit a witness function, which in general may have to be recursive. Since the witness function needs to depend on the target algebra `f`, we must use a recursive combinator to create a recursive function on the fly, depending on `f`. In this case the predefined HOL constant `SIMP_REC` suffices for our needs, taking two arguments, the value of the recursive function at zero, and a function to produce the $n + 1$-th value given the n-th value of the recursive function. Note that both of these arguments for `SIMP_REC` make use of `f`. The proof divides into the two cases of whether the homomorphism property holds for the zero constructor or for the successor constructor, and each is solved by simplification.

For 2), the uniqueness of the homomorphism is expressed as for any two versions of the homomorphism, they must be equal, which by extensionality means that the two homomorphisms give the same answer for every argument. In general this requires an proof by induction on the arguments. In this case we appeal to the induction principle on natural numbers, and then specialize each of the hypotheses for the two versions of the homomorphism with either the value for the zero case or for the successor case, depending on whether we are proving the base case or the step case of the induction. As before, the proof is finished by simplification.

11.5 Catamorphisms

The existence of an initial F-algebra is a very powerful idea, implying for example that there is a recursion principle for proving properties. In addition, it implies that there is one and only one function that exists as a homomorphism from this initial algebra to any other algebra f of the same functor F. This unique function is called the *catamorphism* of f, which Bird and de Moor denote as $(\!| f |\!)$.

```
- val cata_def = new_definition("cata_def", Term                    58
   `cata (  F'  : 'F functor)
         (alpha : ('F,'t)algebra) (* initial object in category of algebras *)
         (  f   : ('F,'a)algebra) =
         @h. homo F' alpha f h`);
> val cata_def =
    |- !F' alpha f. cata F' alpha f = @h. homo F' alpha f h
     : thm
```

This definition of catamorphisms takes three arguments, a functor and two algebras of that functor. The first algebra is intended to be an initial algebra; if it is not, then the result will not have the desired properties. The definition uses the HOL-Omega choice operator `@` to choose some function `h` which is a homomorphism from the first algebra to the second. If several such homomorphisms exist, then this will select one of them (consistently). If no such homomorphisms exist, then this will just pick one function of the underlying type, `'t` → `'a`, and we will know nothing more about that function. But if in fact the first algebra `alpha` is an initial algebra, then exactly one such homomorphism will always exist, it will be chosen by the `@` operator, and the function returned by `cata` will in fact be the one and only homomorphism from `alpha` to `f`.

For example, the catamorphism from an initial algebra to itself is the identity function.

```
- val identity_cata = store_thm                                              59
  ("identity_cata",
  ``functor (F' : 'F functor) /\ ialg F' (alpha : ('F,'t)algebra) ==>
    (cata F' alpha alpha = I)``,
   RW_TAC bool_ss [functor_def,cata_def,ialg_def,EXISTS_UNIQUE_THM]
   THEN POP_ASSUM (STRIP_ASSUME_TAC o SPEC ``alpha: ('F,'t)algebra``
                                    o TY_SPEC ``:'t``)
   THEN SELECT_ELIM_TAC
   THEN CONJ_TAC
   THENL [ EXISTS_TAC ``h:'t -> 't``
           THEN FIRST_ASSUM ACCEPT_TAC,

           RW_TAC combin_ss [homo_def]
         ]
  );
> val identity_cata =
    |- functor F' /\ ialg F' alpha ==> (cata F' alpha alpha = I)
     : thm
```

If `alpha` is an initial algebra, then any catamorphism based on an initial algebra `alpha` is in fact a homomorphism.

```
- val homo_cata = store_thm                                                60
  ("homo_cata",
  ``ialg (F' : 'F functor) (alpha : ('F,'t)algebra) ==>
    !:'a. !f: ('F,'a)algebra.
        homo F' alpha (f: ('F,'a)algebra) (cata F' alpha f)``,
   RW_TAC bool_ss [homo_def,cata_def,ialg_def,EXISTS_UNIQUE_THM]
   THEN REPEAT STRIP_TAC
   THEN POP_ASSUM (STRIP_ASSUME_TAC o SPEC_ALL o TY_SPEC_ALL)
   THEN SELECT_ELIM_TAC
   THEN PROVE_TAC[]
  );
Meson search level: ..
> val homo_cata =
    |- ialg F' alpha ==> !:'a. !f. homo F' alpha f (cata F' alpha f)
     : thm
```

Alternatively, one can say that if h is the catamorphism of f, then h has the homomorphism property.

```
- val cata_property = store_thm                                            61
  ("cata_property",
  ``ialg (F' : 'F functor) (alpha : ('F,'t)algebra) ==>
    !:'a. !(f : ('F,'a)algebra) h.
        ((h = cata F' alpha f) = (h o alpha = f o F' h))``,
   REPEAT STRIP_TAC
   THEN FIRST_ASSUM (STRIP_ASSUME_TAC o SPEC_ALL o TY_SPEC_ALL
                    o REWRITE_RULE[ialg_def,EXISTS_UNIQUE_THM])
   THEN REWRITE_TAC [GSYM homo_def]
   THEN EQ_TAC
   THEN RW_TAC bool_ss [homo_cata]
  );
> val cata_property =
    |- ialg F' alpha ==>
       !:'a. !f h. (h = cata F' alpha f) <=> (h o alpha = f o F' h)
     : thm
```

Another way of saying this is that if h is a homomorphism from an initial algebra α to another algebra f, then h is the catamorphism of f.

```
- val eq_cata = store_thm                                                    62
  ("eq_cata",
  ``ialg (F' : 'F functor) (alpha : ('F,'t)algebra) /\
    homo F' alpha (f: ('F,'a)algebra) h ==>
    (cata F' alpha f = h)``,
   RW_TAC bool_ss [homo_def,cata_def,ialg_def,EXISTS_UNIQUE_THM]
   THEN FIRST_ASSUM (STRIP_ASSUME_TAC o SPEC_ALL o TY_SPEC ``:'a``)
   THEN SELECT_ELIM_TAC
   THEN CONJ_TAC
   THENL [ EXISTS_TAC ``h': 't -> 'a``
           THEN FIRST_ASSUM ACCEPT_TAC,

           ASM_SIMP_TAC bool_ss []
         ]
  );
> val eq_cata =
    |- ialg F' alpha /\ homo F' alpha f h ==> (cata F' alpha f = h)
     : thm
```

Interestingly, if α is an initial algebra, and f and g are algebras of the same functor as α with h being a homomorphism from f to g, then the composition of the catamorphism of f with h is the same arrow as the catamorphism of g.

```
- val cata_fusion = store_thm                                                63
  ("cata_fusion",
  ``ialg (F' : 'F functor) (alpha : ('F,'t)algebra) /\
    homo F' f g (h: 't -> 'a) /\ functor F' ==>
    (h o cata F' alpha f = cata F' alpha g)``,
   STRIP_TAC
   THEN CONV_TAC SYM_CONV
   THEN MATCH_MP_TAC eq_cata
   THEN ASM_REWRITE_TAC[]
   THEN MATCH_MP_TAC homo_comp
   THEN ASM_REWRITE_TAC[]
   THEN EXISTS_TAC ``f: ('F,'t)algebra``
   THEN ASM_SIMP_TAC bool_ss [homo_cata]
  );
> val cata_fusion =
    |- ialg F' alpha /\ homo F' f g h /\ functor F' ==>
       (h o cata F' alpha f = cata F' alpha g)
     : thm
```

For now, we will stop here for this chapter of the tutorial. This development is continued further in the file `aopScript.sml` in the directory `examples/HolOmega`. It culminates with the banana split theorem:

```
SPLIT_def:                                                            64
  |- SPLIT (f:'a -> 'b) (g:'a -> 'c) = \p. (f p, g p)

banana_split:
  |- ialg (phi:'t functor) (alpha : ('t,'a) algebra) /\ functor phi ==>
     (SPLIT (cata phi alpha (f : ('t,'b) algebra))
            (cata phi alpha (g : ('t,'c) algebra))
           = cata phi alpha (SPLIT (f o phi FST) (g o phi SND)))
```

The banana split theorem can be used to automatically synthesize a more efficient algorithm by combining two existing algorithms.

The interested reader is invited to trace through the development on his own.

Chapter 12

Example: Packages

This chapter explains existential types and packages, which are used for information hiding and modularity in proofs. These are very well described in chapter 24 of *Types and Programming Languages* by Benjamin C. Pierce (MIT Press, 2002), and this chapter will draw significantly from Professor Pierce's work. Packages are a new variety of term, and just like the new type abstraction terms, require a new variety of type to serve as their types. In particular, where a type abstraction term has a universal type, a package term has a new type called an *existential type*.

Packages are somewhat similar to objects in object-oriented languages like Java, in the sense that the internal details of how the object's data is represented are to some degree hidden from the users of the object. This is very useful in practice, since it allows the actual representation of the data within the object to be changed at some later time, perhaps for efficiency concerns, while not disturbing the object's appearance to external code that uses the object as a black box. Packages do not include all of the features or flexibility of Java objects, omitting for example inheritance of methods and dynamic dispatch. But they are a first step toward organizing the data structures of a system in an object-oriented way.

There is also some overlap in purpose between packages and the abstract data types described in chapter 10; both shield and abstract away some of the details of internal data structures, for the purposes of information hiding and modularity. The main difference between these two approaches is that abstract data types are actual new full-weight types introduced into the HOL-Omega logic whose properties are truly only partially determined, whereas packages are more light-weight data structures that may be constructed or deconstructed on the fly as first-class values, and whose internal details are fixed and real but intentionally obscured, just as an object's signature only reveals the general patterns of access to the object's internal data, not the precise shape of how it is actually laid out. Thus abstract data types are truly abstract; the internal representation is not simply unknown at present but in fact is completely unknowable. By contrast, the internal representation of a package is secret but determined and fixed. The particular representation might possibly disclose itself through occasions of the use of the package to compute values.

As mentioned earlier, the types of package terms are existential types. These types are distinct from any other types that we have described before; for example, there is no overlap between universal and existential types. Superficially, these two types look almost identical, except that where the universal types use a universal quantification symbol, existential types use an existential quantification symbol. However, despite these superficial parallels, universal and existential types are not really duals in the close sense that normal universal and existential quantification are duals of each other. In particular, the ways that packages are first created and then taken apart and used are more heavy-weight than the ways that type abstraction terms are created and then used. Not only is the syntax more cumbersome, the ideas behind existential types are also somewhat more difficult to grasp at first glance than those behind universal types. Therefore we take some more time in this chapter to ease their introduction.

The goals of this chapter are:

(i) To present how existential types are created and what they mean,

(ii) To show how packages, terms of existential type, can be created and used,

(iii) To show how packages can be applied in an object-oriented way.

To some extent packages and existential types have already been demonstrated in the Appetizers chapter 4. In this chapter we will examine them more closely, and illustrate some of their interesting aspects.

12.1 Existential Types

An existential type is written as $?\alpha.\sigma$ or $\exists\alpha.\sigma$, where the type variable α may appear freely within the type σ. Such occurrences of α are considered bound by the existential quantification. Therefore, although other type variables that are free within σ are also free type variables of $\exists\alpha.\sigma$, the type variable α is never a free type variable of $\exists\alpha.\sigma$.

Notably, α may be of any kind or rank, but σ must have a base kind, just as for universal types. The kind of $\exists\alpha.\sigma$ itself will be a base kind, where its rank is determined the same as for universal types: if α has rank r_α and σ has rank r_σ, then the rank of $\exists\alpha.\sigma$ will be the maximum of $r_\alpha + 1$ and r_σ.

To get an intuition as to the meaning of universal and existential types, a universal type $\forall\alpha.\sigma$ may be thought of as a type whose values are functions that map any type σ_α such that the kind of α is `:>=:` the kind of σ_α to a term value of the type $\sigma[\sigma_\alpha/\alpha]$. These are special functions in that 1) they map from types to term values, not from a value to a value, and 2) the type of the resulting value depends on the actual type σ_α that is input. Because of this, these functions are called *dependent functions*.

In comparison to this, the meaning of an existential type $\exists\alpha.\sigma$ may be thought of as a type whose values are pairs of a type and a term, e.g. (σ_α, t), where σ_α is a type such that the kind of α is `:>=:` the kind of σ_α, and t is a term with the type $\sigma[\sigma_\alpha/\alpha]$. These are special pairs in that 1) they join a type and a term in a pair, not two terms, and 2) the type of the second element of the pair t depends on the actual type σ_α that is the first element of the pair. Because of this, these pairs are called *dependent pairs*.

```
- set_trace "Unicode" 0;                                          1
> val it = () : unit
- new_theory "package";
<<HOL message: Created theory "package">>
> val it = () : unit
```

Syntactically, existential types look very much like universal types, except that the `!` symbol is replaced by `?` (or $\forall$ by $\exists$). Just as for universal types, in $\exists\alpha.\sigma$ the bound variable α has scope over the body, σ. The free type variables of an existential type are the free type variables of the body, minus the bound type variable.

```
- val ety1 = ``:?'a. 'a -> 'a``;                                   2
> val ety1 = ``:?'a. 'a -> 'a`` :
  hol_type

- val ety1_vars = type_vars ety1;
> val ety1_vars = [] : hol_type list

- val ety2 = ``:?'a. 'a -> 'b``;
> val ety2 = ``:?'a. 'a -> 'b`` :
  hol_type

- val ety2_vars = type_vars ety2;
> val ety2_vars = [``:'b``] : hol_type list
```

ML functions `is_exist_type`, `mk_exist_type`, `dest_exist_type`, etc. are provided to test, create, or take apart these types.

```
- is_exist_type ety2;                                             3
> val it = true : bool

- val ety2' = mk_exist_type(gamma, gamma --> beta);
> val ety2' = ``:?'c. 'c -> 'b`` :
  hol_type

- val check = eq_ty ety2 ety2';
> val check = true : bool

- val (bvar,body) = dest_exist_type ety2;
> val bvar = ``:'a`` : hol_type
  val body = ``:'a -> 'b`` : hol_type
```

Similarly, `list_mk_exist_type` and `strip_exist_type` create or take apart multiple instances of existential types.

```
- val ety3 = list_mk_exist_type ([alpha,gamma], alpha --> beta --> gamma); 4
> val ety3 =
    ``:?'a 'c. 'a -> 'b -> 'c``
     : hol_type

- val (ety3_bvars,ety3_body) = strip_exist_type ety3;
> val ety3_bvars =
    [``:'a``, ``:'c``] :
  hol_type list
  val ety3_body =
    ``:'a -> 'b -> 'c``
     : hol_type
```

12.2 Packages

Terms with existential types are called *packages*. With respect to such terms, there are two key issues: how are they created, and how are they used. Therefore there is a need for a new form to create such pairs, and another new form to take them apart. Remember that intuitively a value of an existential type (i.e., a package) is a special kind of pair of a type and a term, where the type of the term depends upon the type which is the first element of the pair.

In HOL-Omega, a package is constructed using the syntax

$$\mathbf{pack}(:\sigma, t)\ ,$$

where **pack** is a reserved keyword, σ is a type, and t is a term. Note the presence of a colon (:) before the type σ to indicate the presence of a type, rather than a term.

In general, this may not be enough information to determine the package uniquely. In such cases, it suffices to add a type annotation to the package syntax, as for example

$$\mathbf{pack}(:\sigma, t) : \exists\alpha.\sigma'\ ,$$

where the type annotation is an existential type $\exists\alpha.\sigma'$ such that the type of the body t is $\sigma'[\sigma/\alpha]$. This annotated version will always determine the package uniquely, so it is advisable to generally include the type annotation if there is any uncertainty.

The following examples are taken from Pierce's book, pages 364-365. The first two show how the same package text, made from the same type and term, can have two different existential types, depending on whether or not a type annotation is provided.

```
- val pkg1 = ``pack (:num, (5, \x:num. SUC x))``;                    5
> val pkg1 = ``pack (:num,(5,(\x. SUC x)))`` :
  term
- val pkg1_ty = type_of pkg1;
> val pkg1_ty =
    ``:?'x. 'x # ('x -> 'x)``
     : hol_type

- val pkg2 = ``pack (:num, (5, \x:num. SUC x)) : ?'x. 'x # ('x -> num)``;
> val pkg2 = ``pack (:num,(5,(\x. SUC x)))`` :
  term
- val pkg2_ty = type_of pkg2;
> val pkg2_ty =
    ``:?'x :(ty:1). 'x # ('x -> num)``
     : hol_type
```

The type of a package **pack**$(:\sigma, t)$ is always an existential type $\exists\alpha.\sigma'$ such that the type of the body t is $\sigma'[\sigma/\alpha]$. Remember the kind of α must be `:>=:` the kind of σ. In the above two cases, the two existential types are either $\exists$`'x.'x # ('x -> 'x)` or $\exists$`'x.'x # ('x -> num)`, but in both cases the type of the body is $\sigma'[\sigma/\alpha]$ which is `num # (num -> num)`. This shows that in general it is a good idea to provide the type annotation, even though it might not be always necessary.

Also, it is entirely possible to make different packages which have the exact same type. In fact, this is part of the point of having packages, because we wish to hide certain information, in particular the exact representation type which is the first element of the pair that is the package value. Here are two different packages with the same type.

```
- val pkg3 = ``pack (:num, 0) : ?'x. 'x``;                            6
> val pkg3 =
    ``pack (:num,(0 :num))`` :
  term
- val pkg3_ty = type_of pkg3;
> val pkg3_ty = ``:?'x. 'x`` :
  hol_type

- val pkg4 = ``pack (:bool, T) : ?'x. 'x``;
> val pkg4 = ``pack (:bool,T)`` : term
- val pkg4_ty = type_of pkg4;
> val pkg4_ty = ``:?'x. 'x`` :
  hol_type

- val check = eq_ty pkg3_ty pkg4_ty;
> val check = true : bool
```

Here is another example of two packages which are different internally, but have the same existential type. We will see an actual use for these packages in what follows.

```
- val pkg5 =                                                        7
    ``pack (:num, (0, \x:num. SUC x)) : ?'x. 'x # ('x -> num)``;
> val pkg5 =
    ``pack (:num,((0 :num),(\(x :num). SUC x)))``
     : term
- val pkg5_ty = type_of pkg5;
> val pkg5_ty =
    ``:?'x. 'x # ('x -> num)``
     : hol_type

- val pkg6 =
    ``pack (:bool, (T, \x:bool. 0)) : ?'x. 'x # ('x -> num)``;
> val pkg6 =
    ``pack (:bool,(T,(\(x :bool). (0 :num))))``
     : term
- val pkg6_ty = type_of pkg6;
> val pkg6_ty =
    ``:?'x. 'x # ('x -> num)``
     : hol_type

- val check = eq_ty pkg5_ty pkg6_ty;
> val check = true : bool
```

This has shown how we construct packages. Now we will see how we take them apart.

In HOL-Omega, packages are taken apart (deconstructed) via the special syntax

$$\textbf{let } (:\alpha, x) = p \textbf{ in } s\ .$$

Here α is a type variable, x is a term variable whose type may mention α, p is a term yielding a package, and s, the body of the **let**, is a term. The idea here is that the package p is opened up, and the pair that is inside the package is bound to the pair of α and x, and then α and x are usable within the body s. Both α and x are bound by this syntax. The scope of α is x and s, while the scope of x is s. We require that the types of the free variables of s do not mention the type variable α, and likewise that the type of s itself does not mention the type variable α. This is only sensible, as the scope of α is only the variable x and the body s, and α has no meaning outside that scope.

This syntax for deconstructing a package is distinguished from the normal **let** syntax for pairs, **let** $(x, y) = e$ **in** e', by the presence of the colon just after the left parenthesis.

Here is an example of deconstructing pkg5 from above.

```
- val unpkg5 = ``let (:'x, t:'x # ('x -> num)) = ^pkg5             8
                 in (SND t) (FST t)``;
> val unpkg5 =
    ``let (:'x,(t :'x # ('x -> num))) =
            (pack (:num,((0 :num),(\(x :num). SUC x))))
      in
        SND t (FST t)``
     : term
```

So now that we can write down how to take a package apart, what does it mean? The effect of deconstructing a package is given by the following HOL-Omega theorem:

$$\vdash (\textbf{let}\ (:\alpha, x) = \textbf{pack}(:\sigma, t)\ \textbf{in}\ s) \ = \ (\lambda{:}\alpha.\lambda x.s)\ [:\sigma:]\ t$$

The body of the **let**, s, is made into a function that expects first a type argument σ, which is bound to the formal type parameter α, and secondly a term argument t, which is bound to the formal term parameter x. Then the body s is executed in this context. This is called *package reduction*, by analogy with beta reduction: $(\lambda x.s)\ t = s[t/x]$.

To show the meaning of such uses of packages, we define a simple evaluation tool using the ML functions SIMP_CONV and srw_ss from the simplification library.

```
- fun eval ths tm = QCONV (SIMP_CONV (srw_ss()) ths) tm;                  9
> val eval = fn : thm list -> term -> thm

- val unpkg5_res = eval [] unpkg5;
<<HOL message: Initialising SRW simpset ... done>>
> val unpkg5_res =
    |- (let (:'x,(t :'x # ('x -> num))) =
              (pack (:num,((0 :num),(\(x :num). SUC x))))
        in
          SND t (FST t)) =
       (1 :
       num) : thm
```

The simplifier knows how to reduce such deconstructions of packages, and how to evaluate the SUC function as well, even though that was hidden within pkg5.

Here is another example, using the same **let** but with pkg6 from above.

```
- val unpkg6 = ``let (:'x, t:'x # ('x -> num)) = ^pkg6                    10
                 in (SND t) (FST t)``;
> val unpkg6 =
    ``let (:'x,(t :'x # ('x -> num))) =
            (pack (:bool,(T,(\(x :bool). (0 :num)))))
      in
        SND t (FST t)``
     : term

- val unpkg6_res = eval [] unpkg6;
> val unpkg6_res =
    |- (let (:'x,(t :'x # ('x -> num))) =
              (pack (:bool,(T,(\(x :bool). (0 :num)))))
        in
          SND t (FST t)) =
       (0 :
       num) : thm
```

Note that the resulting value is different (although the type is the same), because although the body of the **let** is the same, the package that is being unpacked is different.

12.3 Underlying Implementation of Packages

One can use the **pack** and **let** syntax as described above to construct and deconstruct packages. In the following, we describe the implementation of this syntax, which can be skipped by the casual reader without loss. But for those who are interested, the syntax as given so far for both these new forms to construct and deconstruct packages is actually syntactic sugar. Underneath, the actual syntax rests on two new fundamental term constants in the HOL-Omega logic, `PACK` and `UNPACK`. These are part of the theory `bool`, with the following primal types:

$$\begin{array}{rl} \texttt{PACK} : & \forall\psi{:}\kappa.\ \psi\ (\alpha : \kappa \Rightarrow \mathbf{ty}{:}1)\ \texttt{->}\ (\exists\phi{:}\kappa.\ \phi\ \alpha) \\ \texttt{UNPACK} : & (\forall\psi{:}\kappa.\ \psi\ (\alpha : \kappa \Rightarrow \mathbf{ty}{:}1)\ \texttt{->}\ (\beta{:}\mathbf{ty}{:}1))\ \texttt{->}\ (\exists\phi{:}\kappa.\ \phi\ \alpha)\ \texttt{->}\ \beta \end{array}$$

Note that the kind variable κ and the type variables α and β are free in the above types, which provide all the needed flexibility for instances of the constants `PACK` and `UNPACK` to use packages in every appropriate situation. In particular we use the free type variable α in combination with its argument, the type variable ϕ, to represent any possible type expression with free type variable ϕ, and thus any possible existential type $\exists\phi{:}\kappa.\ \phi\ \alpha$.

The surface syntax for packages given previously is actually translated by the parser into uses of `PACK` and `UNPACK`, as follows.

$$\begin{array}{lll} \mathbf{pack}({:}\sigma, t) & \mapsto & \texttt{PACK}\ [{:}\sigma{:}]\ t \\ \mathbf{let}\ ({:}\alpha, x) = p\ \mathbf{in}\ s & \mapsto & \texttt{UNPACK}\ (\lambda{:}\alpha.\lambda x.s)\ p \end{array}$$

12.3.1 Package Axioms

Using `PACK` and `UNPACK`, the reduction of packages is fundamentally expressed by the new HOL-Omega axiom `UNPACK_PACK_AX`:

$$\begin{array}{ll} \vdash & \forall{:}(\alpha : \kappa \Rightarrow \mathbf{ty}{:}1)\ (\beta : \mathbf{ty}{:}1)\ (\phi : \kappa). \\ & \quad \forall(f : \forall\psi{:}\kappa.\ \psi\ \alpha\ \texttt{->}\ \beta)\ (t : \phi\ \alpha). \\ & \quad\quad \texttt{UNPACK}\ f\ (\texttt{PACK}\ [{:}\,\phi\,{:}]\ t) = f\ [{:}\,\phi\,{:}]\ t \end{array} \qquad (\texttt{UNPACK\_PACK\_AX})$$

This axiom is included in the basic simplification set `bool_ss`, so almost every invocation of the simplifier will attempt this reduction of packages by higher-order rewriting with `UNPACK_PACK_AX`, along with `TY_BETA_CONV` and `BETA_CONV` to resolve $(\lambda{:}\alpha.\lambda x.s)\ [{:}\sigma{:}]\ t$.

HOL-Omega also includes the new axiom `PACK_ONTO_AX`:

$$\begin{array}{ll} \vdash & \forall{:}(\alpha : \kappa \Rightarrow \mathbf{ty}{:}1).\ \forall(p : \exists\psi{:}\kappa.\ \psi\ \alpha). \\ & \quad \exists{:}(\phi : \kappa).\ \exists(t : \phi\ \alpha). \\ & \quad\quad p = \texttt{PACK}\ [{:}\,\phi\,{:}]\ t \end{array} \qquad (\texttt{PACK\_ONTO\_AX})$$

`PACK_ONTO_AX` expresses the idea that every package was created using `PACK` from some type and term. Thus `PACK` is the sole constructor of values of existential type.

12.4 Example: Counters

To see how packages can be used to simulate the idea of an object, we consider a counter, as an object for which there are three methods; a `new` method to create a counter with an initialized count, a `get` method to obtain the counter's current value, and an `inc` method to increment the counter's value.

The first step is to create a record type to hold these three methods. The methods will need to be parameterized on the actual type of the counter's contents, since that data structure is what the object is hiding. Eventually we will form a package with an existential type, where the actual representation of the counter's data will be hidden.

When designing the record type, we need to consider that we are working in a logic that does not have all of the features of a sophisticated programming language. Accordingly, the `get` method needs to take that internal data structure as input in order to compute the count that is output. In Java, an object can update its private data fields, but in a functional logic like HOL-Omega, any change must be represented by making a new copy of the original object, with updated contents. Also, whereas in Java a new object can be constructed by a routine that establishes its initial value, here we need to use the `new` method to obtain that initial value, since it must be of the unknown and parameterized type.

The new record type can be introduced to the HOL-Omega logic by the following.

```
- val _ = Hol_datatype                                                    11
       `counter_recd1 =
                     <| new : 'a;
                        get : 'a -> num;
                        inc : 'a -> 'a
                      |>`;
<<HOL message: Defined type: "counter_recd1">>
- val counter_kind = kind_of ``:counter_recd1``;
> val counter_kind = ``::ty => ty`` : kind
```

This introduces a new type constant in the HOL-Omega logic named `counter_recd1`, along with term constants for constructing, accessing, and updating the fields of values of this record type. The type constant created has the arrow kind **ty** => **ty**, so `counter_recd1` is a type operator that expects a type argument of kind **ty**.

The aim now is to create packages as values of this abstract data type, as values of type $\exists\psi.\ \psi$ `counter_recd1`. One way is to simply use a natural number as the hidden data structure, so that the increment function is just the successor function `SUC`.

```
- val counterADT =                                                          12
       ‘‘pack ( :num,
                <| new := 1;
                   get := \i:num. i;
                   inc := \i:num. SUC i
                |> ) : ?’a. ’a counter_recd1‘‘;
> val counterADT =
    ‘‘pack
        (:num,
          <|new := (1 :num); get := (\(i :num). i);
            inc := (\(i :num). SUC i)|>)‘‘
     : term
- val counterADT_type = type_of counterADT;
> val counterADT_type =
    ‘‘:?’x. ’x counter_recd1‘‘
     : hol_type
```

This yields a value with the desired existential type.

Next we wish to test this package by using it. The following code creates a new counter, increments it, and then returns its value.

```
- val counter_ex1 =                                                         13
  ‘‘let (:’Counter,counter) = ^counterADT in
    counter.get (counter.inc counter.new)‘‘;
> val counter_ex1 =
    ‘‘let (:’Counter,(counter :’Counter counter_recd1)) =
            (pack
               (:num,
                 <|new := (1 :num); get := (\(i :num). i);
                   inc := (\(i :num). SUC i)|>))
      in
        counter.get (counter.inc counter.new)‘‘
     : term
```

We can test to see how this works by using the `eval` tool we defined earlier. This provides a theorem where this program is simplified by evaluation.

```
- val ex1_res = eval[] counter_ex1;                                         14
> val ex1_res =
    |- (let (:’Counter,(counter :’Counter counter_recd1)) =
              (pack
                 (:num,
                   <|new := (1 :num); get := (\(i :num). i);
                     inc := (\(i :num). SUC i)|>))
        in
          counter.get (counter.inc counter.new)) =
       (2 :
       num) : thm
```

Here is another example, where we define a routine to increment the counter three times, and then return the result.

```
- val counter_ex2 =                                                    15
  ``let (:'Counter,counter) = ^counterADT in
    let add3 = \c:'Counter. counter.inc (counter.inc (counter.inc c)) in
    counter.get (add3 counter.new)``;
> val counter_ex2 =
    ``let (:'Counter,(counter :'Counter counter_recd1)) =
            (pack
               (:num,
                 <|new := (1 :num); get := (\(i :num). i);
                   inc := (\(i :num). SUC i)|>))
      in
        let (add3 :'Counter -> 'Counter) (c :'Counter) =
              counter.inc (counter.inc (counter.inc c))
        in
          counter.get (add3 counter.new)``
     : term
```

We can use the eval tool as before to evaluate this expression. We need to supply LET_DEF, the definition of the constant LET which is used to create the syntactic sugar **let..in** form, so that it can be reduced as well. (For some reason this is not included in the automatic set of simplifications.)

```
- LET_DEF;                                                             16
> val it =
    |- (LET :('a -> 'b) -> 'a -> 'b) = (\(f :'a -> 'b) (x :'a). f x)
     : thm

- val ex2_res = eval[LET_DEF] counter_ex2;
> val ex2_res =
    |- (let (:'Counter,(counter :'Counter counter_recd1)) =
              (pack
                 (:num,
                   <|new := (1 :num); get := (\(i :num). i);
                     inc := (\(i :num). SUC i)|>))
        in
          let (add3 :'Counter -> 'Counter) (c :'Counter) =
                counter.inc (counter.inc (counter.inc c))
          in
            counter.get (add3 counter.new)) =
       (4 :
       num) : thm
```

12.5 Example: Scheduling Queues

To demonstrate the usefulness of packages, we now will develop a theory of scheduling queues for an operating system.

Consider an operating system in which there are multiple processes, some of which are from time to time suspended in an inactive state, while others are running on the available processors.

The processes which are suspended are remembered in some kind of a data structure. When a new process is created, or when a currently running process becomes suspended, it is added to this data structure. When a process finishes its task and ends, its processor is then assigned to work on one of the other processes in the data structure.

Exactly which process is chosen is a matter of policy. One simple choice is that it should be the oldest process residing in the data structure. This leads to a last-in-first-out policy, which is accomplished by using a queue for the data structure.

However, this is not the only legitimate choice. There may be processes with higher priority, or there may be some more subtle measure of utility which ranks different processes more suitable to be assigned to the new processor.

One could even use a first-in-first-out policy, which would be accomplished by using a stack for the data structure. While this would not possess certain desirable properties like fairness, that each suspended process will eventually run, it might be suitable for certain restricted applications.

Even for a given policy, there may be several possible implementations which may vary in their pragmatics. We may begin, say, with a very simple and clean implementation of queues, and later move to a more complex representation for faster performance.

What is interesting here is that these different policy choices can be modularized by simply making them part of the data structure. Each such policy choice would be implemented by a different data structure. As long as each data structure obeys certain general properties, we can swap any of a variety of data structures for the one used in the implementation of the scheduling algorithm. We can even delay the choice of which policy to use until late in the overall development, or even switch the policy on the fly during runtime, simply by changing the data structure, as long as each of the family of data structures used all obey the same general properties.

The rest of the software development can then rely on the data structure to obey these general properties, but cannot rely on any other special properties of any individual data structure or the policy it represents. This enforces a modularity that is a vital feature of good system design, where design choices are isolated and hidden from the rest of the system, so that later changes have a minimum ripple effect.

Packages are critical to performing this necessary information hiding. This example demonstrates how packages can be used to accomplish the hiding of the actual data structure used. This supports proper modularization, so that the software that uses the scheduling data structure is isolated from the details of its implementation.

To begin, we will consider data structures that collect a number of elements. The types of these data structures will be modeled as a type operator β of kind $\mathbf{ty} \Rightarrow \mathbf{ty}$ that maps an element of type α into a data structure of type $\alpha\ \beta$.

The data structures themselves may have many different definitions, but here we want to concentrate on their fundamental operations and general properties. As a beginning, we want the scheduling data structures to support the following operations:

emptyq	:	$\forall\beta.\ \beta\ \alpha$
insert	:	$\forall\beta.\ \beta \to \beta\ \alpha \to \beta\ \alpha$
remove	:	$\forall\beta.\ \beta\ \alpha \to \beta\ \#\ \beta\ \alpha$
count	:	$\forall\beta.\ \beta \to \beta\ \alpha \to \mathtt{num}$

Note that each operation is polymorphic in the element type β.

The meaning of the emptyq operation is to be the empty version of the queue. Likewise, the insert operation takes an element and a queue, and inserts the element into the queue, returning the new, increased queue. The remove operation takes a queue, selects some element of it, removing that element from the queue, and returns a pair of the selected element and the new, diminished queue. The count operation takes an element and a queue, and returns the number of times (possibly zero) that that element appears in the queue.

These operations can be assembled into a record of related operations that are meant to work together, as follows.

```
- val _ = Hol_datatype                                                    17
        `sched_q_opers = <| emptyq : !'b. 'b 'a;
                            insert : !'b. 'b -> 'b 'a -> 'b 'a;
                            remove : !'b. 'b 'a -> 'b # 'b 'a;
                            count  : !'b. 'b -> 'b 'a -> num  |>`;
<<HOL message: Defined type: "sched_q_opers">>
```

This datatype definition in the HOL-Omega logic not only creates the type `sched_q_opers`, but also introduces term constants so that we can index the fields of any record of this type using the familiar notation `rcd.emptyq`, `rcd.insert`, etc.

Next we specify the properties that we wish to be true of these operations to constitute a valid and proper scheduling queue. These should be broad enough to encompass all the possible implementations we might wish to use, but narrow enough to be a base on which to build the rest of the program that uses this data structure.

The properties we will choose for this example are the following.

1. The count of any item in the empty queue is zero.

2. For the queue resulting from the insertion of an element y into a queue q, the count of any element x should be the count of x in the original queue q plus one if $x = y$, but otherwise simply the count of x in q.

3. If a queue q has at least one element (that is, q is not empty), then the result of removing an element from q is a pair $(y,\ q')$ which has the following property. The count of any element x in the original queue q is equal to the count of x in the result of inserting y into q'.

These properties can be represented in the following definition.

```
- val is_scheduling_q_def = Define                                        18
     ‘is_scheduling_q (ops:’a sched_q_opers) =
        (!:’b. !(x:’b). ops.count x (ops.emptyq[:’b:]) = 0) /\
        (!:’b. !(q:’b ’a) (x:’b) (y:’b).
                  ops.count x (ops.insert y q) =
                    if x = y then ops.count x q + 1
                              else ops.count x q) /\
        (!:’b. !q:’b ’a.
                if (!x:’b. ops.count x q = 0) then T else
                  let (y, q’) = ops.remove q in
                  !x:’b. ops.count x q = ops.count x (ops.insert y q’))‘;
Definition has been stored under "is_scheduling_q_def"
```

The system will respond with the theorem of the definition, which we omit here.

This covers the properties of a scheduling data structure. We will represent the data structure in an object-oriented way, by forming a record that contains both the current value of the data structure, and also all of the operations that can be performed on it. First, we create the type of this record.

```
- val _ = Hol_datatype                                                    19
        ‘sched_q = <| this : ’b ’a;
                      ops  : ’a sched_q_opers  |>‘;
<<HOL message: Defined type: "sched_q">>
```

For our first scheduling queue implementation, we choose to use a simple list, where new elements are added to the list at the front, and when an element is removed it is taken from the back of the list. For simplicity, instead of a complex type representing processes, we choose to use just a simple natural number. We intend to build a record of scheduling operations as follows.

```
val reference_q_def = Define
   ‘reference_q =
      <| this := [] : num list;
         ops  := <| emptyq := \:’b. [] : ’b list;
                    insert := \:’b. \(x:’b) xs. CONS x xs;
                    remove := \:’b. \xs:’b list. (LAST xs, FRONT xs);
                    count  := \:’b. \(x:’b) xs. COUNT x xs
                  |>
       |>‘;
```

Fortunately, we have predefined operations in the `list` library for most of this. Both `[]` and `CONS` are commonly used, and `FRONT` and `LAST` are predefined in the `list` library. `LAST` returns the last element of a list, while `FRONT` returns all of the list except for the last element. Both of these are undefined if they are applied to an empty list. What is not available in the `HOL` libraries is the `COUNT` function, but this is easy enough to create.

```
- open listTheory;                                                     20
. . .
- val COUNT_DEF = Define
    `(COUNT (x:'a) [] = 0) /\
     (COUNT x (y::ys) = if x = y then COUNT x ys + 1
                                 else COUNT x ys)`;
Definition has been stored under "COUNT_def"
> val COUNT_DEF =
    |- (!(x :'a). COUNT x ([] :'a list) = (0 :num)) /\
       !(x :'a) (y :'a) (ys :'a list).
         COUNT x (y::ys) =
         if x = y then COUNT x ys + (1 :num) else COUNT x ys
     : thm
```

Then the following properties of `COUNT` are proven by straightforward means.

```
> val ALL_COUNT_ZERO =                                                 21
    |- !(xs :'a list).
         (!(x :'a). COUNT x xs = (0 :num)) <=> (xs = ([] :'a list))
     : thm

> val ALL_COUNT_ZERO_2 =
    |- !(xs :'a list) (ys :'a list).
         (!(x :'a). (COUNT x xs = (0 :num)) /\ (COUNT x ys = (0 :num))) <=>
         (xs = ([] :'a list)) /\ (ys = ([] :'a list))
     : thm

> val COUNT_FRONT =
    |- !(xs :'a list) (x :'a).
         xs <> ([] :'a list) ==>
         x <> LAST xs ==>
         (COUNT x (FRONT xs) = COUNT x xs)
     : thm

> val COUNT_LAST =
    |- !(xs :'a list).
         xs <> ([] :'a list) ==>
         (COUNT (LAST xs) xs = COUNT (LAST xs) (FRONT xs) + (1 :num))
     : thm

> val COUNT_APPEND =
    |- !(xs :'a list) (ys :'a list) (x :'a).
         COUNT x ((xs ++ ys) :'a list) = COUNT x xs + COUNT x ys
     : thm
```

```
> val COUNT_REVERSE =                                                  22
    |- !(xs :'a list) (x :'a). COUNT x (REVERSE xs) = COUNT x xs
     : thm
```

The tactics to prove each are provided in the `packageScript.sml` file in the directory `examples/HolOmega`, but not covered further here.

The definition of `COUNT` now allows us to create our reference implementation of scheduling queues, as a very simple and clean implementation which is not necessarily efficient, but for which the necessary properties should be easy to prove.

```
- val reference_q_def = Define                                        23
     ‘reference_q =
        <| this := [] : num list;
           ops  := <| emptyq := \:’b. [] : ’b list;
                      insert := \:’b. \(x:’b) xs. CONS x xs;
                      remove := \:’b. \xs:’b list. (LAST xs, FRONT xs);
                      count  := \:’b. \(x:’b) xs. COUNT x xs
                    |>
         |>‘;
Definition has been stored under "reference_q_def"
> val reference_q_def =
    |- (reference_q :(list, num) sched_q) =
       <|this := ([] :num list);
         ops :=
           <|emptyq := (\:’b. ([] :’b list));
             insert := (\:’b. (\(x :’b) (xs :’b list). x::xs));
             remove := (\:’b. (\(xs :’b list). (LAST xs,FRONT xs)));
             count := (\:’b. (\(x :’b) (xs :’b list). COUNT x xs))|> |>
     : thm
```

Now we can prove that the operations of this reference scheduling queue satisfies the properties required to be a scheduling queue.

```
- val reference_q_is_scheduling_q = store_thm(                        24
   "reference_q_is_scheduling_q",
   ‘‘is_scheduling_q reference_q.ops‘‘,
   SRW_TAC [ARITH_ss] [reference_q_def,is_scheduling_q_def,COUNT_DEF]
   THEN REWRITE_TAC [ALL_COUNT_ZERO]
   THEN STRIP_ASSUME_TAC (ISPEC ‘‘q:’b list‘‘ list_CASES)
   THEN ASM_REWRITE_TAC [NOT_CONS_NIL]
   THEN GEN_TAC
   THEN SIMP_TAC list_ss [GSYM COUNT_LAST]
   THEN COND_CASES_TAC
   THEN SRW_TAC [] [COUNT_FRONT]
  );
> val reference_q_is_scheduling_q =
    |- is_scheduling_q (reference_q :(list, num) sched_q).ops
     : thm
```

Our intention here is to eventually show that this is one possible implementation of scheduling queues, but in the process to hide the actual data structure being used. In this case the data structure is lists, but we don’t wish this to be visible. What we would like is to use existential types to hide this, through the use of packages, like the following.

```
- val sched_queue_ty = ``:?'a:ty => ty. ('a,num)sched_q``;        25
> val sched_queue_ty =
    ``:?'a :ty => ty. ('a, num) sched_q``
     : hol_type

- val sched_queue_ty' = ty_antiq sched_queue_ty;
> val sched_queue_ty' =
    ``(ty_antiq( `:?'a :ty => ty. ('a, num) sched_q `))``
     : term
```

Here we create the desired type (sched_queue_ty) as an existential type, wrapping around the type ('a,num)sched_q, but hiding the actual type operator 'a by the existential type quantification. (The type antiquotation is required to use the type inside terms that are being parsed.)

Then a term of this existential type can be created using packages, taking our reference queue implementation and abstracting away from the actual list datatype.

```
- val reference_q_pkg = ``pack(:list, reference_q) : ^sched_queue_ty'``;   26
> val reference_q_pkg =
    ``pack (:list,(reference_q :(list, num) sched_q))``
     : term
```

We now create another possible implementation of priority queues. This one will use a pair of lists, where new additions to the queue are added to the first list at its front, and removals from the queue are taken from the second list at its front. In the case when the second list is empty, the first list is reversed and then replaces the second list.

```
val efficient_q_def = Define
   `efficient_q =
      <| this := ([] : num list, [] : num list);
         ops  := <| emptyq := \:'b. ([] : 'b list, [] : 'b list);
                    insert := \:'b. \x (xs,ys). (CONS x xs, ys);
                    remove := \:'b. \(xs,ys). REMOVE xs ys;
                    count  := \:'b. \x (xs,ys). COUNT x xs + COUNT x ys
                 |>
       |>`;
```

The operation of removing an element from this data structure is clearly more complex than before, and we express this using a subsidiary operator REMOVE that we intend to define. But REMOVE is nontrivial to define, as seen if we try the normal Define tool.

```
- val REMOVE_def = Define                                              27
    `(REMOVE (xs:'a list) ([]:'a list) = REMOVE [] (REVERSE xs)) /\
     (REMOVE xs (y::ys) = (y, (xs,ys)))`;
Initial goal:


?(R :'a list # 'a list -> 'a list # 'a list -> bool).
  WF R /\
  !(xs :'a list). R (([] :'a list),REVERSE xs) (xs,([] :'a list))



Exception raised at TotalDefn.Define:
between line 141, character 3 and line 142, character 36:
at TotalDefn.defnDefine:

Unable to prove termination!

Try using "TotalDefn.tDefine <name> <quotation> <tac>".
The termination goal has been set up using Defn.tgoal <defn>.
! Uncaught exception:
! HOL_ERR
```

The normal `Define` machinery is not able to automatically prove the termination of this definition, so the system does not accept this as a valid definition.

Instead, the definition package is directing us to Konrad Slind's excellent total recursive function definition package, which we will now use. We begin by reforming our definition using the `Hol_defn` tool as described in the *DESCRIPTION* manual.

```
- val REMOVE_defn = Hol_defn "REMOVE"                                  28
  `(REMOVE [] [] = (ARB, ([],[]))) /\
   (REMOVE (xs:'a list) ([]:'a list) = REMOVE [] (REVERSE xs)) /\
   (REMOVE xs (y::ys) = (y, (xs,ys)))`;
<<HOL message: mk_functional:
  pattern completion has added 1 clause to the original specification.>>
> val REMOVE_defn =
    HOL function definition (recursive)

    Equation(s) :
     [..]
    |- REMOVE ([] :'a list) ([] :'a list) =
       ((ARB :'a),([] :'a list),([] :'a list))
     [..]
    |- REMOVE ((v2 :'a)::(v3 :'a list)) ([] :'a list) =
       REMOVE ([] :'a list) (REVERSE (v2::v3))
     [..]
    |- REMOVE ([] :'a list) ((y :'a)::(ys :'a list)) = (y,([] :'a list),ys)
     [..]
    |- REMOVE ((v4 :'a)::(v5 :'a list)) ((y :'a)::(ys :'a list)) =
       (y,v4::v5,ys)
     Induction :                                                 29
      [..]
     |- !(P :'a list -> 'a list -> bool).
          P ([] :'a list) ([] :'a list) /\
          (!(v2 :'a) (v3 :'a list).
             P ([] :'a list) (REVERSE (v2::v3)) ==>
             P (v2::v3) ([] :'a list)) /\
          (!(y :'a) (ys :'a list). P ([] :'a list) (y::ys)) /\
          (!(v4 :'a) (v5 :'a list) (y :'a) (ys :'a list).
             P (v4::v5) (y::ys)) ==>
          !(v :'a list) (v1 :'a list). P v v1

     Termination conditions :
       0. !(v3 :'a list) (v2 :'a).
            (R :'a list # 'a list -> 'a list # 'a list -> bool)
              (([] :'a list),REVERSE (v2::v3)) (v2::v3,([] :'a list))
       1. WF (R :'a list # 'a list -> 'a list # 'a list -> bool)
      : defn
```

This has created an ML data structure (here stored in REMOVE_defn) which contains the elements of a provisional recursive function definition. In particular, it contains both the equations which are the reflections of our original specification, as well as an induction principle which can be used to induct over this particular pattern of recursion.

But none of this is available for us to use until we prove the termination of the definition. This we will have to do by hand. Fortunately, Slind's package provides a number of useful tools to aid our task.

First, we set up the termination property as a goal to be proved.

```
- Defn.tgoal REMOVE_defn;                                        30
> val it =
    Proof manager status: 1 proof.
    1. Incomplete goalstack:
         Initial goal:

         ?(R :'a list # 'a list -> 'a list # 'a list -> bool).
           WF R /\
           !(v3 :'a list) (v2 :'a).
             R (([] :'a list),REVERSE (v2::v3)) (v2::v3,([] :'a list))


      : proofs
```

We need to supply a well-founded relation as a witness for R, that strictly decreases for each call. A natural choice is `measure(\(xs,ys). LENGTH xs)`. This uses the constant `measure` which transforms a function of type `'a -> num` into a relation on `'a`.

```
- e (WF_REL_TAC `measure (\(xs,ys). LENGTH xs)`);                    31
OK..
1 subgoal:
> val it =

    !(v3 :'a list) (v2 :'a). LENGTH ([] :'a list) < LENGTH (v2::v3)

     : proof
```

This is easily solved by simplification.

```
- e (SIMP_TAC list_ss []);                                            32
OK..

Goal proved.
|- !(v3 :'a list) (v2 :'a). LENGTH ([] :'a list) < LENGTH (v2::v3)

> val it =
    Initial goal proved.
    |- ((REMOVE ([] :'a list) ([] :'a list) =
          ((ARB :'a),([] :'a list),([] :'a list))) /\
        (REMOVE ((v2 :'a)::(v3 :'a list)) ([] :'a list) =
         REMOVE ([] :'a list) (REVERSE (v2::v3))) /\
        (REMOVE ([] :'a list) ((y :'a)::(ys :'a list)) =
         (y,([] :'a list),ys)) /\
        (REMOVE ((v4 :'a)::(v5 :'a list)) (y::ys) = (y,v4::v5,ys))) /\
       !(P :'a list -> 'a list -> bool).
         P ([] :'a list) ([] :'a list) /\
         (!(v2 :'a) (v3 :'a list).
            P ([] :'a list) (REVERSE (v2::v3)) ==>
            P (v2::v3) ([] :'a list)) /\
         (!(y :'a) (ys :'a list). P ([] :'a list) (y::ys)) /\
         (!(v4 :'a) (v5 :'a list) (y :'a) (ys :'a list).
            P (v4::v5) (y::ys)) ==>
         !(v :'a list) (v1 :'a list). P v v1
     : proof
```

Now that we know the tactics to use, we can convert `REMOVE_defn` into a real definition by a single call to `Defn.tprove`.

```
- val (REMOVE_def,REMOVE_ind) =                                         33
  Defn.tprove (REMOVE_defn,
    WF_REL_TAC `measure (\(xs,ys). LENGTH xs)`
    THEN SIMP_TAC list_ss []
  );

> val REMOVE_def =
    |- (REMOVE ([] :'a list) ([] :'a list) =
        ((ARB :'a),([] :'a list),([] :'a list))) /\
       (REMOVE ((v2 :'a)::(v3 :'a list)) ([] :'a list) =
        REMOVE ([] :'a list) (REVERSE (v2::v3))) /\
       (REMOVE ([] :'a list) ((y :'a)::(ys :'a list)) =
        (y,([] :'a list),ys)) /\
       (REMOVE ((v4 :'a)::(v5 :'a list)) (y::ys) = (y,v4::v5,ys))
     : thm
  val REMOVE_ind =
    |- !(P :'a list -> 'a list -> bool).
         P ([] :'a list) ([] :'a list) /\
         (!(v2 :'a) (v3 :'a list).
            P ([] :'a list) (REVERSE (v2::v3)) ==>
            P (v2::v3) ([] :'a list)) /\
         (!(y :'a) (ys :'a list). P ([] :'a list) (y::ys)) /\
         (!(v4 :'a) (v5 :'a list) (y :'a) (ys :'a list).
            P (v4::v5) (y::ys)) ==>
         !(v :'a list) (v1 :'a list). P v v1
     : thm
```

Even better, we can wrap both the definition and its proof of termination all up in one piece by a single call to `tDefine`.

```
- val REMOVE_def =                                                      34
  tDefine "REMOVE"
   `(REMOVE [] [] = (ARB, ([],[]))) /\
    (REMOVE (xs:'a list) ([]:'a list) = REMOVE [] (REVERSE xs)) /\
    (REMOVE xs (y::ys) = (y, (xs,ys)))`
  (WF_REL_TAC `measure (\(xs,ys). LENGTH xs)`
   THEN SIMP_TAC list_ss []);
<<HOL message: mk_functional:
  pattern completion has added 1 clause to the original specification.>>
Equations stored under "REMOVE_def".
Induction stored under "REMOVE_ind".
> val REMOVE_def =
    |- (REMOVE ([] :'a list) ([] :'a list) =
        ((ARB :'a),([] :'a list),([] :'a list))) /\
       (REMOVE ((v2 :'a)::(v3 :'a list)) ([] :'a list) =
        REMOVE ([] :'a list) (REVERSE (v2::v3))) /\
       (REMOVE ([] :'a list) ((y :'a)::(ys :'a list)) =
        (y,([] :'a list),ys)) /\
       (REMOVE ((v4 :'a)::(v5 :'a list)) (y::ys) = (y,v4::v5,ys))
     : thm
```

The induction theorem is not returned directly, but we can obtain it from the current theory using the ML `theorem` command.

```
- val REMOVE_ind = theorem "REMOVE_ind";                           35
> val REMOVE_ind =
    |- !(P :'a list -> 'a list -> bool).
         P ([] :'a list) ([] :'a list) /\
         (!(v2 :'a) (v3 :'a list).
            P ([] :'a list) (REVERSE (v2::v3)) ==>
            P (v2::v3) ([] :'a list)) /\
         (!(y :'a) (ys :'a list). P ([] :'a list) (y::ys)) /\
         (!(v4 :'a) (v5 :'a list) (y :'a) (ys :'a list).
            P (v4::v5) (y::ys)) ==>
         !(v :'a list) (v1 :'a list). P v v1
     : thm
```

Now we prove a number of elementary facts about `REMOVE`.

```
- val REMOVE_CONS = store_thm(                                     36
    "REMOVE_CONS",
    ``!xs ys (y:'a). REMOVE xs (y::ys) = (y,(xs,ys))``,
    Cases
    THEN REWRITE_TAC [REMOVE_def]
   );
> val REMOVE_CONS =
    |- !(xs :'a list) (ys :'a list) (y :'a). REMOVE xs (y::ys) = (y,xs,ys)
     : thm

- val REVERSE_CONS_NOT_NIL = store_thm(
    "REVERSE_CONS_NOT_NIL",
    ``!xs (x:'a). ~(REVERSE (x::xs) = [])``,
    SIMP_TAC list_ss []
   );
> val REVERSE_CONS_NOT_NIL =
    |- !(xs :'a list) (x :'a). REVERSE (x::xs) <> ([] :'a list)
     : thm
```

The interaction of `REMOVE` with `COUNT` is more interesting, and we prove the following lemma for use later. It begins with a use of the `REMOVE` induction principle to follow the same recursion structure that `REMOVE` itself does.

```
- val REMOVE_COUNT = store_thm(                                        37
   "REMOVE_COUNT",
   ``!xs ys (u:'a) us vs.
        (REMOVE xs ys = (u,(us,vs))) ==>
        ~((xs = []) /\ (ys = [])) ==>
          !z. COUNT z xs + COUNT z ys =
              if z = u then COUNT z us + COUNT z vs + 1
                       else COUNT z us + COUNT z vs``,
   HO_MATCH_MP_TAC REMOVE_ind
   THEN REPEAT CONJ_TAC
   THEN REPEAT GEN_TAC
   THEN SIMP_TAC list_ss [REMOVE_def,COUNT_DEF,COUNT_APPEND,COUNT_REVERSE]
   THEN CONV_TAC (RATOR_CONV (ONCE_DEPTH_CONV SYM_CONV))
   THEN STRIP_TAC
   THEN GEN_TAC
   THEN COND_CASES_TAC
   THEN ASM_SIMP_TAC arith_ss [COUNT_DEF]
  );
> val REMOVE_COUNT =
    |- !(xs :'a list) (ys :'a list) (u :'a) (us :'a list) (vs :'a list).
         (REMOVE xs ys = (u,us,vs)) ==>
         ~((xs = ([] :'a list)) /\ (ys = ([] :'a list))) ==>
         !(z :'a).
           COUNT z xs + COUNT z ys =
           if z = u then
             COUNT z us + COUNT z vs + (1 :num)
           else
             COUNT z us + COUNT z vs
     : thm
```

Now we are ready to create our new scheduling queue implementation. We will call this the "efficient" implementation, because it is faster than the reference implementation, and would be a better candidate for eventual deployment.

```
- val efficient_q_def = Define                                         38
     `efficient_q =
        <| this := ([] : num list, [] : num list);
           ops  := <| emptyq := \:'b. ([] : 'b list, [] : 'b list);
                      insert := \:'b. \x (xs,ys). (CONS x xs, ys);
                      remove := \:'b. \(xs,ys). REMOVE xs ys;
                      count  := \:'b. \x (xs,ys). COUNT x xs + COUNT x ys
                    |>
         |>`;
```

```
Definition has been stored under "efficient_q_def"                      39
> val efficient_q_def =
    |- (efficient_q :('b list prod o list, num) sched_q) =
       <|this := (([] :num list),([] :num list));
         ops :=
           <|emptyq := (\:'b. (([] :'b list),([] :'b list)));
             insert :=
               (\:'b. (\(x :'b) ((xs :'b list),(ys :'b list)). (x::xs,ys)));
             remove :=
               (\:'b. (\((xs :'b list),(ys :'b list)). REMOVE xs ys));
             count :=
               (\:'b.
                  (\(x :'b) ((xs :'b list),(ys :'b list)).
                     COUNT x xs + COUNT x ys))|> |>
     : thm
```

The efficient implementation can be proven to satisfy the conditions to be a scheduling queue.

Before we start, we need to import the pair library, to be able to easily simplify expressions involving pairs, such as the pair of lists involved in this implementation.

```
- local open pairLib in end;                                            40
```

Here is the proof that the efficient queue is a scheduling queue. It uses the lemmas we have proven before, and a good deal of simplification.

```
- val efficient_q_is_scheduling_q = store_thm(                          41
   "efficient_q_is_scheduling_q",
   ``is_scheduling_q efficient_q.ops``,
   SRW_TAC [ARITH_ss] [efficient_q_def,is_scheduling_q_def,COUNT_DEF]
   THEN REPEAT (POP_ASSUM MP_TAC)
   THEN PairCases_on `q`
   THEN SRW_TAC [ARITH_ss] [COUNT_DEF]
   THEN REWRITE_TAC [ALL_COUNT_ZERO_2]
   THEN POP_ASSUM MP_TAC
   THEN Cases_on `q0`
   THEN Cases_on `q1`
   THEN PairCases_on `q'`
   THEN DISCH_TAC
   THEN IMP_RES_THEN MP_TAC REMOVE_COUNT
   THEN SRW_TAC [] [COUNT_DEF]
   THEN COND_CASES_TAC
   THEN ASM_SIMP_TAC arith_ss []
  );
> val efficient_q_is_scheduling_q =
    |- is_scheduling_q (efficient_q :('b list prod o list, num) sched_q).ops
     : thm
```

Similar to before, we can form a package from this scheduling queue, hiding the actual data structure.

```
- val efficient_q_pkg = ``pack(:\'a. 'a list # 'a list, efficient_q)      42
                          : ^sched_queue_ty'``;
> val efficient_q_pkg =
    ``pack
        (:'a list prod o list,
          (efficient_q :('b list prod o list, num) sched_q))``
     : term
```

We can even check to see if the two package terms for the two different implementations have the same type.

```
- val check = eq_ty (type_of reference_q_pkg) (type_of efficient_q_pkg);  43
> val check = true : bool
```

So we have now shown two different implementations of priority queues, that both satisfy the required properties, as specified in is_scheduling_q. Both of these implementations, despite their differences, are actually functionally equivalent; they both implement queues.

But this is not actually necessary to satisfy the properties of is_scheduling_q. We could implement a variety of other kinds of data structures that do not operate as pure queues, and as long as they satisfy is_scheduling_q, they will be acceptable. As an example, let's take an extreme variant. Instead of a first-in-first-out queue, we can implement a last-in-first-out stack. This will not have one very desirable property that queues do, of fairness, which guarantees that each element entered into the queue is eventually removed. But since that is not mentioned in is_scheduling_q, a stack is nevertheless an acceptable option.

```
- val stack_q_def = Define                                                 44
     `stack_q =
        <| this := [] : num list;
           ops  := <| emptyq := \:'b. [] : 'b list;
                      insert := \:'b. \x xs. CONS x xs;
                      remove := \:'b. \xs. (HD xs,TL xs);
                      count  := \:'b. \x xs. COUNT x xs
                    |>
         |>`;
Definition has been stored under "stack_q_def"

> val stack_q_def =
    |- (stack_q :(list, num) sched_q) =
       <|this := ([] :num list);
         ops :=
           <|emptyq := (\:'b. ([] :'b list));
             insert := (\:'b. (\(x :'b) (xs :'b list). x::xs));
             remove := (\:'b. (\(xs :'b list). (HD xs,TL xs)));
             count := (\:'b. (\(x :'b) (xs :'b list). COUNT x xs))|> |>
     : thm
```

This is a simple implementation, so it is not hard to prove that it is a scheduling queue.

```
- val stack_q_is_scheduling_q = store_thm(                            45
   "stack_q_is_scheduling_q",
   ``is_scheduling_q stack_q.ops``,
   SRW_TAC [] [stack_q_def,is_scheduling_q_def,COUNT_DEF]
   THEN Cases_on `q`
   THEN SRW_TAC [] [COUNT_DEF]
  );
> val stack_q_is_scheduling_q =
    |- is_scheduling_q (stack_q :(list, num) sched_q).ops
     : thm
```

As for the other implementations, the scheduling stack can be wrapped up in a package, hiding its implementation type (`list`).

```
- val stack_q_pkg = ``pack(:list, stack_q) : ^sched_queue_ty``;      46
> val stack_q_pkg =
    ``pack (:list,(stack_q :(list, num) sched_q))``
     : term
```

The purpose of the scheduling stack is not to suggest this as a suitable data structure to actually schedule processes, but just to show how different implementations may have significantly different functional behavior. Whatever the properties we specify, those are what any suitable candidate implementation must meet. Furthermore, those are also the properties, and only the properties, which the rest of the program can expect any scheduling queue to meet. Thus this choice of the properties of the data structure helps to form a boundary, isolating information between parts of the program, which contributes to good system design and modularity.

The advantage of wrapping these implementations up as packages is that we hide their implementation types, so that we can write common, general routines that make use of any of them, interchangably.

The following function counts how many elements in a scheduling package are equal to a given element.

```
- val countp_def = Define                                              47
   ‘countp i (p:^sched_queue_ty’) =
      let (:’a,q) = p in
      q.ops.count i q.this‘;
Definition has been stored under "countp_def"
> val countp_def =
    |- !(i :num) (p :?’a :ty => ty. (’a, num) sched_q).
         countp i p =
         (let (:’a :ty => ty,(q :(’a, num) sched_q)) = p
          in
            q.ops.count [:num:] i q.this)
     : thm
```

Here is a function to take a package and create an empty version of that same kind of package.

```
- val emptyp_def = Define                                              48
   ‘emptyp (p:^sched_queue_ty’) =
      let (:’a,q) = p in
        pack(:’a, <| this := q.ops.emptyq [:num:];
                     ops  := q.ops |> )‘;
Definition has been stored under "emptyp_def"
> val emptyp_def =
    |- !(p :?’a :ty => ty. (’a, num) sched_q).
         emptyp p =
         (let (:’a :ty => ty,(q :(’a, num) sched_q)) = p
          in
            pack
              (:’a :ty => ty,
                <|this := q.ops.emptyq [:num:]; ops := q.ops|>))
     : thm
```

Here is a function to insert an element into a scheduling package.

```
- val insertp_def = Define                                             49
   ‘insertp i (p:^sched_queue_ty’) =
      let (:’a,q) = p in
        pack(:’a, <| this := q.ops.insert i q.this;
                     ops  := q.ops |> )‘;
Definition has been stored under "insertp_def"
> val insertp_def =
    |- !(i :num) (p :?’a :ty => ty. (’a, num) sched_q).
         insertp i p =
         (let (:’a :ty => ty,(q :(’a, num) sched_q)) = p
          in
            pack
              (:’a :ty => ty,
                <|this := q.ops.insert [:num:] i q.this; ops := q.ops|>))
     : thm
```

This function removes an element of the scheduling package and returns it, paired with the diminished scheduling package.

50
```
- val removep_def = Define
   `removep (p:^sched_queue_ty') =
      let (:'a,q) = p in
      let (x,this') = q.ops.remove q.this in
        (x, pack(:'a, <| this := this';
                         ops  := q.ops |>))`;
Definition has been stored under "removep_def"
> val removep_def =
    |- !(p :?'a :ty => ty. ('a, num) sched_q).
         removep p =
         (let (:'a :ty => ty,(q :('a, num) sched_q)) = p
          in
            let ((x :num),(this' :num 'a)) = q.ops.remove [:num:] q.this
            in
              (x,(pack (:'a :ty => ty,<|this := this'; ops := q.ops|>))))
     : thm
```

Similarly, we can lift the definition of the properties of a scheduling queue to packages.

51
```
- val is_scheduling_p_def = Define
   `is_scheduling_p (p : ?'b. ('b,'a)sched_q) =
      let (:'a,q) = p in
      is_scheduling_q q.ops`;
Definition has been stored under "is_scheduling_p_def"
> val is_scheduling_p_def =
    |- !(p :?'b :ty => ty. ('b, 'a) sched_q).
         is_scheduling_p p <=>
         (let (:'b :ty => ty,(q :('b, 'a) sched_q)) = p
          in
            is_scheduling_q q.ops) : thm
```

The type checking of packages ensures that the representation type of the package is not disclosed outside of the **let** ... **in** form. If we try to do this, say by returning the internal data structure that holds the elements of the queue,

52
```
- val thisp_def = Define
   `thisp (p:^sched_queue_ty') =
      let (:'a,q) = p in
        q.this`;
```

we would see an error message like the following.

```
Exception raised at Preterm.typecheck:                                  53
roughly on line 326, characters 8-13:

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

(UNPACK :(!('x :ty => ty). ('x, num) sched_q -> num ('a :ty => ty)) ->
         (?('y :ty => ty). ('y, num) sched_q) -> num 'a)

roughly on line 325, characters 10-20

to

\:'a :ty => ty. (\(q :('a, num) sched_q). q.this)

roughly on line 326, characters 8-13

which has type

:!'a :ty => ty. ('a, num) sched_q -> num 'a

unification failure message: unify failed

! Uncaught exception:
! HOL_ERR
```

Now we can take each of our implementations, define their packaged versions, and show that the packaged version satisfies the predicate `is_scheduling_p`.

Here we define the packaged version of the reference scheduling queue.

```
- val reference_p_def = Define                                          54
     `reference_p = pack(:list, reference_q) : ^sched_queue_ty'`;
Definition has been stored under "reference_p_def"
> val reference_p_def =
    |- (reference_p :?'a :ty => ty. ('a, num) sched_q) =
       (pack (:list,(reference_q :(list, num) sched_q)))
     : thm

- val reference_p_is_scheduling_p = store_thm(
     "reference_p_is_scheduling_p",
     ``is_scheduling_p reference_p``,
     SIMP_TAC bool_ss [is_scheduling_p_def,reference_p_def,
                       reference_q_is_scheduling_q]
  );
> val reference_p_is_scheduling_p =
    |- is_scheduling_p (reference_p :?'a :ty => ty. ('a, num) sched_q)
     : thm
```

This defines the packaged version of the efficient scheduling queue.

```
- val efficient_p_def = Define                                         55
     ‘efficient_p = pack(:\’a. ’a list # ’a list, efficient_q)
                     : ^sched_queue_ty’‘;
Definition has been stored under "efficient_p_def"
> val efficient_p_def =
    |- (efficient_p :?’a :ty => ty. (’a, num) sched_q) =
       (pack
          (:’a list prod o list,
            (efficient_q :(’b list prod o list, num) sched_q)))
     : thm

- val efficient_p_is_scheduling_p = store_thm(
     "efficient_p_is_scheduling_p",
     ‘‘is_scheduling_p efficient_p‘‘,
     SIMP_TAC bool_ss [is_scheduling_p_def,efficient_p_def,
                       efficient_q_is_scheduling_q]
  );
> val efficient_p_is_scheduling_p =
    |- is_scheduling_p (efficient_p :?’a :ty => ty. (’a, num) sched_q)
     : thm
```

And finally, this defines the packaged version of the stack scheduling queue.

```
- val stack_p_def = Define                                             56
     ‘stack_p = pack(:list, stack_q) : ^sched_queue_ty’‘;
Definition has been stored under "stack_p_def"
> val stack_p_def =
    |- (stack_p :?’a :ty => ty. (’a, num) sched_q) =
       (pack (:list,(stack_q :(list, num) sched_q)))
     : thm

- val stack_p_is_scheduling_p = store_thm(
     "stack_p_is_scheduling_p",
     ‘‘is_scheduling_p stack_p‘‘,
     SIMP_TAC bool_ss [is_scheduling_p_def,stack_p_def,
                       stack_q_is_scheduling_q]
  );
> val stack_p_is_scheduling_p =
    |- is_scheduling_p (stack_p :?’a :ty => ty. (’a, num) sched_q)
     : thm
```

Programmers on a project using these scheduling queues will need to be able to access the desired properties of the queue, even if they do not know exactly what the implementation is. Furthermore, since they will not have access to the internal record structure of the implementation, we should express these properties in terms of the functions they do have access to, that is, the functions defined above that work on packages, namely `countp`, `emptyp`, `insertp`, and `removep`.

To make the properties of a scheduling package easily available, we will first prove that any package satisfying is_scheduling_p will be guaranteed of fulfilling the following properties.

57

```
- val scheduling_p_props = store_thm(
   "scheduling_p_props",
   ``!(p: ^sched_queue_ty').
        is_scheduling_p p ==>
        (!x. countp x (emptyp p) = 0) /\
        (!x y. countp x (insertp y p) =
               if x = y then countp x p + 1
                        else countp x p) /\
        (~(!x. countp x p = 0) ==>
               let (y,p') = removep p
               in !x. countp x p = countp x (insertp y p'))``,
   GEN_TAC
   THEN REWRITE_TAC [is_scheduling_p_def]
   THEN STRIP_ASSUME_TAC (ISPEC ``p:^sched_queue_ty'`` PACK_ONTO_AX)
   THEN ASM_REWRITE_TAC []
   THEN SIMP_TAC bool_ss []
   THEN REWRITE_TAC [is_scheduling_q_def]
   THEN REPEAT STRIP_TAC
   THENL
     [ SRW_TAC [] [countp_def,emptyp_def],

       SRW_TAC [] [countp_def,insertp_def],

       POP_ASSUM MP_TAC
       THEN SRW_TAC [] [countp_def,removep_def,insertp_def]
       THEN FIRST_ASSUM (MP_TAC o Q.SPEC `t.this` o TY_SPEC ``:num``)
       THEN COND_CASES_TAC
       THENL
         [ POP_ASSUM (STRIP_ASSUME_TAC o SPEC ``x:num``),

           SRW_TAC [] [LET_DEF]
         ]
     ]
  );
> val scheduling_p_props =
    |- !(p :?'a :ty => ty. ('a, num) sched_q).
         is_scheduling_p p ==>
         (!(x :num). countp x (emptyp p) = (0 :num)) /\
         (!(x :num) (y :num).
            countp x (insertp y p) =
            if x = y then countp x p + (1 :num) else countp x p) /\
         (~(!(x :num). countp x p = (0 :num)) ==>
          (let ((y :num),(p' :?'a :ty => ty. ('a, num) sched_q)) = removep p
           in
             !(x :num). countp x p = countp x (insertp y p')))
     : thm
```

For convenience, we can break this up into three theorems for each of the properties, named by the operation they concentrate on.

58

```
- val emptyp_prop = store_thm(
   "emptyp_prop",
   ``!(p: ^sched_queue_ty').
        is_scheduling_p p ==>
        !x. countp x (emptyp p) = 0``,
   SIMP_TAC bool_ss [scheduling_p_props]
  );
> val emptyp_prop =
    |- !(p :?'a :ty => ty. ('a, num) sched_q).
         is_scheduling_p p ==> !(x :num). countp x (emptyp p) = (0 :num)
     : thm

- val insertp_prop = store_thm(
   "insertp_prop",
   ``!(p: ^sched_queue_ty').
        is_scheduling_p p ==>
        !x y. countp x (insertp y p) =
               if x = y then countp x p + 1
                        else countp x p``,
   SIMP_TAC bool_ss [scheduling_p_props]
  );
> val insertp_prop =
    |- !(p :?'a :ty => ty. ('a, num) sched_q).
         is_scheduling_p p ==>
         !(x :num) (y :num).
           countp x (insertp y p) =
           if x = y then countp x p + (1 :num) else countp x p
     : thm

- val removep_prop = store_thm(
   "removep_prop",
   ``!(p: ^sched_queue_ty').
        is_scheduling_p p ==>
        ~(!x. countp x p = 0) ==>
               let (y,p') = removep p
               in !x. countp x p = countp x (insertp y p')``,
   SIMP_TAC bool_ss [scheduling_p_props]
  );
> val removep_prop =
    |- !(p :?'a :ty => ty. ('a, num) sched_q).
         is_scheduling_p p ==>
         ~(!(x :num). countp x p = (0 :num)) ==>
         (let ((y :num),(p' :?'a :ty => ty. ('a, num) sched_q)) = removep p
          in
            !(x :num). countp x p = countp x (insertp y p'))
     : thm
```

These properties hold about any scheduling queue package that satisfies `is_scheduling_p`. In particular, the properties hold for the example implementations defined earlier.

59

```
- val _ = set_trace "types" 0;
- val reference_p_props = save_thm(
   "reference_p_props",
   MATCH_MP scheduling_p_props reference_p_is_scheduling_p);
> val reference_p_props =
    |- (!x. countp x (emptyp reference_p) = 0) /\
       (!x y.
          countp x (insertp y reference_p) =
          if x = y then
            countp x reference_p + 1
          else
            countp x reference_p) /\
       (~(!x. countp x reference_p = 0) ==>
        (let (y,p') = removep reference_p
         in
           !x. countp x reference_p = countp x (insertp y p')))
     : thm
- val efficient_p_props = save_thm(
   "efficient_p_props",
   MATCH_MP scheduling_p_props efficient_p_is_scheduling_p);
> val efficient_p_props =
    |- (!x. countp x (emptyp efficient_p) = 0) /\
       (!x y.
          countp x (insertp y efficient_p) =
          if x = y then
            countp x efficient_p + 1
          else
            countp x efficient_p) /\
       (~(!x. countp x efficient_p = 0) ==>
        (let (y,p') = removep efficient_p
         in
           !x. countp x efficient_p = countp x (insertp y p')))
     : thm
- val stack_p_props = save_thm(
   "stack_p_props",
   MATCH_MP scheduling_p_props stack_p_is_scheduling_p);
> val stack_p_props =
    |- (!x. countp x (emptyp stack_p) = 0) /\
       (!x y.
          countp x (insertp y stack_p) =
          if x = y then countp x stack_p + 1 else countp x stack_p) /\
       (~(!x. countp x stack_p = 0) ==>
        (let (y,p') = removep stack_p
         in
           !x. countp x stack_p = countp x (insertp y p')))
     : thm
```

This concludes the exercise on scheduling queues. The purpose of this exercise was to show how a modular boundary can be established between the implementation of a data structure and the rest of the program that uses that data structure. The specific properties are freely chosen by the system design team, and form a good documentation of the interface at this modular boundary.

In this fashion, packages and existential types promote good programming practices, and this also contributes to a cleaner overall system, more resilient under future changes, and a considerable lessening of the proof effort required to revalidate the system when such changes occur.

Chapter 13

More Examples

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

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

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

`euclid.sml` This example is the same as that covered in Chapter 6: a proof of Euclid's theorem on the infinitude of the prime numbers, extracted and modified from a much larger development due to John Harrison. It illustrates the automation of HOL on a classic proof.

`ind_def` This directory contains some examples of an inductive definition package in action. Featured are an operational semantics for a small imperative programming language, a small process algebra, and combinatory logic with its type system. The files were originally developed by Tom Melham and Juanito Camilleri and are extensively commented. The last is the basis for Chapter 8.

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

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

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

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

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

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

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

In addition to the examples above, the examples covered in this tutorial concerning the HOL-Omega logic are also present, as well as further developments, including the following:

`appetizersScript.sml` This file gives a series of examples that in a light and easy way briefly demonstrate the essential new features of the HOL-Omega logic.

`functorScript.sml` This example shows how a simple version of category theory can be nicely realized as a shallow embedding within the new logic. Both functors and natural transformations are defined, and examples of each are demonstrated. This is similar to a development for HOL2P originally written by Norbert Völker.

`aopScript.sml` Building on the functor theory above, this shows several examples taken from *The Algebra of Programming*, by Richard Bird and Oege de Moor. These include homomorphisms, initial algebras, catamorphisms, and the banana split theorem. This development was originally written by Norbert Völker for HOL2P.

`monadScript.sml` Also building on the functor theory above, this defines the concept of a monad in three different ways, and proves the three are equivalent. Multiple examples of monads are presented, and also how one can convert a monad from one of the styles of definition to another style.

`type_specScript.sml` This file contains examples of creating new types using the new definitional principle for type specification which has been added to the HOL-Omega theorem prover. In particular, this is used to create a new type by specifying it as the initial algebra of a signature. The example used is taken from a 1993 paper by Tom Melham, "The HOL Logic Extended with Quantification over Type Variables."

`packageScript.sml` This example shows more completely how packages and existential types may be created and used to hide the information about data types. Many of the examples are taken from and related to chapter 24 of the book ”Types and Programmng Languages” by Benjamin C. Pierce, MIT Press, 2002. There is also an extended example on process scheduling queues.

`burali_fortiScript.sml` This file contains a development of the Burali-Forti paradox in the HOL-Omega logic, which attempts to prove false by a clever manipulation of the type system. This is the same as Girard's Paradox, which showed that the naïve combination of higher order logic and an advanced type system was inconsistent. The development in this file demonstrates how the HOL-Omega logic, which is both a higher order logic and an advanced type system, prevents the inconsistency that the paradox attempts to expose. This is not a proof of the logic's consistency, but it is a strong demonstration of its resilience in the face of a sophisticated and subtle attack. This work is described further in an upcoming paper, "The HOL-Omega Logic and Girard's Paradox."

`interim` This directory contains an extensive, worked example of a generalized version of category theory, created by Jeremy Dawson of the Australian National University. This generalizes the notions of functor and natural transformation from those in `functorScript.sml`, to allow for a much richer realization of category theory. For example, multiple categories, each with their own composition and identity operations, may have functors defined between them. The development of category theory is continued through the definition of adjoints, and introduces an innovative extension of monads. This example extensively exercises the kind structure of HOL-Omega, to manage the types relating different categories and the operations among them.

References

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

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

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

[4] M. Davis, G. Logemann and D. Loveland, 'A machine program for theorem proving', *Communications of the ACM*, Vol. 5 (1962), pp. 394–397.

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

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

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

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

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

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

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

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

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