VERSION 2 (for HOL88.2.0) **December 30, 1993**

The HOL System

DESCRIPTION

University of Cambridge DSTO SRI International

Preface

This volume contains the description of the HOL system. It is one of three volumes making up the documentation for HOL:

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

(ii) *DESCRIPTION*: a description of higher order logic, the ML programming language, and theorem proving methods in the HOL system;

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

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

This document, *DESCRIPTION*, is intended to serve both as a definition of HOL and as an advanced guide for users with some prior experience of the system. Beginners should start with the companion document *TUTORIAL*.

The HOL system is designed to support interactive theorem proving in higher order logic (hence the acronym 'HOL'). To this end, the formal logic is interfaced to a general purpose programming language (ML, for meta-language) in which terms and theorems of the logic can be denoted, proof strategies expressed and applied, and logical theories developed. This document presents the definitions of the meta-language and the logic, and it explains the means by which meta-language functions can be used to generate proofs in the logic.

The version of higher order logic used in HOL is predicate calculus with terms from the typed lambda calculus (i.e. simple type theory). This was originally developed as a foundation for mathematics [3]. The primary application area of HOL was initially intended to be the specification and verification of hardware designs. (The use of higher order logic for this purpose was first advocated by Keith Hanna [10].) However, the logic does not restrict applications to hardware; HOL has been applied to many other areas.

The approach to mechanizing formal proof used in HOL is due to Robin Milner [9], who also headed the team that designed and implemented the language ML. That work centred on a system called LCF (logic for computable functions), which was intended for interactive automated reasoning about higher order recursively defined functions. The interface of the logic to the meta-language was made explicit, using the type structure of ML, with the intention that other logics eventually be tried in place of the original logic.

The HOL system is a direct descendant of LCF; this is reflected in everything from its structure and outlook to its incorporation of ML, and even to parts of its implementation. Thus HOL satisfies the early plan to apply the LCF methodology to other logics.

The original LCF was implemented at Edinburgh in the early 1970's, and is now referred to as 'Edinburgh LCF'. Its code was ported from Stanford Lisp to Franz Lisp by Gérard Huet at INRIA, and was used in a French research project called 'Formel'. Huet's Franz Lisp version of LCF was further developed at Cambridge by Larry Paulson, and became known as 'Cambridge LCF'. The HOL system is implemented on top of an early version of Cambridge LCF and consequently many features of both Edinburgh and Cambridge LCF were inherited by HOL. For example, the axiomatization of higher order logic used is not the classical one due to Church, but an equivalent formulation influenced by LCF.

The language ML has now achieved status as a programming language in its own right, although it was originally designed as the proof management language for LCF. It is a functional language distinguished particularly for its type inference mechanism, that gives type security without overburdening the user. Types, and especially abstract types, are the basis for distinguishing the theorems of a logic from arbitrary formulae, in a secure way.

A standard has now been established for ML ([11]), but for historical reasons, the HOL system includes an earlier version of ML. There are, however, two new implementations of HOL in progress that are based on Standard ML rather than Lisp (see below). The most recent version of Cambridge LCF, which is documented in Paulson's book [16], is also based on Standard ML.

In this document, the acronym 'HOL' refers to both the interactive theorem proving system and to the version of higher order logic that the system supports; where there is serious ambiguity, the former is called 'the HOL system' and the latter 'the HOL logic'.

An enhanced and rationalized version of HOL, called HOL88, was released (in 1988), after the original HOL system had been in use for several years. HOL88 Version 2.0 (HOL88.2.0) is the system described in this document. This is intended to serve as a stable platform for a number of research projects and technology transfer activities that are in progress at Cambridge, and elsewhere, at the time of writing. It is also the supported version of the system for the international HOL community. The main differences between Version 2.0 and earlier releases are described in Appendix A.

The future

As mentioned above, there are two new implementations of HOL nearing completion, both based on Standard ML instead of Lisp. One is a public domain system developed at the University of Calgary (preliminary versions of this are already available). The other, which is being developed by ICL Secure Systems, is a 'highly assured' system for commercial verification projects in the safety and security critical areas.

The contact address for Calgary HOL is:

Graham Birtwistle
Computer Science Department
University of Calgary
2500 University Drive
Calgary
Alberta
Canada T2N 1N4

(email: graham@cpsc.ucalgary.ca)

and the contact address for ICL HOL is:

Roger B. Jones
International Computers Limited
Eskdale Road
Winnersh
Wokingham
Berkshire RG11 5TT
England

(email: R.B.Jones@win0103.uucp

The Cambridge group intends to continue to support the Lisp based version of HOL at least until it is clearly superseded (which is not expected to be for several years). The emphasis will be on improving the existing system and documentation, rather than adding new features. However, there is a research project in progress at Inria (Sofia-Antipoles, France), to develop a new and experimental interface to HOL. This research is being done in collaboration with the Cambridge HOL group and aims to answer the question of whether HOL can be made significantly easier to use with a state-of-the-art mouse and window interface. It is also exploring the use of colour for managing complexity. If this experiment is successful, then a production version of the new interface may be produced for general use. It is not expected that this will be undertaken before 1992.

Acknowledgements

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 first edition of *TUTORIAL* contained case studies on microprocessor systems (by Jeff Joyce), protocol verification (by Rachel Cardell-Oliver) and modular arithmetic based on group theory (by Elsa Gunter). These are now separate documents in the HOL distribution directory `hol/Training/studies`. The chapter in *TUTORIAL* on the proof of the binomial theorem in HOL was written by Andy Gordon.

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

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 conversion of the `troff` sources of *The ML Handbook* to LaTeX was done by Inder Dhingra and John Van Tassel. The cover design is by Arnold Smith, who used a photograph of a 'snow watching lantern' taken by Avra Cohn (in whose garden the original object resides). John Van Tassel composed the LaTeX picture of the lantern.

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

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

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

Contents

I The Meta-language ML **1**

1 The History of ML **3**
1.1 Preface to 'The ML Handbook' 4
1.2 Preface to 'Edinburgh LCF' 4

2 Introduction and Examples **7**
2.1 Expressions 7
2.2 Declarations 7
2.3 Assignment 8
2.4 Functions 9
2.5 Recursion 10
2.6 Iteration 11
2.7 Lists 12
2.8 Strings 13
2.9 Polymorphism 13
2.10 Lambda-expressions 14
2.11 Failure 15
2.12 Type abbreviations 17
2.13 Concrete types 17
2.14 Abstract types 19
2.15 Type constructors 20

3 Syntax of ML **23**
3.1 Syntax equations for ML 24
3.2 Notes on the syntax equations for ML 26
3.3 Identifiers and other lexical matters 27
3.3.1 Identifiers 27
3.3.2 Constant expressions 29
3.3.3 Prefixes and infixes 29

4 Semantics of ML **31**
4.1 Declarations 33
4.1.1 The evaluation of bindings 33

4.1.2 Matching patterns and expression values 34
4.2 Expressions 35

5 ML Types **39**
5.1 Types and objects 39
5.1.1 The syntax of types 40
5.2 Typing of ML phrases 41
5.3 Discussion of type constraints 45
5.4 Type abbreviations 47
5.5 Concrete types 48
5.6 Abstract types 49

6 Primitive ML Identifier Bindings **51**
6.1 Predeclared ordinary identifiers 51
6.2 Predeclared dollared identifiers 53

7 General Purpose and List Processing Functions **55**
7.1 General purpose functions and combinators 55
7.2 Miscellaneous list processing functions 58
7.3 List mapping and iterating functions 59
7.4 List searching functions 61
7.5 List transforming functions 62
7.6 Functions for lists representing sets 64
7.7 Miscellaneous string processing functions 65
7.8 Failure handling functions 67

8 ML System Functions **69**
8.1 Input and output 69
8.1.1 Reading and writing files 69
8.1.2 Terminal input and output 70
8.1.3 Loading ML files 70
8.1.4 Compiling ML files 72
8.2 Output 72
8.2.1 Pretty printing 72
8.3 Exiting and re-entering the system 76
8.4 Autoloading 76
8.5 Interpreting lists of numbers as ML input 79
8.6 Initialization 80
8.7 Operating system calls from ML 80
8.8 Getting the version number 81
8.9 Ordering of ML values 81

8.10 Printing defined types . . . 82
8.11 Lisp in ML . . . 82
8.12 Fast arithmetic . . . 82
8.13 Flags . . . 83
8.14 Modifying the ML read-eval-print loop . . . 85
8.14.1 Changing the ML prompt . . . 85
8.14.2 Changing the top-level printing of values in ML . . . 86
8.14.3 Suppressing the printing of ML types . . . 87
8.15 The help system . . . 87
8.16 Syntax blocks . . . 88
8.17 Relocating HOL . . . 89
8.18 Libraries . . . 90
8.18.1 Libraries available in Version 2.0 . . . 91
8.18.2 Location of libraries . . . 92
8.18.3 Adding a new library . . . 92
8.19 Experimental support for a new interface . . . 93

II The HOL Logic 97

9 Syntax and Semantics 99
9.1 Introduction . . . 99
9.2 Types . . . 100
9.2.1 Type structures . . . 102
9.2.2 Semantics of types . . . 102
9.2.3 Instances and substitution . . . 104
9.3 Terms . . . 105
9.3.1 Terms-in-context . . . 106
9.3.2 Semantics of terms . . . 107
9.3.3 Substitution . . . 109
9.4 Standard notions . . . 111
9.4.1 Standard type structures . . . 111
9.4.2 Standard signatures . . . 111

10 Theories 113
10.1 Introduction . . . 113
10.2 Sequents . . . 114
10.3 Logic . . . 115
10.3.1 The HOL deductive system . . . 115
10.3.2 Soundness theorem . . . 117
10.4 HOL Theories . . . 118

10.4.1 The theory `MIN` . . . 118
10.4.2 The theory `LOG` . . . 119
10.4.3 The theory `INIT` . . . 121
10.4.4 Consistency . . . 122
10.5 Extensions of theories . . . 122
10.5.1 Extension by constant definition . . . 123
10.5.2 Extension by constant specification . . . 125
10.5.3 Remarks about constants in HOL . . . 127
10.5.4 Extension by type definition . . . 128
10.5.5 Extension by type specification[3] . . . 130

III The HOL System 135

11 The HOL Logic in ML 137
11.1 Lexical matters . . . 137
11.1.1 Identifiers . . . 138
11.1.1.1 Separators . . . 139
11.1.2 Type variable names . . . 140
11.2 Types . . . 140
11.3 Terms . . . 141
11.4 Quotation . . . 142
11.4.1 Type quotation . . . 144
11.4.2 Term quotation . . . 144
11.4.3 Special syntactic forms . . . 145
11.5 Syntax for restricted quantification . . . 146
11.5.1 Conditionals . . . 149
11.5.2 Paired abstractions . . . 150
11.5.3 `let`-terms . . . 151
11.6 Syntax for sets . . . 152
11.6.1 Antiquotation . . . 154
11.6.2 User-programmable quotation typechecker . . . 154
11.7 Theorems . . . 156
11.8 Theories . . . 158
11.8.1 Primitive ML functions for creating theories . . . 159
11.8.2 Functions for creating definitional extensions . . . 161
11.8.2.1 Constant definitions . . . 161
11.8.2.2 Constant specifications . . . 163
11.8.2.3 Type definitions . . . 164
11.8.2.4 Defining bijections . . . 165

11.8.2.5 Properties of type bijections . . . 166
11.8.3 Primitive ML function for accessing theories . . . 167
11.8.4 The theory `HOL` . . . 168
11.8.4.1 The theory `bool` . . . 168
11.8.4.2 The theory `ind` . . . 170
11.8.5 Primitive rules of inference of the HOL Logic . . . 171
11.8.5.1 Assumption introduction . . . 171
11.8.5.2 Reflexivity . . . 171
11.8.5.3 Beta-conversion . . . 171
11.8.5.4 Substitution . . . 172
11.8.5.5 Abstraction . . . 172
11.8.5.6 Type instantiation . . . 173
11.8.5.7 Discharging an assumption . . . 173
11.8.5.8 Modus Ponens . . . 173
11.8.6 Type abbreviations . . . 174
11.9 The ancestry of the theory `HOL` . . . 175
11.9.1 The theory `PPLAMB` . . . 176
11.9.2 The theory `bool` . . . 176
11.9.2.1 Pairs and the type `prod` . . . 177
11.9.3 The theory `BASIC-HOL` . . . 179
11.9.4 The theory `num` . . . 179
11.9.4.1 Numerals . . . 179
11.9.5 The theory `prim_rec` . . . 180
11.9.5.1 Primitive recursive definitions . . . 180
11.9.5.2 The less-than relation . . . 183
11.9.6 The theory `arithmetic` . . . 184
11.9.7 The theory `list` . . . 184
11.9.8 The theory `combin` . . . 186
11.9.9 The theories `tree` and `ltree` . . . 187
11.9.9.1 The theory `tree` . . . 187
11.9.9.2 The theory `ltree` . . . 188
11.9.10 The theory `tydefs` . . . 189
11.9.11 The theory `sum` . . . 189
11.9.12 The theory `one` . . . 190
11.10 The type definition package . . . 190
11.10.1 Defining types . . . 191
11.10.1.1 Input syntax . . . 192
11.10.1.2 The type specified . . . 192
11.10.1.3 The output . . . 192
11.10.1.4 Examples . . . 194

11.10.2 Defining recursive functions . . . 195
11.10.2.1 More examples . . . 197
11.10.3 Structural induction . . . 198
11.10.3.1 Examples . . . 199
11.10.4 Structural induction tactics . . . 199
11.10.5 Other tools . . . 201
11.10.5.1 Examples . . . 201
11.11 Miscellaneous system features . . . 202
11.11.1 Flags for the HOL logic . . . 203
11.11.2 Sticky types and default types . . . 204
11.11.3 Interface maps . . . 205
11.11.4 Changing the lambda symbol . . . 209
11.11.5 Changing the turnstile symbol . . . 209
11.11.6 Hiding constants . . . 210
11.11.7 Autoloading of axioms, definitions and theorems . . . 211
11.11.8 Cached theories . . . 211
11.11.9 Adjusting the pretty-print depth . . . 212
11.11.10 Timing and counting theorems . . . 213

IV Theorem Proving with HOL 215

12 Derived Inference Rules 217
12.1 Simple derivations . . . 217
12.2 Rewriting . . . 220
12.3 Derivation of the standard rules . . . 221
12.3.1 Adding an assumption . . . 223
12.3.2 Undischarging . . . 224
12.3.3 Symmetry of equality . . . 224
12.3.4 Transitivity of equality . . . 224
12.3.5 Application of a term to a theorem . . . 225
12.3.6 Application of a theorem to a term . . . 225
12.3.7 Modus Ponens for equality . . . 225
12.3.8 Implication from equality . . . 226
12.3.9 T-Introduction . . . 226
12.3.10 Equality-with-T elimination . . . 226
12.3.11 Specialization (∀-elimination) . . . 227
12.3.12 Equality-with-T introduction . . . 227
12.3.13 Generalization (∀-introduction) . . . 228
12.3.14 Simple α-conversion . . . 228

12.3.15 η-conversion . . . 229
12.3.16 Extensionality . . . 230
12.3.17 ε-introduction . . . 230
12.3.18 ε-elimination . . . 231
12.3.19 $\exists$-introduction . . . 231
12.3.20 $\exists$-elimination . . . 232
12.3.21 Use of a definition . . . 232
12.3.22 Use of a definition . . . 233
12.3.23 $\wedge$-introduction . . . 233
12.3.24 $\wedge$-elimination . . . 234
12.3.25 Right $\vee$-introduction . . . 234
12.3.26 Left $\vee$-introduction . . . 235
12.3.27 $\vee$-elimination . . . 235
12.3.28 Classical contradiction rule . . . 236

13 Conversions **237**
13.1 Conversion combining operators . . . 239
13.2 Writing compound conversions . . . 243
13.3 Built in conversions . . . 246
13.3.1 Generalized beta-reduction . . . 247
13.3.2 Arithmetical conversions . . . 247
13.3.3 List processing conversions . . . 248
13.3.4 Simplifying `let`-terms . . . 248
13.3.5 Skolemization . . . 249
13.3.6 Quantifier movement conversions . . . 250
13.4 Rewriting tools . . . 252

14 Goal Directed Proof: Tactics and Tacticals **255**
14.1 Tactics, goals and justifications . . . 256
14.1.1 Details of proving theorems . . . 261
14.2 The subgoal package . . . 262
14.3 Some tactics built into HOL . . . 265
14.3.1 Acceptance of a theorem . . . 265
14.3.2 Adding an assumption . . . 266
14.3.3 Specialization . . . 266
14.3.4 Conjunction . . . 267
14.3.5 Discharging an assumption . . . 267
14.3.6 Combined simple decompositions . . . 267
14.3.7 Substitution . . . 267
14.3.8 Case analysis on a boolean term . . . 268

14.3.9 Case analysis on a disjunction . . . 268
14.3.10 Rewriting . . . 268
14.3.11 Resolution by Modus Ponens . . . 269
14.3.12 Identity . . . 270
14.3.13 Null . . . 270
14.3.14 Splitting logical equivalences . . . 270
14.3.15 Solving existential goals . . . 271
14.4 Tacticals . . . 271
14.4.1 Alternation . . . 271
14.4.2 First success . . . 272
14.4.3 Change detection . . . 272
14.4.4 Sequencing . . . 272
14.4.5 Selective sequencing . . . 274
14.4.6 Successive application . . . 274
14.4.7 Repetition . . . 275
14.5 Tactics for manipulating assumptions . . . 275
14.5.1 Theorem continuations with popping . . . 276
14.5.2 Theorem continuations without popping . . . 283

References **287**

A Version 2.0 **289**
A.1 Directory reorganization . . . 290
A.2 Location of libraries . . . 291
A.3 More flexible help system . . . 292
A.4 Syntax for restricted quantification . . . 293
A.5 Syntax for sets . . . 295
A.6 Revised set theory libraries . . . 297
A.7 Syntax blocks . . . 299
A.8 User-programmable quotation typechecker . . . 300
A.9 New conversions . . . 301
A.9.1 Generalized beta-reduction . . . 301
A.9.2 Arithmetical conversions . . . 302
A.9.3 List processing conversions . . . 302
A.9.4 Simplifying `let`-terms . . . 303
A.9.5 Skolemization . . . 304
A.9.6 Quantifier movement conversions . . . 305
A.10 New built-in theorems . . . 306
A.10.1 New arithmetical theorems . . . 306
A.10.2 New theorems about lists . . . 308

A.10.3 New propositional theorems 308
A.11 Additions and deletions 308
A.11.1 Additions 309
A.11.2 Deletions 310

Index **313**

Part I

The Meta-language ML

Chapter 1

The History of ML

Several versions of the programming language ML have appeared over the years, between the time it was first designed and implemented by Milner, Morris and Wadsworth at the University of Edinburgh in the early 1970's, and the time it was settled and standardised, in the mid-1980's. The original ML, the meta-language of the Edinburgh LCF system, is defined in [9]. The genealogy of the ML used in the HOL system is rather complex; in brief, it is a version roughly midway between the original ML and Standard ML, frozen while the language was in flux. More precisely, it is a subset of the version of ML developed jointly by Huet and Paulson in the early 1980's, at INRIA and the University of Cambridge, respectively. The ML of Huet and Paulson is described in the preface (reproduced below) to 'The ML Handbook' [5]. Huet used this version in the Formel project; and it subsequently evolved into a version of ML called CAML. Paulson also used it, as part of the first version of Cambridge LCF (but not the version of Cambridge LCF described in [16], which used an early Standard ML).

There are now two mature descendents of the original ML: Standard ML (also from Edinburgh) and CAML (from INRIA). The ML in HOL is neither of these, but, as suggested above, is rather closer to CAML than to Standard ML. A completely new implementation of HOL, with Standard ML as its meta-language, is currently in progress at the University of Calgary. This is expected to be completed in mid-1990. A second new implementation is planned by ICL Defence Systems, also based on Standard ML.

The ML of Huet and Paulson was implemented in Franz Lisp. The porting of HOL from Franz Lisp to Common Lisp was done by John Carroll of the University of Cambridge, with the joint support of SRI International and Hewlett Packard Limited.

The description of ML that appears in Chapters 3 to 6 is based very closely on 'The ML Handbook' [5]. It was adapted for HOL purposes by Mike Gordon, and modified for the present document by Tom Melham. The draft was produced by Inder Dhingra and Mike Gordon (with help from John Van Tassel) from the `troff` sources of 'The ML Handbook'.

For completeness (and historical interest), the preface to 'The ML Handbook'[1] and the preface to 'Edinburgh LCF: a Mechanised Logic of Computation' are reproduced below.

[1] Minor errors have been corrected in this.

1.1 Preface to 'The ML Handbook'

This handbook is a revised edition of Chapter 2 of 'Edinburgh LCF', by M. Gordon, R. Milner, and C. Wadsworth, published in 1979 as Springer Verlag Lecture Notes in Computer Science n^{o} 78. ML was originally the meta-language of the LCF system. The ML system was adapted to Maclisp on Multics by Gérard Huet at INRIA in 1981, and a compiler was added. Larry Paulson from the University of Cambridge completely redesigned the LCF proving system, which stabilised in 1984 as Cambridge LCF. Guy Cousineau from the University Paris VII added concrete types in the summer of 1984. Philippe Le Chenadec from INRIA implemented an interface with the Yacc parser generator system, for the versions of ML running under Unix. This permits the user to associate a concrete syntax with a concrete type.

The ML language is still under design. An extended language was implemented on the VAX by Luca Cardelli in 1981. It was then decided to completely re-design the language, in order to accommodate in particular the call by pattern feature of the language HOPE designed by Rod Burstall and David MacQueen. A committee of researchers from the Universities of Edinburgh and Cambridge, the Bell Laboratories and INRIA, headed by Robin Milner, is currently working on the new extended language, called Standard ML. Progress reports appear in the Polymorphism Newsletter, edited by Luca Cardelli and David MacQueen from Bell Laboratories. The design of a core language is now frozen, and its description will appear in a forthcoming report of the University of Edinburgh, as 'The Standard ML Core Language' by Robin Milner.

This handbook is a manual for ML version 6.1, released in December 1984. The language is somewhere in between the original ML from LCF and standard ML, since Guy Cousineau added the constructors and call by patterns. This is a LISP based implementation, compatible for Maclisp on Multics, Franzlisp on VAX under Unix, Zetalisp on Symbolics 3600, and Le_Lisp on 68000, VAX, Multics, Perkin-Elmer, etc... Video interfaces have been implemented by Philippe Le Chenadec on Multics, and by Maurice Migeon on Symbolics 3600. The ML system is maintained and distributed jointly by INRIA and the University of Cambridge.

1.2 Preface to 'Edinburgh LCF'

ML is a general purpose programming language. It is derived in different aspects from ISWIM, POP2 and GEDANKEN, and contains perhaps two new features. First, it has an escape and escape trapping mechanism, well-adapted to programming strategies which may be (in fact usually are) inapplicable to certain goals. Second, it has a polymorphic type discipline which combines the flexibility of programming in a typeless language with the security of compile-time type checking (as in other languages, you may also define your own types, which may be abstract and/or recursive).

For those primarily interested in the design of programming languages, a few remarks here may be helpful both about ML as a candidate for comparison with other recently designed languages, and about the description of ML which we provide. On the first point, although we did not set out with programming language design as a primary aim, we believe that ML does contain features worthy of serious consideration; these are the escape mechanism and the polymorphic type discipline mentioned above, and also the attempt to make programming with functions—including those of higher type—as easy and natural as possible. We are less happy about the imperative aspects of the language, and would wish to give them further thought if we were mainly concerned with language design. In particular, the constructs for controlling iteration both by boolean conditions and by escape-trapping (which we included partly for experiment) are perhaps too complex taken together, and we are sensitive to the criticism that escape (or failure, as we call it) reports information only in the form of a string. This latter constraint results mainly from our type discipline; we do not know how best to relax the constraint while maintaining the discipline.

Concerning the description of ML, we have tried both to initiate users by examples of programming and to give a precise definition.

Chapter 2

Introduction and Examples

ML is an interactive language. At top-level one can:

- evaluate expressions
- perform declarations

To give a first impression of the system, we reproduce below a session at a terminal in which simple uses of various ML constructs are illustrated. To make the session easier to follow, it is split into a sequence of sub-sessions. A complete description of the syntax and semantics of ML is given in Chapter 3 and Chapter 4 respectively.

2.1 Expressions

The ML prompt is `#` so lines beginning with this contain the user's contribution; all other lines are output by the system.

1

```
#2+3;;
5 : int

#it;;
5 : int
```

ML prompted with `#`, the user then typed `2+3;;` followed by a carriage return; ML then responded with `5 : int`, a new line, and then prompted again. The user then typed `it;;` followed by a carriage return, and the system responded by typing `5 : int` again. In general to evaluate an expression e one types e followed by a carriage return; the system then prints e's value and type (the type prefaced by a colon). The value of the last expression evaluated at top level is remembered in the identifier `it`.

2.2 Declarations

The declaration `let` x`=`e evaluates e and binds the resulting value to x.

```
#let x=2*3;;                                                    2
x = 6 : int

#it=x;;
false : bool
```

Notice that declarations do not affect `it`. To bind the variables $x_1, \ldots, x_n$ simultaneously to the values of the expressions $e_1, \ldots, e_n$ one can perform either the declaration `let` x_1`=`e_1 `and` x_2`=`$e_2 \ldots$ `and` x_n`=`e_n or `let` $x_1, x_2, \ldots, x_n$`=`$e_1, e_2, \ldots, e_n$. These two declarations are equivalent.

```
#let y=10 and z=x;;                                             3
y = 10 : int
z = 6 : int

#let x,y = y,x;;
x = 10 : int
y = 6 : int
```

A declaration d can be made local to the evaluation of an expression e by evaluating the expression d `in` e. The expression e `where` b (where b is a *binding* such as `x=2`) is equivalent to `let` b `in` e.

```
#let x=2 in x*y;;                                               4
12 : int

#x;;
10 : int

#x*y where x=2;;
12 : int
```

2.3 Assignment

Identifiers can be declared assignable using `letref` instead of `let`. Values bound to such identifiers[1] can be changed with the assignment expression x`:=`e, which changes the value bound to x to be the value of e. Attempts to assign to non-assignable variables are detected by the type checker.

[1] In this document, it is sometimes said that values are bound to identifiers, but it is also sometimes said that identifiers are bound to values. This imprecision will be corrected in later versions.

```
#x:=1;;                                                              5

unbound or non-assignable variable x
1 error in typing
typecheck failed

#letref x=1 and y=2;;
x = 1 : int
y = 2 : int

#x:=6;;
6 : int

#x;;
6 : int
```

The value of an assignment x`:=`e is the value of e (hence the value of `y:=6` is 6). Simultaneous assignments can also be done:

```
#x,y := y,x;;                                                        6
(2,6) : (int # int)

#x,y;;
(2,6) : (int # int)
```

The type `(int # int)` is the type of pairs of integers.

2.4 Functions

To define a function f with formal parameter x and body e one performs the declaration: `let` f x `=` e. To apply the function f to an actual parameter e one evaluates the expression: f e.

```
#let f x = 2*x;;                                                     7
f = - : (int -> int)

#f 4;;
8 : int
```

Functions are printed as a dash, `-`, followed by their type, since a function as such is not printable. Application binds more tightly than anything else in the language; thus, for example, f `3 + 4` means `(`f `3)+4` not f`(3+4)`. Functions of several arguments can be defined:

```
#let add x y = x+y;;                                                    8
add = - : (int -> int -> int)

#add 3 4;;
7 : int

#let f = add 3;;
f = - : (int -> int)

#f 4;;
7 : int
```

Application associates to the left so `add 3 4` means `(add 3)4`. In the expression `add 3`, the function `add` is partially applied to `3`; the resulting value is the function of type `int -> int` which adds `3` to its argument. Thus `add` takes its arguments one at a time. We could have made `add` take a single argument of the cartesian product type `(int # int)`:

```
#let add(x,y) = x+y;;                                                   9
add = - : ((int # int) -> int)

#add(3,4);;
7 : int

#let z = (3,4) in add z;;
7 : int

#add 3;;

ill-typed phrase: 3
has an instance of type  int
which should match type  (int # int)
1 error in typing
typecheck failed
```

As well as taking structured arguments (e.g. `(3,4)`) functions may also return structured results.

```
#let sumdiff(x,y) = (x+y,x-y);;                                         10
sumdiff = - : ((int # int) -> (int # int))

#sumdiff(3,4);;
(7, -1) : (int # int)
```

2.5 Recursion

The following is an attempt to define the factorial function:

```
#let fact n = if n=0 then 1 else n*fact(n-1);;                    11

unbound or non-assignable variable fact
1 error in typing
typecheck failed
```

The problem is that any free variables in the body of a function have the bindings they had just before the function was declared; `fact` is such a free variable in the body of the declaration above, and since it is not defined before its own declaration, an error results. To make things clear consider:

```
#let f n = n+1;;                                                  12
f = - : (int -> int)

#let f n = if n=0 then 1 else n*f(n-1);;
f = - : (int -> int)

#f 3;;
9 : int
```

Here `f 3` results in the evaluation of `3*f(2)`, but now the first `f` is used so `f(2)` evaluates to `2+1=3`, hence the expression `f 3` results in `3*3=9`. To make a function declaration hold within its own body, `letrec` instead of `let` must be used. The correct recursive definition of the factorial function is thus:

```
#letrec fact n = if n=0 then 1 else n*fact(n-1);;                 13
fact = - : (int -> int)

#fact 3;;
6 : int
```

2.6 Iteration

Here is an iterative definition of `fact` using two local assignable variables: `count` and `result`.

```
#let fact n =                                                     14
#    letref count=n and result=1
#      in while count > 0 do count,result := count-1,count*result
#         ;result;;
fact = - : (int -> int)
```

The semicolon denotes sequencing. When an expression $e_1;\ldots;e_n$ is evaluated, each e_i is evaluated in turn and the value of the entire expression is the value of e_n. The while construct binds more tightly than sequencing.

The construct `if` e_1 `then` e_2 `loop` e_3 is the same as `if` e_1 `then` e_2 `else` e_3 in the true case; when e_1 evaluates to false, e_3 is evaluated and control loops back to the front of the construct again. As an illustration, here is an alternative definition of `fact`.

15
```
#let fact n =
#    letref count=n and result=1
#    in     if count=0
#           then result
#           loop count,result := count-1,count*result;;
fact = - : (int -> int)

#fact 4;;
24 : int
```

Replacing the `then` in `if` e_1 `then` e_2 `else` e_3 by `loop` causes iteration when e_1 evaluates to true: e.g., `if` e_1 `loop` e_2 `else` e_3 is equivalent to `if not(`e_1`) then` e_3 `loop` e_2. The conditional/loop construct can have a number of conditions, each preceded by `if`. The expression guarded by each condition may be preceded by `then`, or by `loop` when the whole construct is to be re-evaluated after evaluating the guarded expression:

16
```
#let gcd(x,y) =
#    letref x,y = x,y
#    in     if x>y loop x:=x-y
#           if x<y loop y:=y-x
#           else x;;
gcd = - : ((int # int) -> int)

#gcd(12,20);;
4 : int
```

2.7 Lists

If $e_1, \ldots, e_n$ all have type ty then the ML expression `[`e_1`;...;`e_n`]` has type (*ty* `list`). The standard functions on lists are `hd` (head), `tl` (tail), `null` (which tests whether a list is empty—i.e. is equal to `[]`), and the infixed operators `.` (cons) and `@` (append, or concatenation).

17
```
#let m = [1;2;(2+1);4];;
m = [1; 2; 3; 4] : int list

#hd m , tl m;;
(1, [2; 3; 4]) : (int # int list)

#null m , null [];;
(false, true) : (bool # bool)

#0.m;;
[0; 1; 2; 3; 4] : int list

#[1; 2] @ [3; 4; 5; 6];;
[1; 2; 3; 4; 5; 6] : int list

#[1;true;2];;

ill-typed phrase: true
has an instance of type   bool
which should match type   int
1 error in typing
typecheck failed
```

All the members of a list must have the same type (although this type could be a sum, or disjoint union type—see Chapter 5).

2.8 Strings

A sequence of characters enclosed between string quotes (‘ – i.e. ascii 96) is a string.

18
```
#‘this is a string‘;;
‘this is a string‘ : string

#‘‘;;
‘‘ : string
```

2.9 Polymorphism

The list processing functions `hd`, `tl` etc. can be used on all types of lists.

19
```
#hd [1;2;3];;
1 : int

#hd [true;false;true];;
true : bool

#hd [1,2;3,4];;
(1, 2) : (int # int)
```

Thus `hd` has several types; for example, it is used above with types `(int list) -> int`, `(bool list) -> bool` and `(int # int) list -> (int # int)`. In fact if *ty* is *any* type then `hd` has the type (*ty* `list) -> ` *ty*. Functions, like `hd`, with many types are called *polymorphic*, and ML uses type variables `*`, `**`, `***` etc. to represent their types.

20
```
#hd;;
- : (* list -> *)

#letrec map f l = if null l then []
#                           else f(hd l).map f (tl l);;
map = - : ((* -> **) -> * list -> ** list)

#map fact [1;2;3;4];;
[1; 2; 6; 24] : int list
```

The ML function `map` takes a function f (with argument type `*` and result type `**`), and a list l (of elements of type `*`), and returns the list obtained by applying f to each element of l (which is a list of elements of type `**`). `map` can be used at any instance of its type: above, both `*` and `**` were instantiated to `int`; below, `*` is instantiated to `(int list)` and `**` to `bool`. Notice that the instance need not be specified; it is determined by the type checker.

21
```
#map null [[1;2]; []; [3]; []];;
[false; true; false; true] : bool list
```

2.10 Lambda-expressions

The expression `\`x`.` e evaluates to a function with formal parameter x and body e. Thus the declaration `let` f x `=` e is equivalent to `let` f `=` `\`$x.e$. Similarly `let` f`(`x`,`y`)`z `=` e is equivalent to `let` f `=` `\(`x`,`y`). \`z`.` e. Repeated `\`'s, as in `\(`x`,`y`). \`z`.` e, may be abbreviated by `\(`x`,`y`)` $z.e$. The character `\` is our representation of lambda, and expressions like `\`x`.` e and `\(`x`,`y`)` $z.e$ are called lambda-expressions.

22
```
#\x.x+1;;
- : (int -> int)

#it 3;;
4 : int

#map (\x.x*x) [1;2;3;4];;
[1; 4; 9; 16] : int list

#let doubleup = map (\x.x@x);;
doubleup = - : (* list list -> * list list)

#doubleup [ [1;2]; [3;4;5] ];;
[[1; 2; 1; 2]; [3; 4; 5; 3; 4; 5]] : int list list

#doubleup [];;
[] : * list list
```

2.11 Failure

Some standard functions *fail* at run-time on certain arguments, yielding a string (which is usually the function name) to identify the sort of failure. A failure with string ‘t‘ may also be generated explicitly by evaluating the expression `failwith` ‘t‘ (or more generally `failwith` e where e has type `string`).

23
```
#hd(tl[2]);;
evaluation failed     hd

#1/0;;
evaluation failed     div

#(1/0)+1000;;
evaluation failed     div

#failwith (hd [‘a‘;‘b‘]);;
evaluation failed     a
```

A failure can be trapped by `?`; the value of the expression e_1`?`e_2 is that of e_1, unless e_1 causes a failure, in which case it is the value of e_2.

```
#hd(tl[2]) ? 0;;                                                    24
0 : int

#(1/0)?1000;;
1000 : int

#let half n =
#    if n=0 then failwith `zero`
#           else let m=n/2
#                in  if n=2*m then m else failwith`odd`;;
half = - : (int -> int)
```

The function `half` only succeeds on non-zero even numbers; on `0` it fails with `` `zero` ``, and on odd numbers it fails with `` `odd` ``.

```
#half 4;;                                                           25
2 : int

#half 0;;
evaluation failed zero

#half 3;;
evaluation failed odd

#half 3 ? 1000;;
1000 : int
```

Failures may be trapped selectively (on string) by `??`; if e_1 fails with string t, then the value of e_1 `??` `[`t_1`;...;`t_n`]` e_2 is the value of e_2 if t is one of t_1,...,t_n; otherwise the expression fails with the value of t.

```
#half(0) ?? [`zero`;`plonk`] 1000;;                                 26
1000 : int

#half(1) ?? [`zero`;`plonk`] 1000;;
evaluation failed     odd
```

One may add several `??` traps to an expression, and one may add a `?` trap at the end as a catch-all.

```
#half(1)                                                            27
#  ??[`zero`] 1000
#  ??[`odd`]  2000;;
2000 : int

#hd(tl[half(4)])
#  ??[`zero`] 1000
#  ??[`odd`]  2000
#  ? 3000;;
3000 : int
```

One may use `!` or `!!` in place of `?` or `??` to cause re-iteration of the whole construct, analogously to using `loop` in place of `then`.

28
```
#let same(x,y) =
#    if x>y then failwith `greater`
#    if x<y then failwith `less`
#           else x;;
same = - : ((int # int) -> int)

#let gcd(x,y) =
#    letref x,y = x,y
#    in  same(x,y)
#          !![`greater`] x:=x-y
#          !![`less`]    y:=y-x;;
gcd = - : ((int # int) -> int)

#gcd(12,20);;
4 : int
```

2.12 Type abbreviations

Types can be given names:

29
```
#lettype intpair = int # int;;
type intpair defined

#let p = 12,20;;
p = (12, 20) : intpair
```

The new name is simply an abbreviation; for example, `intpair` and `int#int` are completely equivalent. The system always uses the most recently defined name when printing types.

30
```
#gcd;;
- : (intpair -> int)

#gcd p;;
4 : int
```

2.13 Concrete types

New types (rather than mere abbreviations) can also be defined. Concrete types are types defined by a set of constructors which can be used to create objects of that type and also (in patterns) to decompose objects of that type. For example, to define a type `card` one could use the construct `type`:

```
#type card = king | queen | jack | other of int;;

New constructors declared:
king : card
queen : card
jack : card
other : (int -> card)
```
31

Such a declaration declares `king`, `queen`, `jack` and `other` as constructors and gives them values. The value of a 0-ary constructor such as `king` is the constant value `king`. The value of a constructor such as `other` is a function that given an integer value n produces `other`(n).

```
#king;;
king : card

#other 9;;
other 9 : card
```
32

To define functions that take their argument from a concrete type, we shall extend the notion of lambda-expression to expressions of the form `fun` $pat_1.e_1$ |... | $pat_n.e_n$ The idea is that such an expression denotes a function that given a value v selects the first pattern that matches v, say pat_i, binds the variables of pat_i to the corresponding components of the value and then evaluates the expression e_i. We use `fun` rather than `\` in such expressions because `\` tends to be difficult to see. However, when function definitions have only one case, we use indifferently `fun` or `\`. We could for example define the values of the different cards in the following way:

```
#let value = fun  king . 500
#              |  queen . 200
#              |  jack . 100
#              |  (other n) . 5*n;;
value = - : (card -> int)
```
33

The notion of concrete type is very basic and could enable us to build ML elementary types from scratch. For example, the booleans could be defined simply by:

```
#type bool = true | false;;

New constructors declared:
true : bool
false : bool
```
34

and the positive integers by:

```
#rectype int = zero | suc of int;;                                    35

New constructors declared:
zero : int
suc : (int -> int)
```

Notice that we have employed `rectype` here instead of `type` because the type `int` is defined in terms of itself. In a similar way, LISP S-expressions could be defined by:

```
#rectype sexp =   litatom of string                                   36
#               | numatom of int
#               | cons    of sexp # sexp;;

New constructors declared:
litatom : (string -> sexp)
numatom : (int -> sexp)
cons : ((sexp # sexp) -> sexp)

#let car (cons(x,y)) = x
#and cdr (cons(x,y)) = y;;
car = - : (sexp -> sexp)
cdr = - : (sexp -> sexp)
```

A definition of the form `let` *fn pat* `=` *e* is an equivalent form for `let` *fn* `= fun` *pat* `.` *e* or `let` *fn* `= \`*pat* `.` *e* which can be used when a function definition has only one case. We see from the example of `car` and `cdr` that the set of patterns used in a function definition need not be exhaustive. When such a function is applied to a value and if that value cannot be matched by any pattern, a failure occurs with value ‘*pattern*‘.

```
#car (litatom ‘foo‘);;                                                37
evaluation failed    pattern
```

The case construct permits one to compute by cases on an expression of a concrete type. The expression `case` *e* `of` pat_1 `.` e_1 `|` ... `|` pat_n `.` e_n, is an equivalent form for the application `(fun` pat_1 `.` e_1 `|` ... `|` pat_n `.` e_n`)` *e*.

2.14 Abstract types

New types can also be defined by abstraction. For example, to define a type `time` we could use the construct `abstype`:

```
#abstype time = int # int                                              38
# with maketime(hrs,mins) = if  hrs<0 or 23<hrs or
#                              mins<0 or 59<mins
#                            then fail
#                            else abs_time(hrs,mins)
# and hours t = fst(rep_time t)
# and minutes t = snd(rep_time t);;
maketime = - : (intpair -> time)
hours = - : (time -> int)
minutes = - : (time -> int)
```

This defines an abstract type `time` and three primitive functions: `maketime`, `hours` and `minutes`. In general, an abstract type declaration has the form `abstype` ty`=`ty' `with` b where b is a binding, i.e. the kind of phrase that can follow `let` or `letrec`. Such a declaration introduces a new type ty which is represented by ty'. Only within b can one use the (automatically declared) functions `abs_`ty (of type ty'`->`ty) and `rep_`ty (of type ty`->`ty'), which map between a type and its representation. In the example above `abs_time` and `rep_time` are only available in the definitions of `maketime`, `hours` and `minutes`; these latter three functions, on the other hand, are defined throughout the scope of the declaration. Thus an abstract type declaration simultaneously declares a new type together with primitive functions for the type. The representation of the type (i.e. ty'), and of the primitives (i.e. the right hand sides of the definitions in b), is not accessible outside the `with`-part of the declaration.

```
#let t = maketime(8,30);;                                              39
t = - : time

#hours t , minutes t;;
(8, 30) : intpair
```

Notice that values of an abstract type are printed as `-`, like functions.

2.15 Type constructors

Both `list` and `#` are examples of type constructors; `list` has one argument (hence `* list`) whereas `#` has two (hence `* # **`). Each type constructor has various primitive operations associated with it, for example `list` has `null`, `hd`, `tl`, ... etc, and `#` has `fst`, `snd` and the infix '`,`'.

```
#let z = it;;                                                          40
z = (8, 30) : intpair

#fst z;;
8 : int

#snd z;;
30 : int
```

Another standard constructor of two arguments is `+`; `* + **` is the disjoint union of types `*` and `**`, and associated with it are the following primitives:

`isl : (* + **) -> bool`	tests membership of left summand
`inl : * -> (* + **)`	injects into left summand
`inr : * -> (** + *)`	injects into right summand
`outl : (* + **) -> *`	projects out of left summand
`outr : (* + **) -> **`	projects out of right summand

These are illustrated by:

41

```
#let x = inl 1
#and y = inr 2;;
x = inl 1 : (int + *)
y = inr 2 : (* + int)

#isl x;;
true : bool

#isl y;;
false : bool

#outl x;;
1 : int

#outl y;;
evaluation failed     outl

#outr x;;
evaluation failed     outr

#outr y;;
2 : int
```

Note that `isr` is not provided as it is easily defined as the complement of `isl`.

Concrete types such as `card` or `sexp` and abstract types such as `time` defined above can be thought of as type constructors with no arguments (i.e. nullary constructors). The `type...` and `abstype...with...` constructs may also be used to define non-nullary type constructors. For example, the disjoint union of types can be easily redefined by:

42

```
#type (*,**) sum = inl of * | inr of **;;

New constructors declared:
inl : (* -> (*,**) sum)
inr : (* -> (**,*) sum)
```

The other primitives of the type `sum` can be defined by:

```
#let isl = fun (inl x) . true | (inr x) . false;;                    43
isl = - : ((*,**) sum -> bool)

#let outl = fun (inl x) . x | (inr x) . failwith `outl`;;
outl = - : ((*,**) sum -> *)

#let outr = fun (inl x) . failwith `outr` | (inr x) . x;;
outr = - : ((*,**) sum -> **)
```

The type constructor `list` can be redefined by:

```
#rectype * list = nil | cons of * # * list;;                         44

New constructors declared:
nil : * list
cons : ((* # * list) -> * list)

#let hd (cons(x,y)) = x
#and tl (cons(x,y)) = y
#and null = fun [] . true | _ . false;;
hd = - : (* list -> *)
tl = - : (* list -> * list)
null = - : (* list -> bool)
```

This is like the built-in type of lists, except that the empty list is `nil` rather than `[]`.

The symbol _ is a special symbol that can be used in patterns to match anything and saves the burden of inventing useless variable names. It can have multiple occurrences in a pattern with possibly incompatible types.

The type constructor `set` can be defined in the following way:

```
#abstype * set = * list                                              45
#with emptyset = abs_set []
#and  isempty = \s . null (rep_set s)
#and  member  = letrec member = fun ( _ , []) . false
#                                | (x,(y.z)) . (x=y) or member(x,z)
#               in \(u,v) . member(u,rep_set v)
#and  add = letrec add = fun (x,[])    . [x]
#                         | (x,(y.z)) . if x=y then y.z else y.add(x,z)
#           in \(u,v) . abs_set(add(u,rep_set v));;
emptyset = - : * set
isempty = - : (* set -> bool)
member = - : ((* # * set) -> bool)
add = - : ((* # * set) -> * set)
```

Chapter 3

Syntax of ML

We shall use variables to range over the various constructs of ML as follows:

Variable	Ranges over
var	variables
con	constructors
ce	constant expressions
ty	types
tab	type abbreviation bindings (see 5.4)
cb	concrete type bindings (see 5.5)
ab	abstract type bindings (see 5.6)
d	declarations
b	bindings
p	patterns
e	expressions

Variables and constructors are both represented by identifiers but they are different syntax classes. Identifiers and constant expressions are described in Section 3.3 below. Types and type-bindings are explained in Chapter 5. Declarations, bindings, patterns and expressions are defined by the following BNF-like syntax equations in which:

1. Each variable ranges over constructs as above.

2. The numbers following the various variables are there merely to distinguish between different occurrences—this will be convenient when we describe the semantics in Chapter 4.

3. {C} denotes an optional occurrence of C, and for $n > 1$ {C_1 | C_2 ... | C_n} denotes a choice of exactly one of $C_1, C_2, \ldots, C_n$.

4. The constructs are listed in order of decreasing binding power.

5. 'L' or 'R' following a construct means that it associates to the left (L) or right (R) when juxtaposed with itself (where this is syntactically admissible).

6. Certain constructs are equivalent to others and this is indicated by 'equiv.' followed by the equivalent construct.

3.1 Syntax equations for ML

Declarations d

d ::=	`let` b	ordinary variables
|	`letref` b	assignable variables
|	`letrec` b	recursive functions
|	`lettype` tab	type abbreviations
|	`type` cb	
|	`rectype` cb	recursive concrete types
|	`abstype` ab	abstract types
|	`absrectype` ab	recursive abstract types

Bindings b

b ::=	p`=`e	simple binding
|	`var` p_1 $p_2 \ldots p_n$ `{:`ty`} =` e	function definition
|	b_1 `and` $b_2 \ldots$ `and` b_n	multiple binding

Patterns p

p ::=	`_`	wildcard
|	ce	constant
|	var	variable
|	con p	construction
|	p`:`ty	type constraint
|	p_1`.`p_2	R list cons
|	p_1`,`p_2	R pairing
|	`[]`	empty list
|	`[`p_1`;`$p_2 \ldots$`;`p_n`]`	list of n elements
|	`(`p`)`	equiv. p

Expressions e

e ::=	ce	constant
|	var	variable
|	con	constructor
|	$e_1 e_2$	L function application
|	e:ty	type constraint
|	-e	unary minus
|	e_1*e_2	L multiplication
|	e_1/e_2	L division
|	e_1+e_2	L addition
|	e_1-e_2	L subtraction
|	e_1<e_2	less than
|	e_1>e_2	greater than
|	e_1.e_2	R list cons
|	e_1@e_2	R list append
|	e_1=e_2	L equality
|	not e	negation
|	e_1&e_2	R conjunction
|	e_1 or e_2	R disjunction
|	e_1=>e_2|e_3	R equiv. if e_1 then e_2 else e_3
|	do e	evaluate e for side effects
|	e_1,e_2	R pairing
|	p:=e	assignment
|	fail	equiv. failwith 'fail'
|	failwith e	failure with explicit string
|	if e_1 {then|loop} e_1' {if e_2 {then|loop} e_2' ⋮ if e_n {then|loop} e_n'} {{else|loop} e_n''}	conditional and loop

| e {??|!!} $e_1 e_1'$ {??|!!} $e_2 e_2'$ $\vdots$ {??|!!} $e_n e_n'$ {{?|!|?id|!id} e_n''}	R failure trap and loop
| `while` e_1 `do` e	while loop
| e_1;e_2 ... ;e_n	sequencing
| `[]`	empty list
| [e_1;e_2 ... ; e_n]	list of n elements
| e `where` b	R equiv. `let` b `in` e
| e `whereref` b	R equiv. `letref` b `in` e
| e `whererec` b	R equiv. `letrec` b `in` e
| e `wheretype` db	equiv. `lettype` db `in` e
| e `whereabstype` ab	equiv. `abstype` ab `in` e
| e `whereabsrectype` ab	equiv. `absrectype` ab `in` e
| d `in` e	local declaration
| `case` e `of` $p_1.e_1$ | ...| $p_n.e_n$	case statement, equiv. (`fun` $p_1.e_1$ | ... | $p_n.e_n$) e
| `\` $p_1 p_2 \ldots p_n$. e	abstraction
| `fun` $p_1.e_1$ | $p_2.e_2$ | ... | $p_n.e_n$	pattern selection abstraction
| (e)	equiv. e

3.2 Notes on the syntax equations for ML

1. In the syntax equations constructs are listed in order of decreasing binding power. For example, since $e_1 e_2$ is listed before e_1;e_2 function application binds more tightly than sequencing and thus $e_1 e_2$;e_3 parses as ($e_1 e_2$);e_3. This convention determines only the relative binding power of different constructs. The left or right association of a construct is indicated explicitly by 'L' for left and 'R' for right. For example, as application associates to the left, the expression $e_1 e_2 e_3$ parses as ($e_1 e_2$)e_3, and since e_1=>e_2|e_3 associates to the right, the expression e_1=>e_2|e_3=>e_4|e_5 parses as e_1=>e_2|(e_3=>e_4|e_5).

2. Only functions can be defined with `letrec`. For example, `letrec x = 2-x` would cause a syntax error.

3. All the variables occurring in a pattern must be distinct. On the other hand, a pattern can contain multiple occurrences of the wildcard '_'.

4. Spaces (ASCII 32), carriage returns (ASCII 13), line feeds (ASCII 10) form feeds (^L,

ASCII 12) and tabs (`^I`, ASCII 9) can be inserted and deleted arbitrarily without affecting the meaning (as long as obvious ambiguities are not introduced). For example, the space in `-` x but not in `not` x can be omitted. Comments can be of two kinds:

(a) Arbitrary sequences of characters surrounded by `%`'s.

(b) Text enclosed in the nesting comment brackets `%<` and `>%`. The only way to terminate a comment starting with `%<` is with a matching `>%`. If `%`x, where x is not `<`, occurs outside a `%<` ... `>%` pair, then the comment is terminated by the next `%`. Thus `%<` and `>%` can be used to comment out comments of the form `%` ... `%`, but not vice versa.

Comments can be inserted anywhere a space is allowed.

3.3 Identifiers and other lexical matters

In this section the lexical structure of ML is defined. The details pertaining to the HOL logic are given in Section 11.1.

3.3.1 Identifiers

The structure of ML identifiers is partly programmable. The default is that an identifier can be of two forms:

(i) A sequence of alphanumerics starting with a letter, where the default structure of an alphanumeric is that it is a letter, a digit, a prime (`'`) or an underbar (`_`).

(ii) A special symbol chosen from the following list

```
**    ++    <--   <->   -->    ---    ><    >>    >=    <==   <=>
===   ==>   \/    //    /\     !?     !!    !\    ?!    ??    ?\
:=    <>    <-    <<    <=     --     ->    =>    ==
```

A letter is a member of the list:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

ML is case-sensitive: upper and lower case letters are considered to be different.

The function

```
new_letter : string -> void
```

makes a new character behave like a letter. (The unique object of ML type `void` is the value of the expression `()`.) For example:

```
new_letter `+`;;
```

will make `+int` and `foo+bar` become allowable identifiers. Failure occurs if the argument string is longer than one character.

The function

```
is_letter : string -> bool
```

tests if a unit string is a letter; it fails if the string has more than one character.

Alphanumerics are, by default, letters or digits. A digit is one of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`. The function

```
new_alphanum : string -> void
```

makes a new character behave like an alphanumeric. For example:

```
new_alphanum `+`;;
```

will make `foo+bar` and `foo+` become allowable names (but not `+bar`). Failure occurs if the argument string is longer than one character.

The function

```
is_alphanum : string -> bool
```

tests if a unit string is an alphanumeric; it fails if the string has more than one character.

It is a consequence of the way lexical analysis is implemented that any initial subsequence of a special symbol is also a special symbol (e.g. since `==>` is a special symbol, so are `==` and `=`). The function

```
new_special_symbol : string -> bool
```

makes the argument string, and all its initial substrings, special symbols. It fails if

(i) the argument string is a single character, or

(ii) it starts with a letter or alphanumeric.

For example:

```
new_special_symbol `===>>>`;;
```

makes `===>>>`, `===>>`, `===>` and `===` into new special symbols (`==` is already a special symbol because `==>` is).

The function

```
special_symbols : void -> string list
```

returns the list of currently declared special symbols.

3.3.2 Constant expressions

The ML constant expressions (*ce*'s) are:

1. Integers—i.e. strings of the digits `0,1...,9`.
2. Truth values `true` and `false`.
3. Strings consist of any sequence of characters surrounded by string quotes (‘), e.g. ‘`This is a single string`‘. In any string the occurrence of `\`*x* has the following meanings for different *x*'s:

   `\0` = ten spaces
   `\n` = `n` spaces (0<`n`<10)
   `\S` = one space
   `\R` = return
   `\L` = line-feed
   `\T` = tab
   `\F` = form feed
   `\x` = `x` taken literally otherwise (e.g. `\‘` to include quotes in a string)
4. The expression `()`, called *empty*, which evaluates to the unique object of ML type `void`.
5. expressions of the form *con* where *con* is a 0-ary constructor or `(`*con ce*`)` where *con* is a constructor and *ce* a constant expression.

The following function changes the string escape character (\).

```
set_string_escape : int -> int
```

This changes the escape character to be the character with the supplied ascii code (initially this is `92`, viz. ‘\‘). The old escape character is returned.

3.3.3 Prefixes and infixes

The ML *prefixes* *px* and *infixes* *ix* are given by:

px `::= not | - | do`
ix `::= * | / | + | - | . | @ | = | < | > | & | or | ,`

In addition, any identifier (and certain single characters) can be made into an infix. Such user-defined infixes bind more tightly than `...=>...|...` but more weakly than `or`. All of them have the same power binding and associate to the left.

Except for `&` and `or`, each infix *ix* (or prefix *px*) has correlated with it a special identifier `$`*ix* (or `$`*px*) which is bound to the associated function. For example, the identifier `$+` is

bound to the addition function, and `$@` to the list-append function (see Chapter 6 for the meaning of `$`-ed infixes). This is useful for passing functions as arguments; for example, f `$@` applies f to the append function.

See the descriptions of the functions `ml_paired_infix` and `ml_curried_infix` in Section 6.1 for details of how to give an identifier (or admissible special symbol) infix status.

Chapter 4

Semantics of ML

The evaluation of all ML constructs takes place in the context of an *environment* and a *store*. The environment specifies what the variables and constructors in use denote. Variables may be bound either to *values* or to *locations*. The contents of locations—which must be values—are specified in the *store*. If a variable is bound to a location then (and only then) is it *assignable*. Thus bindings are held in the environment, whereas location contents are held in the store. Constructors may only be bound to values (constructor constants or constructor functions) and this binding occurs when they are declared in a concrete type definition (see Section 5.5).

The evaluation of ML constructs may either *succeed* or *fail*. In the case of success:

1. The evaluation of a declaration, d say, changes the bindings in the environment of the identifiers declared in d. If d is at top-level, then the scope of the binding is everything following d. In d `in` e the scope of d is the evaluation of e, and so when this is finished the environment reverts to its original state (see Section 4.1).

2. The evaluation of an expression yields a value: the value of the expression (see Section 4.2).

If an assignment is done during an evaluation, then the store will be changed—we shall refer to these changes as *side effects* of the evaluation.

If the evaluation of a construct fails, then failure is signalled, and a string is passed to the context which invoked the evaluation. This string is called the *failure string*, and it normally indicates the cause of the failure. During evaluation, failures may be generated either *implicitly* by certain error conditions, or *explicitly* by the construct `failwith` e (which fails with e's value as failure string). For example, the evaluation of the expression `1/0` fails implicitly with failure string `'div'`, whilst that of `failwith` *'str'* fails explicitly with failure string *'str'*. We shall say two evaluations fail *similarly* if they both fail with the same failure string. For example, the evaluation of `1/0` and `failwith 'div'` fail similarly. Side effects are not undone by failures.

If during the evaluation of a construct a failure is generated, then unless the construct is a failure trap (i.e. an expression built from `?` and/or `!`) the evaluation of the construct itself fails similarly. Thus failures propagate up until trapped, or reaching top level. For example, when evaluating `(1/0)+1000`, the expression `1/0` is first evaluated, and the failure which this evaluation generates causes the evaluation of the whole expression (viz.

`(1/0)+1000`) to fail with `'div'`. On the other hand, the evaluation of `(1/0)?1000` traps the failure generated by the evaluation of `1/0`, and succeeds with value `1000`. (In general, the evaluation of e_1`?`e_2 proceeds by first evaluating e_1, and if this succeeds with value E, then E is returned as the value of e_1`?`e_2; however, if e_1 fails, then the result of evaluating e_1`?`e_2 is determined by evaluating e_2).

In describing evaluations, when we say that we *pass control* to a construct, we mean that the outcome of the evaluation is to be the outcome of evaluating the construct. For example, if when evaluating e_1`?`e_2 the evaluation of e_1 fails, then we pass control to e_2.

Expressions and patterns can be optionally decorated with types by writing `:`*ty* after them (e.g. `[]:int list`). The effect of this is to force the type checker to assign an instance of the asserted type to the construct; this is useful as a way of constraining types more than the type checker would otherwise (i.e. more than context demands), and it can also serve as helpful documentation. Details of types and type checking are given in Chapter 5, and will be ignored in describing the evaluation of ML constructs in the rest of this section.

If we omit types, precedence information and those constructs which are equivalent to others, then the syntax of ML can be summarised by:

d `::=` `let` *b* | `letref` *b* | `letrec` *b*

b `::=` *p*`=`*e* | `var` $p_1 p_2 \ldots p_n$`=` *e* | b_1 `and` $b_2 \ldots$ `and` b_n

p `::=` `_` | *ce* | *var* | *con p* | p_1`.`p_2 | p_1`,`p_2 | `[]` | `[`p_1`;`$p_2 \ldots$ `;`p_n`]`

e `::=` *ce* | *var* | *con* | $e_1 e_2$

| *px e* | e_1 *ix* e_2 | *p*`:=`*e* | `failwith` *e*

| `if` e_1 `{then|loop}` e_1'
`{if` e_2 `{then|loop}` e_2'
⋮
`if` e_n `{then|loop}` e_n'`}` `{{else|loop}` e'`}`

| *e* `{??|!!}` $e_1 e_1'$
`{??|!!}` $e_2 e_2'$
⋮
`{??|!!}` $e_n e_n'$ `{{?|!|?\`*id*`|!\`*id*`}` e'`}`

| `while` e_1 `do` e_2
| e_1`;`$e_2 \ldots$ `;`e_n | `[]` | `[`e_1`;`$e_2 \ldots$ `;`e_n`]` | *d* `in` *e*
| `\`$p_1 p_2 \ldots p_n$`.` *e*
| `fun` p_1 `.` e_1 | p_2 `.` e_2 | ... | p_n `.` e_n

4.1 Declarations

Any declaration must be one of the three kinds: `let` b, `letref` b or `letrec` b, where b is a binding. Each such declaration is evaluated by first evaluating the binding b to produce a (possibly empty) set of variable-value pairs, and then extending the environment (in a manner determined by the kind of declaration) so that each variable in this set of pairs denotes its corresponding value. The evaluation of bindings is described below in Section 4.1.1.

1. Evaluating `let` b declares the variables specified in b to be an ordinary (i.e. non assignable) variable, and binds (in the environment) each one to the corresponding value produced by evaluating b. To understand what are the variables defined in a declaration may require some knowledge about the environment. For example, a declaration `let f` x `=` e declares x if `f` is a constructor and declares `f` as the function `(\`x `.` e`)` otherwise.

2. Evaluating `letref` b declares the variables specified in b to be assignable and thus binds (in the environment) each one to a new location, whose contents (in the store) is set to the corresponding value. The effect of subsequent assignments to the variables will be to change the contents of the locations they are bound to. Bindings (in the environment) of variables to locations can only be changed by evaluating another declaration to supersede the original one.

3. Evaluating `letrec` b is similar to evaluating `let` b except that:

   (a) The binding b in `letrec` b must consist only of function definitions.

   (b) These functions are made mutually recursive.

   For example, consider:

   - `let f n = if n=0 then 1 else n*f(n-1)`
   - `letrec f n = if n=0 then 1 else n*f(n-1)`

   The meaning of `f` defined by the first case depends on whatever `f` is bound before the declaration is evaluated, whilst the meaning of `f` defined by the second case is independent of this (and is the factorial function).

4.1.1 The evaluation of bindings

There are three kinds of variable binding each of which, when evaluated, produces a set of variable-value pairs (or fails):

1. *Simple bindings*
   which have the form p`=`e where p is a pattern and e an expression.

2. *Function definitions*
   which have the form $id\ p_1 \ldots p_n$ `=` e. This is just an abbreviation for the simple binding id `=` $\backslash p_1 \ldots p_n .\ e$.

3. *Multiple bindings*
   which have the form b_1 `and` $b_2 \ldots$ `and` b_n where $b_1, b_2 \ldots, b_n$ are simple bindings or function definitions. As a function definition is just an abbreviation for a certain simple binding, each b_i $(0 < i < n + 1)$ either is, or is an abbreviation for, some simple binding $v_i = e_i$. The multiple binding b_1 `and` $b_2 \ldots$ `and` b_n then abbreviates p_1`,`$p_2 \ldots$`,`p_n `=` e_1`,`$e_2 \ldots e_n$, which is a simple binding.

As function definitions and multiple bindings are abbreviations for simple bindings we need only describe the evaluation of the latter.

A simple binding p`=`e is evaluated by first evaluating e to obtain a value E (if the evaluation fails then the evaluation of p`=`e fails similarly). Next the pattern p is *matched* with E to see if they have the same form (precise details are given in Section 4.1.2). If so, then to each identifier in p there is a corresponding component of E. The evaluation of p`=`e then returns the set of each identifier paired with its corresponding component. If p and E don't match then the evaluation of p`=`e fails with failure string '*pattern*'.

4.1.2 Matching patterns and expression values

When a pattern p is matched with a value E, either the match succeeds and a set of identifier-value pairs is returned (each identifier in p being paired with the corresponding component of E), or the match fails. We describe, by cases on p, the conditions for p to match E and the sets of pairs returned:

- `_`
  Always matches E. The empty set of pairs is returned.

- *ce*
  Matches E only if *ce* is equal to E. The empty set of pairs is returned.

- *var*
  Always matches E. The set consisting of *var* paired with E is returned.

- *con p*
  If E is not a value built with constructor *con*, the match fails. Otherwise E = *con*`(`E'`)` and the result is obtained by matching p with E'.

- p_1`.`p_2
  E must be a non-empty list E_1`.`E_2 such that p_1 matches E_1 and p_2 matches E_2. The union of the sets of pairs returned from matching p_1 with E_1 and p_2 with E_2 is returned.

- p_1 `,`p_2
  E must be a pair E_1`,`E_2 such that p_1 matches E_1 and p_2 matches E_2. The union of the sets of pairs returned from matching p_1 with E_1 and p_2 with E_2 is returned.

- `[`p_1 `;`p_2 `...;`p_n`]`
  E must be a list `[`E_1 `;`E_2 `... ;`E_n`]` of length n such that for each i (where $0 < i < n + 1$) p_i matches E_i. The union of the sets of pairs returned by matching p_i with E_i is produced.

Thus if p matches E, then p and E have a similar 'shape', and each identifier in p corresponds to some component of E (namely that component paired with the identifier in the set returned by the match).

Here are some examples:

1. `[`x`;`y`;`z`]` matches `[1;2;3]` with x, y and z corresponding to `1`, `2` and `3` respectively.
2. `[`x`;`y`;`z`]` doesn't match `[1;2]`
3. x`.`y matches `[1;2;3]` with x and y corresponding to `1` and `[2;3]` respectively.
4. x`.`y doesn't match `1` or `[]`.
5. x`,`y matches `1,2` with x and y corresponding to `1` and `2` respectively.
6. x`,`y doesn't match `[1;2]`.
7. `(`x`,`y`),[(`z`.`w`);_]` matches `(1,2),[[3;4;5];[6;7]]` with x, y, z and w corresponding to `1`, `2`, `3` and `[4;5]` respectively.

4.2 Expressions

If the evaluation of an expression terminates, then either it succeeds with some value, or it fails; in either case assignments performed during the evaluation may cause side effects. If the evaluation succeeds with some value we shall say that value is *returned*.

We shall describe the evaluation of expressions by considering the various cases, in the order in which they are listed in the syntax equations.

- *ce*
  The appropriate constant value is returned.

- *var*
  The value associated with *var* is returned. If *var* is ordinary, then the value returned is the value bound to *var* in the environment. If *var* is assignable, then the value returned is the contents of the location to which *var* is bound.

- *con*
  The value associated with *con* is returned. (see Section 5.5)

- e_1 e_2
  e_1 and e_2 are evaluated and the result of applying the value of e_1 (which must be a function) to that of e_2 is returned. Due to optimisations in the ML compiler, the order of evaluation may vary.

- px e
  e is evaluated and then the result of applying px to the value of e is returned. `-`e and `not` e have the obvious meanings; `do` e evaluates e for its side effects and then returns void.

- e_1 ix e_2
  e_1`&`e_2 is equivalent to `if` e_1 `then` e_2 `else false` (so sometimes only e_1 needs be evaluated to evaluate e_1`&`e_2)

  e_1 `or` e_2 is equivalent to `if` e_1 `then true else` e_2 (so sometimes only e_1 needs to be evaluated to evaluate e_1 `or` e_2)

  Except when ix is `&` or `or`, e_1 and e_2 are evaluated in that order, and the result of applying ix to their two values returned.

  e_1`,`e_2 returns the pair with first component the value of e_1, and second component the value of e_2; the meaning of the other infixes are given in Chapter 6.

- p`:=`e
  Every variable in p must be assignable and bound to some location in the environment. The effect of the assignment is to update the contents of these locations (in the store) with the values corresponding to the variables produced by evaluating the binding p`=`e (see Section 4.1.1). If the evaluation of e fails, then no updating of locations occurs, and the assignment fails similarly. If the matching to p fails, then the assignment fails with '*pattern*'. The value of p`:=`e is the value of e.

- `failwith` e
  e is evaluated and then a failure with e's value (which must be a string) is generated.

- `if` e_1 `{then|loop}` e'_1
  `{if` e_2 `{then|loop}` e'_2
  ⋮
  `if` e_n `{then|loop}` e'_n`}` `{{else|loop}` e'`}`

  e_1`,`e_2`, ... ,`e_n are evaluated in turn until one of them, e_m say, returns true (each e_i (for $0 < i < n+1$) must return either true or false). When the phrase following e_m is `then` e'_m control is passed to e'_m; however when the phrase is `loop` e'_m, then e'_m

is evaluated for its side effects, and then control is passed back to the beginning of the whole expression again (i.e. to the beginning of `if` $e_1 \ldots$).

In the case that all of e_1`,`$e_2 \ldots$ `,`e_n return false, and there is a phrase following e'_n, then if this is `else` e' control is passed to e', whilst if it is `loop` e' then e' is evaluated for its side effects and control is then passed back to the beginning of the whole expression again.

In the case that all of $e_1, \ldots, e_n$ return false, but no phrase follows e'_n—i.e. the option `{{else|loop}` e'`}` is absent—then empty (the unique value of type `void`) is returned.

- e `{??|!!}` $e_1 e'_1$
  `{??|!!}` $e_2 e'_2$
  $\vdots$
  `{??|!!}` $e_n e'_n$ `{{?|!|?\`id`|!\`id`}` e'`}`

  e is evaluated and if this succeeds its value is returned.

  In the case that e fails, with failure string str say, then each of e_1`,`$e_2 \ldots$ `,`e_n are evaluated in turn until one of them, e_m say, returns a string list containing str (each e_i (for $0 < i < n+1$) must return a string list). If `??` immediately precedes e_m, then control is passed to e'_m; however if `!!` precedes it, then e'_m is evaluated and control is passed back to the beginning of the whole expression (i.e. to the beginning of e `{??|!!} ...`).

  If none of e_1`,`$e_2 \ldots$ `,`e_n returns a string list containing str, and nothing follows e'_n (i.e. the option `{{ ... }\` e'`}` is absent), then the whole expressions fails with str (i.e. fails similarly to e).

  If none of $e_1, e_2 \ldots, e_n$ produces a string list containing str, and `?\` e' follows e'_n, then control is passed to e'. But if `!\` e' follows e'_n, then e' is evaluated, and control is passed back to the beginning of the whole expression.

  If `?\`id e' or `!\`id e' follows e'_n, then e' is evaluated in an environment in which id is bound to the failure string str (i.e. an evaluation equivalent to `let` id`=`str `in` e' is done), and then depending on whether it was `?\` or `!\` that occurred, the value of e' is returned or control is passed back to the beginning of the whole expression respectively.

- `while` e_1 `do` e_2
  The evaluation of this expression always returns `()` when it succeeds. e_1 is evaluated first. If the result is false the evaluation stops and returns `()`. If the result is true e_2 is evaluated and then the whole construct again.

- e_1`;`e_2 `...` `;`e_n
  e_1`,`e_2 `...` `,`e_n are evaluated in that order, and the value of e_n is returned.

- `[`e_1`;`e_2 `...` `;`e_n`]`
  $e_1, e_2, \ldots, e_n$ are evaluated in that order and the list of their values returned. `[]` evaluates to the empty list.

- d `in` e
  d is evaluated, and then e is evaluated in the extended environment and its value returned. The declaration d is local to e, so that after the evaluation of e, the former environment is restored.

- `\`$p_1 p_2 \ldots p_n$`.` e
  The evaluation of \-expressions always succeeds and yields a function value. The environment in which the evaluation occurs (i.e. in which the function value is created) is called the *definition environment*.

  1. Simple \-expressions: `\`p`.` e
     `\`p`.` e evaluates to that function which when applied to some argument yields the result of evaluating e in the current (i.e. application time) store, and in the environment obtained from the definition environment by binding any variables in p to the corresponding components of the argument (see 3.1.1).
  2. Compound \-expressions: `\`$p_1 p_2 \ldots p_n$`.` e
     A \-expression with more than one parameter is curried i.e. `\`$p_1 p_2 \ldots p_n$`.` e is exactly equivalent to `\`p_1`.(\`p_2 `...` `.\`p_n`.` e`)` ... whose meaning is given by 1. above.

  Thus the free variables in a function keep the same binding they had in the definition environment. So if a free variable is non-assignable in that environment, then its value is fixed to the value it has there. On the other hand, if a free variable is assignable in the definition environment, then it will be bound to a location. Although that binding is fixed, the contents of the location in the store is not, and can be subsequently changed with assignments.

- `fun` p_1 `.` e_1 `|` p_2 `.` e_2 `|` ... `|` p_n `.` e_n This expression evaluates to that function which when applied to some value E, matches each p_i in turn to E until one succeeds. If say p_i matches E, then e_i is evaluated in the environment obtained from the definition environment by binding any variable in p_i to the corresponding components of E. If no match succeeds, the evaluation fails with '*pattern*'.

Chapter 5

ML Types

So far, little mention has been made of types. For ML in its original role as the meta-language for proof in LCF, the importance of strict type checking was principally to ensure that every computed value of the type representing theorems was indeed a theorem. This use of type checking was due to Milner, and the idea is equally vital in HOL. The same effect could probably have been achieved by run-time type checking, but compile-time type checking was adopted instead, in the design of ML. This was partly for the considerable debugging aid that it provides; partly for efficient execution; and partly to explore the possibility of combining polymorphism with type checking.[1]

The ML type checking system is implemented in such a way that, although the user may occasionally (either for documentation or as a constraint) ascribe a type to an ML expression or pattern, it is hardly ever necessary to do so. The user of ML will almost always be content with the types ascribed and presented by the type checker, which checks every top-level phrase before it is evaluated. (The type checker may sometimes find a more general type assignment than expected.)

5.1 Types and objects

Every data object in ML possesses a type. Such an object may possess many types, in which case it is said to be *polymorphic* and possesses a *polytype*—i.e. a type containing type variables (for which we use a sequence of asterisks possibly followed by an identifier or integer)—and moreover it possesses all types which are *instances* of its polytype, formed by substituting types for zero or more type variables in the polytype. A type containing no type variables is a *monotype.*

We saw several examples of types in Chapter 2. To understand the following syntax, note that `list` is a postfixed unary (one-argument) type constructor (thereafter abbreviated to *tycon*). The user may introduce new n-argument type constructors. A binary type operator `directory`, for example, can be introduced. The following type expressions will then be types of different kinds of `directory`:

- `(string, int)directory`

[1] Milner's idea was that there are many operations (such as list mapping functions, functional composition, etc.) which work at an infinity of types, so their types should therefore somehow be parameterized—but without having to mention the particular type intended at each of their uses.

- `(int, int->int)directory`

The user may even deal with lists of directories, with the type

- `(int, bool)directory list`

5.1.1 The syntax of types

The syntax of ML types is:

Types *ty*

ty	`::=` *sty*	Standard (non-infix) type
	`|` *ty* `#` *ty*	R Cartesian product
	`|` *ty* `+` *ty*	R Disjoint sum
	`|` *ty* `->` *ty*	R Function type

Standard types *sty*

sty	`::= void | int | bool | string`	Basic types
	`|` *vty*	Type variable
	`|` *tycon*	Type abbreviation (see Section 5.4)
	`|` *tycon*	Nullary concrete type (see Section 5.5)
	`|` *tyarg tycon*	L Concrete type
	`|` *tycon*	Nullary abstract type
	`|` *tyarg tycon*	L Abstract type (see Section 5.6)
	`|` `(`*ty*`)`	

Type arguments *tyarg*

tyarg	`::=` *sty*	Single type argument
	`|` `(`*ty*`,...,`*ty*`)`	One or more type arguments

Type variables *vty*

vty `::=` `*` `|` `**` `|` `...`
`|` `*`*id* `|` `**`*id* `|` `...`
`|` `*0` `|` `**0` `|` `... | *1 | **1 |` `...`

Type abbreviations are introduced by a '`lettype`' declaration (see Section 5.4 below) which allows an identifier to abbreviate an arbitrary monotype. By contrast, a concrete type consists of an identifier (a type operator introduced by a '`type`' or '`rectype`' declaration; see Section 5.5 below) postfixed to zero or more type arguments. An abstract type likewise consists of an identifier (introduced by an '`abstype` or '`absrectype`' declaration; see Section 5.6) postfixed to zero or more type arguments. Two or more arguments must be separated by commas and enclosed by parentheses. The type operator `list` is a

predeclared unary type operator; and `#`, `+` and `->` may be regarded as infix forms of three predeclared binary type operators.

For an object to possess[2] a type means the following: For basic types, all integers possess `int`, both booleans possess `bool`, all strings possess `string`, etc. The only object possessing `void` is that denoted by `()` in ML. For a type abbreviation *tycon*, an object possesses *tycon* (during execution of phrases in the scope of the declaration of *tycon*) if and only if it possesses the type which *tycon* abbreviates.

For compound monotypes,

1. The type *ty* `list` is possessed by any list of objects, all of which possess type *ty* (so that the empty list possesses type *ty* `list` for every *ty*).

2. The type ty_1`#`ty_2 is possessed by any pair of objects possessing the types ty_1 and ty_2, respectively.

3. The type ty_1`+`ty_2 is possessed by the left-injection of any object possessing ty_1, and by the right-injection of any object possessing ty_2. These injections are denoted by the ML function identifiers `inl : * -> * + **` and `inr : ** -> * + **` (see Chapter 6).

4. A function possesses type ty_1`->`ty_2 if whenever its argument possesses type ty_1, its result (if defined) possesses type ty_2. (This is not an exact description; for example, a function defined in ML with non-local variables may possess this type even though some assumption about the types of the values of these non-local variables is necessary for the above condition to hold. The constraints on programs listed below ensure that the non-locals will always have the right types).

5. An object possesses the concrete type *tyarg tycon* if and only if it is constructed with a constructor *con* from an object which possess the *tyarg* instance of some type *ty* and *con* `of` *ty* appears in the declaration of *tycon*.

6. An object possesses the abstract type *tyarg id* if and only if it is represented (via the abstract type representation) by an object possessing the *tyarg* instance of the right-hand side of the declaration of *id*.

Finally, an object possesses a polytype *ty* if and only if it possesses all monotypes which are substitution instances of *ty*.

5.2 Typing of ML phrases

We now explain the constraints used by the type checker in ascribing types to ML expressions, patterns and declarations.

[2] We shall talk of objects *possessing* types and phrases *having* types, to emphasise the distinction.

The significance of expression e having type ty is that the value of e (if evaluation terminates successfully) possesses type ty. As consequences of the well-typing constraints listed below, it is impossible for example to apply a non-function to an argument, or to form a list of objects of different types, or (as mentioned earlier) to compute an object of the type corresponding to theorems which is not a theorem.

The type ascribed to a phrase depends in general on the entire surrounding ML program. In the case of top-level expressions and declarations, however, the type ascribed depends only on preceding top-level phrases. Thus you know that types ascribed at top-level are not subject to further constraint.

Before each top-level phrase is executed, types are ascribed to all its sub-expressions, sub-declarations and sub-patterns according to the following rules. Most of the rules are fairly natural; those which are less so are discussed later. You are only presented with the types of top-level phrases; the types of sub-phrases will hardly ever concern you.

Before giving the list of constraints, let us discuss an example which illustrates some important points. To map a function over a list we may define the polymorphic function `map` recursively as follows (where we have used an explicit abstraction, rather than `letrec map f l = ...`, to make the typing clearer):

```
letrec map = \f.\l. null l => [] | f(hd l).map f(tl l) ;;
```

From this declaration the type checker will infer a *generic* type for `map`. By 'generic' we mean that each later occurrence of `map` will be ascribed a type which is a substitution instance of the generic type.

Now the free identifiers in this declaration are `null`, `hd` and `$.`, which are ML primitives whose *generic* (poly)types are `* list -> bool`, `* list -> *`, and `* # * list -> * list` respectively. The first constraint used by the type checker is that the occurrences of these identifiers in the declaration are ascribed instances of their generic types. Other constraints which the type checker will use to determine the type of `map` are:

- All occurrences of a `\`-bound variable receive the same type.
- Each arm of a conditional receives the same type, and the condition receives type `bool`.
- In each application e `=` $(e_1 e_2)$, if e_2 receives ty and e receives ty' then e_1 receives ty`->`ty'.
- In each abstraction e `=` `\`v`.`e_1, if v receives ty and e_1 receives ty' then e receives ty`->`ty'.
- In a `letrec` declaration, all free occurrences of the declared variable receive the same type.

Now the type checker will ascribe the type `(*->**)->* list->** list` to `map`. This is in fact the most general type consistent with the constraints mentioned. Moreover, it can be shown that any instance of this type also allows the constraints to be satisfied; this is what allows us to claim that the declaration is indeed polymorphic.

In the following constraint list, we say p has ty to indicate that the phrase p is ascribed a type ty which satisfies the stated conditions. We use x, p, e, d to stand for variables, patterns, expressions and declarations respectively.

1. Constants

   (a) `()` has `void`

   (b) `0` has `int`, `1` has `int`, ...

   (c) `true` has `bool`, `false` has `bool`

   (d) `'...'` has `string`

   (e) *con* and *con ce* have an instance of *tyarg tycon* where *tycon* is the type of which *con* is a constructor (see Section 5.6).

2. Variable and constructors
   The constraints described here are discussed in Section 5.3 below.

   (a) If x is a variable bound by `\`, `fun` or `letref`, then `x` is ascribed the same type as its binding occurrence. In the case of `letref`, this must be monotype if

      i. the `letref` is top-level or

      ii. an assignment to x occurs within a `\`-expression within its scope.

   (b) If x is bound by `let` or `letrec`, then x has ty, where ty is an instance of the type of the binding occurrence of x (i.e. the *generic* type of x), in which type variables occurring in the types of current `\`-bound or `letref`-bound identifiers are not instantiated.

   (c) If x is not bound in the program (in which case it must be an ML primitive), then x has ty, where ty is an instance of the type of x given in Chapter 6.

   (d) If x is a constructor, x has ty where ty is an instance of the type ascribed to x in the concrete type declaration where it was bound.

3. Patterns
   Cases for a pattern p:

   - `_`
     p has ty, where ty is any type.

- *con* p'
  p has an instance of *tyarg tycon* where *tycon* is the type of which *con* is a constructor.
- p_1`:`ty
  p_1 and p have an instance of ty.
- p_1`,`p_2
  If p_1 has ty_1 and p_2 has ty_2, then p has ty_1`#`ty_2.
- p_1`.`p_2
  If p_1 has ty then p_2 and p have ty `list`.
- `[`p_1`; ... ;`p_n`]`
  For some ty, each p_i has ty and p has ty `list`.

4. Expressions
   Cases for an expression e (not a constant or identifier):

- $e_1 e_2$
  If e_2 has ty and e has ty' then e_1 has ty`->`ty'.
- e_1`:`ty
  e_1 and e have an instance of ty.
- px e_1
  Treated as `$`px`(`e_1`)` when px is a prefix. If e is `-`e_1, then e and e_1 have `int`.
- e_1 ix e_2
  Treated as `$`ix`(`e_1`,`e_2`)` if ix is introduced with `ml_paired_infix`, and as `(`ix e_1 e_2`)` if ix is introduced by `ml_curried_infix`. If e is `(`e_1 `&` e_2`)` or `(`e_1 `or` e_2`)`, then e, e_1 and e_2 have `bool`.
- e_1`,`e_2
  If e_1 has ty_1 and e_2 has ty_2 then e has ty_1`#`ty_2.
- p`:=`e_1
  For some ty, p, e_1 and e all have ty.
- `failwith` e_1
  e_1 has string, and e has any type.
- `if` e_1 `then` e'_1 ... `if` e_n `then` e'_n `else` e'
  Each $e_i (0 < i < n+1)$ has bool, and `-` for some $ty - e$, each e'_i and e' all have ty. However, this constraint does not apply to an e'_i preceded by `loop` in place of `then`, nor to e' preceded by `loop` in place of `else`. If e' is absent, then ty `= void` .

- e'_0 `??` e_1 e'_1 ... `??` e_n e'_n `?{\`x`}` e'
  Each e_i $(0 < i < n+1)$ has `string list`, and `-` for some ty `-` e, e'_0, each e'_i and e' all have ty. However, this constraint does not apply to an e'_i preceded by `!!` in place of `??` nor to e' preceded by `!` in place of `?`. If `\`x is present, x has `string`.

- `while` e_1 `do` e_2
  e_1 must have type `bool`, e_2 may have any type and e has type `void`.

- e_1`;` ... `;`e_n
  If e_n has ty then e has ty.

- `[`e_1`;` ... `;`e_n`]`
  For some ty, each e_i has ty and e has ty `list`.

- d `in` e_1
  If e_1 has ty then e has ty. See also (2) above. If d is a type definition (see Sections 5.5 and 5.6) then ty must contain no type defined in d.

- `\`p`.`e_1
  If p has ty and e_1 has ty' then e has ty`->`ty'. See also (2) above.

- `fun` p_1`.`e_1 `|` ... `|` p_n`.`e_n
  If each p_i has ty and each e_i has ty' then e has ty`->`ty'. See also (2) above.

5. Declarations

   (a) Each binding x $p_1 \ldots p_n$ `=` e is treated as x `=` `\`p_1`.` ... `\`p_n`.` e.

   (b) `let` p_1 `=` e_1 `and` ... `and` p_n `=` e_n is treated as `let` p_1`,...,`p_n `=` e_1`,...,`e_n (similarly for `letrec`, `letref`).

   (c) If d is `let` p `=` e, then `-` for some ty `-` d, p and e all have ty (similarly for `letref`). Note that e is not in the scope of the declaration.

   (d) If d is `letrec` x_1`,...,`x_n `=` e_1`,...,`e_n then `-` for some ty_i `-` x_i and e_i have ty_i, and d has ty_1`#...#`ty_n. (In addition, each free occurrence of x_i in e_1`, ... ,`e_n has ty_i, so that the type of recursive calls of x_i is the same as the declaring type).

5.3 Discussion of type constraints

We give here reasons for our constraints on the types ascribed to occurrences of identifiers. The reader may like to skip this section at first reading.

1. Consider constraint (2a) for \-bound identifiers. This constraint implies that the expression `let` x `=` e `in` e' may be well-typed even if the semantically-equivalent expression `let` f x `=` e' `in` f e is not, since in the former expression x may occur in e' with two incompatible types which are both instances of the declaring type. The greater constraint on f is associated with the fact that f may be applied to many different arguments during evaluation. To show the need for the constraint, suppose that it is replaced by the weaker constraint for `let`-bound identifiers, so that for example

   ```
   let f x = if x then 1+x else x(1)
   ```

   is a well-typed declaration of type `*->int`, in which the four occurrences of `x` receive types `*`, `bool`, `int`, `int->int` respectively. In the absence of an explicit argument for the abstraction, no constraint exists for the type of the binding occurrence of `x`. But, because `f` is `let`-bound, expressions such as `f(true)` and `f('dog')` are admissible in the scope of `f`, although their evaluation should result in either nonsense or *run-time* type-errors; one of our purposes is to preclude these.

   The only exception to this rule is for expressions of the form `(\`x`.` e'`)` e, which is treated exactly as `let` x`=`e `in` e'. Here we know the unique instance of type of the argument x, namely the type of e.

2. The analogous restriction for `letref`-bound identifiers is also due to the possibility that the identifier-value binding may change during evaluation (this time because of assignments). Consider the following:

   ```
   letref x = [] in
          (if e then do(x := 1.x) else do(x := [true]) ;
           hd x) ;;
   ```

   If `letref` were treated like `let`, this phrase would be well-typed and indeed have type `*` , despite the fact that the value returned is either `1` or `true`. So, calling the whole expression e, all manner of larger expressions involving e would be well-typed, even including e`(`e`)` !

3. Top level `letref`s must be monomorphic (see (2ai)) to avoid retrospective type constraints at top-level. If this restriction were removed the following would be allowed:

   ```
   letref x = [] ;;
       ⋮
       2.x ;;
   ```

But on type checking the last phrase, it would appear that the type of `x` at declaration should have been `int list`, not `* list`, and the types of intervening phrases may likewise need constraining.

4. To see the need for the exclusion of polymorphic non-local assignments (see (2aii)), consider this example in the HOL system (this example is originally due to Lockwood Morris). (The type `thm` is the type of theorems, as is explained later in Section 11.7.)

```
let store,fetch =
    letref x = [] in (\y. x:=[y]) , (\(). hd x ) ;;
store "T = F" ;;
let eureka :thm = fetch() ;;
```

Now suppose we lift our constraint. Then in the declaration, `x` has type `* list` throughout its (textual) scope, and `store`, `fetch` receive types `*->* list`, `**->*` respectively. In the two ensuing phrases they get respective types `form->form list`, `void->thm` (instances of their declaring types), and the value of `eureka` is a contradictory formula masquerading as a theorem!

The problem is that the type checker has no simple way of discovering the types of all values assigned to the polymorphic `x`, since these assignments may be invoked by calls of the function `store` outside the (textual) scope of `x`. This is not possible under our constraint.

However, polymorphic assignable identifiers are still useful: consider

```
let rev l =
     letref l,l' = l,[] in
       if null l then l' loop (l,l':= tl l, hd l.l') ;;
```

Such uses of assignable identifiers for iteration may be avoided given a suitable syntax for iteration, but assignable identifiers are useful for a totally different purpose—namely as 'own variables' shared between one or more functions (as in the store-fetch example). Our constraint of course requires them to be monomorphic; this is one of the few cases where the user occasionally needs to add an explicit type to a program.

5.4 Type abbreviations

The syntax of type abbreviation bindings *tab* is

tab `::=` id_1 `=` ty_1 `and` ... `and` id_n `=` ty_n

Then the declaration

```
lettype tab
```

in which each ty_i must be a monotype, built from basic types and previously defined types, allows you to introduce new names id_i for the types ty_i. Within the scope of the declaration the expression e:id_i behaves exactly like e:ty_i, and the type ty_i will always be printed as id_i.

One aspect of such type abbreviations should be emphasised. Suppose you declare

```
lettype rat = int # int ;;
```

for the rational numbers, and set up the standard operations on rationals. Within the scope of this declaration *any* expression of type `int#int` will be treated as though it had type `rat`, and this could be not only confusing but also incorrect (in which case it ought to cause a type failure). If you wish to introduce the type `rat`, isomorphic to `int#int` but not matching it for type checking purposes, then you should use abstract types.

5.5 Concrete types

Concrete type operators may be introduced by a special kind of declaration, in which type variables are used as dummy arguments (or formal parameters) of the operators, just as ordinary variables are so used in declaring functions. We need the syntax

vtyarg	::=	Empty dummy argument list
	| *vty*	One dummy argument
	| (*vty*, ... ,*vty*)	One or more dummy arguments

Then the syntax of concrete type bindings *cb* is

cb ::= *vtyarg tycon* = con_1 {of ty_1} | ... | con_n {of ty_n}
| cb_1 and ... and cb_n

where each $vtyarg_i$ must contain no type variable more than once, and all the type variables in ty_i must occur in *vtyarg*. A concrete type declaration takes the form

```
{type|rectype} cb
```

The declaration introduces a set of type constructors, and also a set of constructors. If `of` ty_i is absent then the value of con_i is the constant value con_i and its type is *vtyarg tycon*; otherwise the value of con_i is the function which maps v to con_i`(`v`)` and its type is ty_i `->` *vtyarg tycon*. The constructors have the same scope as the type constructors introduced by the declaration.

For non-top-level declarations a simple constraint ensures that a value of concrete type cannot exist except during the execution of phrases within the scope of the type declaration. In the expression

`{rec}type` ...$vtyarg_i$ id_i `=` ... `in` e

the type of e, and the types of any non-local assignments within b and e, must not involve any of the id_i.

5.6 Abstract types

As with concrete types, abstract type constructors may be introduced by a declaration in which type variables are used as dummy arguments (or formal parameters) of the operators. The syntax of abstract type bindings ab is

ab `::=` $vtyarg_1$ $tycon_1$ `=` ty_1 `and` ... `and` $vtyarg_n$ $tycon_n$ `=` ty_n `with` b

where each $vtyarg_i$ must contain no type variable more than once, and all the type variables in ty_i must occur in $vtyarg_i$. An abstract type declaration takes the form

`{abstype|absrectype}` ab

The declaration introduces a set of type operators, and also incorporates a normal binding b (treated like `let`) of ML identifiers. Throughout the scope of the abstract type declaration the type operators and ML identifiers are both available, but it is only within b that the *representation* of the type operators (as declared in terms of other operators) is available.

In an abstract type declaration

`abs{rec}type` $vtyarg_1$ id_1 `=` ty_1 `and` ... `and` $vtyarg_n$ id_n `=` ty_n `with` b

The sense in which the representation of each id_i is available only within b is as follows: the isomorphism between objects of types ty_i and $vtyarg_i$ id_i is available (only in b) via a pair of implicitly declared polymorphic functions

`abs_`id_i `:` ty_i `->` $vtyarg_i$ id_i `rep_`id_i `:` $vtyarg_i$ id_i `->` ty_i

which are to be used as coercions between the abstract types and their representations. Thus in the simple case

`abstype` a `=` ty `with` x `=` e' `in` e

the scope of a is e' and e, the scope of `abs_`a and `rep_`a is e', and the scope of x is e.

As an illustration, consider the definition of the type `rat` of rational numbers, represented by pairs of integers, together with operations `plus` and `times` and the conversion functions

```
inttorat : int->rat
rattoint : rat->int
```

Since `rat` is a nullary type operation, no type variables are involved, and `rat` can be defined by:

```
abstype rat = int#int
   with plus(x,y) = (abs_rat(x1*y2+x2*y1, x2*y2)
                     where x1,x2 = rep_rat x
                       and y1,y2 = rep_rat y    )
   and times(x,y) = (abs_rat(x1*y1, x2*y2)
                     where x1,x2 = rep_rat x
                       and y1,y2 = rep_rat y    )
   and inttorat n = abs_rat(n,1)
   and rattoint x = ((x1/x2)*x2=x1 => x1/x2 | failwith `rattoint`
                     where x1,x2 = rep_rat x    ) ;;
```

Most abstract type declarations are probably used at top-level, so that their scope is the remainder of the top-level program. But for non-top-level declarations, a simple constraint ensures that a value of abstract type cannot exist except during the execution of phrases within the scope of the type declaration. In the expression

`abs{rec}type` $\ldots vtyarg_i\ id_i$ `=` $ty_i \ldots$ `with` b `in` e

The type of e, and the types of any non-local assignments within b and e, must not involve any of the id_i.

Finally, in keeping with the abstract nature of objects of abstract type, the value of a top-level expression of abstract type is printed as a dash, `-`, as functional values are. Users who wish to 'see' such an object should declare a coercion function in the 'with' part of the type declaration, to yield a suitable concrete representation of the abstract objects.

Chapter 6

Primitive ML Identifier Bindings

The primitive ML identifier bindings are described in this chapter. Some useful derived functions are in Chapter 7. The primitive bindings are of two kinds:

- ordinary bindings;
- dollared bindings (which are preceded by `$`) having prefix or infix status.

The description of the ML value to which an identifier is bound is omitted if the semantics is clear from the identifier name and type given. For those functions whose application may fail, the failure string is the function identifier.

Predeclared identifiers are not regarded as constants of the language. As with all other ML identifiers, the user is free to rebind them, by `let`, `letref`, etc., but note that in the case of infix or prefix operators rebinding the dollared operator will affect even its non-dollared uses. Predeclared bindings are to be understood as if they had been bound by `let`, rather than by `letref`. In particular, therefore, none of them can be changed by assignment (except, of course, within the scope of a rebinding of the identifier by a `letref`-declaration).

6.1 Predeclared ordinary identifiers

```
fst  : * # ** -> *
snd  : * # ** -> **

null : * list -> bool
hd   : * list -> *
tl   : * list -> * list

inl  : * -> * + **
inr  : ** -> * + **
outl : * + ** -> *
outr : * + ** -> **
isl  : * + ** -> bool
```

The functions `hd` and `tl` fail if their argument is an empty list. The functions `outl` and `outr` fail if their arguments are not in the left or right summand, respectively. A function `isr` is not provided because it is just the complement of `isl`.

```
explode : string -> string list
implode : string list -> string
```

The function `explode` maps a string into the list of its single character strings in order. The function `implode` maps a list of single character strings (fails if any string is not of length one) into the string obtained by concatenating these characters. For example:

1

```
#explode `whosit`;;
[`w`; `h`; `o`; `s`; `i`; `t`] : string list

#implode [`c`;`a`;`t`];;
`cat` : string

#implode [`cd`;`ab`;`tu`];;
evaluation failed     implode
```

```
ascii      : int -> string
ascii_code : string -> int
```

The function `ascii` returns the one character string whose ASCII code corresponds to its argument, which must be non-negative and less than `128`. The function `ascii_code` returns the ASCII code of the first character of its argument, which must be a non-empty string.

```
string_of_int : int -> string
int_of_string : string -> int
```

These are bound to the obvious type coercion functions, with `int_of_string` failing if its argument is not a non-negative integer string.

```
ml_curried_infix : string -> void
ml_paired_infix  : string -> void
```

The functions `ml_curried_infix` and `ml_paired_infix` declare their argument strings to the ML parser as having *infix status*. Infixed functions can either be curried or take a pair as an argument. For example, after executing:

```
ml_paired_infix `plus`;;
let x plus y = x+y;;
```

`1 plus 2` is synonymous with `$plus(1,2)` and after executing

```
ml_curried_infix `plus` ;;
let x plus y = x+y ;;
```

`1 plus 2` is synonymous with `$plus 1 2`.[1]

[1]Only ordinary identifiers and certain single character, non-digit, non-layout strings can be used as infixes; infixing other strings may have unpredictable effects on the parser.

6.2 Predeclared dollared identifiers

The following prefix and infix operators are provided as primitives (where the dollar symbol is omitted from the table; the constants are `$do`, and so on):

```
do          : * -> void
not         : bool -> bool
*, /, +, -  : int # int -> int
>, <        : int # int -> bool
 =          : * # * -> bool
 @          : * list # * list -> * list
 .          : * # * list -> * list
```

Clarifying remarks:

- `$do` is equivalent to `\x.()`; `do` e evaluates e for its side effects.
- `/` returns the integer part of the result of a division, for example:

  ```
  $/(7,3)  =  7/3  =  2
  $/(-7,3) = -7/3  = -2
  ```

  The failure string for division by zero is `'div'`.
- `-` is the binary subtraction function. Negation (unary minus) is not available as a predeclared function of ML, only as a prefix operator. Of course, the user can define negation if he or she wishes—e.g. by

  ```
  let minus x = -x
  ```

- Not all dollared infix operators are included above: `$,` is not provided since it would be equivalent (as a function) to the identity on pairs, nor is `&` as it has no corresponding call-by-value function (because e `&` e' evaluates to `false` when e does even if evaluation of e' would fail to terminate), nor is `or` analogously.
- The period symbol '`.`' is an infixed Lisp cons:

  $$x\,.\,[x_1;\ \ldots\ ;x_n] = [x;x_1;\ \ldots\ ;x_n]$$

- `=` is bound to the expected predicate for an equality test at non-function types, but is necessarily rather weak, and may give surprising results, at function types. You can be sure that semantically (i.e. extensionally) different functions are not equal, and that semantically equivalent functions are equal when they originate from the same evaluation of the same textual occurrence of a function-denoting expression; for other cases the equality of functions is unreliable (i.e. implementation dependent). For example, after the top-level declarations

```
let f x = x+1 and g x = x+2;;

let f' = f and h x = f x and h' x = x+1;;
```

`f=f'` evaluates to `true` and `f=g` evaluates to `false`, but the truth values of `f=h`, `f=h'`, and `h=h'` are unreliable. Furthermore, after declaring

```
let plus = \x.\y.x+y;;

let f = plus 1 and g = plus 1;;
```

the truth value of `f=g` is also unreliable.

- `@` is a predeclared list concatenation operator; the symbol `@` has a special parser status and cannot be redeclared as a curried infix.

Chapter 7

General Purpose and List Processing Functions

This chapter describes a selection of commonly useful ML functions applicable to pairs, lists and other ML values. All the functions are definable in ML. Each function is documented by:

(i) Its name and type.

(ii) A brief description.

(iii) An ML declaration defining the function (note that this is not necessarily the definition used: some of the functions are coded directly in Lisp).

Functions preceded by `$` may be used as infix operators (without the `$`), or in normal prefix form or as arguments to other functions (with the `$`).

The functions usually fail with failure string equal to their name; sometimes, however, the failure string is the one generated by the subfunction that caused the failure.

7.1 General purpose functions and combinators

The standard primitive combinators are:

```
I : * -> *
K : * -> ** -> *
S : (* -> ** -> ***) -> (* -> **) -> * -> ***
```

Description

`I` $x = x$ `K` $x\ y = x$ `S` $f\ g\ x = f\ x\ (g\ x)$

Definition

```
let I x = x

let K x y = x

let S f g x = f x (g x)
```

The derived combinators KI (the dual of K), C (the permutator), W (the duplicator), B (the compositor) and CB (which is declared to be infix) have types:

```
KI : * -> ** -> **
C  : (* -> ** -> ***) -> ** -> * -> ***
W  : (* -> * -> **) -> * -> **
B  : (* -> **) -> (*** -> *) -> *** -> **
CB : (* -> **) -> (** -> ***) -> * -> ***
```

Description

KI $x\ y = y$ C $f\ x\ y = f\ y\ x$ B $f\ g\ x = f(g\ x)$ W $f\ x = f\ x\ x$

CB $f\ g\ x = g(f\ x)$

Definition

```
let KI = K I

let C f x y = f y x

let W f x = f x x

let B f g x = f(g x)

let (f CB g) x = g(f x)
```

The next group of functions are various useful infixed function-composition operators:

```
$o  : ((* -> **) # (*** -> *)) -> *** -> **
$#  : ((* -> **) # (*** -> ****)) -> (* # ***) -> (** # ****)
$Co : ((* -> ** -> ***) # (**** -> *)) -> ** -> **** -> ***
```

Description

$(f \circ g)\ x = f(g\ x)$

$(f \#\ g)(x, y) = (f\ x,\ g\ y)$

$(f$ Co $g)\ x\ y =$ C$(f \circ g)\ x\ y = f\ (g\ y)\ x$

Definition

```
ml_paired_infix `o`
let (f o g) x = f(g x)

ml_paired_infix `#`
let (f # g)(x,y) = (f x, g y)

ml_paired_infix `Co`
let (f Co g) x y = f (g y) x
```

The following two functions convert between curried and uncurried versions of a binary function.

```
curry   : ((* # **) -> ***) -> (* -> ** -> ***)
uncurry : (* -> ** -> ***) -> ((* # **) -> ***)
```

Description

`curry` $f\ x\ y = f(x,y)$ `uncurry` $f\ (x,y) = f\ x\ y$

Definition

```
let curry f x y = f (x,y)

let uncurry f (x,y) = f x y
```

The next function tests for failure.

```
can : (* -> **) -> * -> bool
```

Description

`can` $f\ x$ evaluates to `true` if the application of f to x succeeds; it evaluates to `false` if the evaluation fails.

Definition

```
let can f x = (f x; true) ? false
```

The next function iterates a function a fixed number of times.

```
funpow : int -> (* -> *) -> * -> *
```

Description

`funpow` $n\ f\ x$ applies f to x n-times: `funpow` $n\ f = f^n$

Definition

```
letrec funpow n f x = if n=0 then x else funpow (n-1) f (f x)
```

7.2 Miscellaneous list processing functions

The function `length` computes the length of a list.

```
length : * list -> int
```

Description

`length` $[x_1; \ldots; x_n] = n$

Definition

```
letrec length = fun [] . 0 | (_.l) . 1+(length l)
```

The function `append` concatenates lists; `@` is an uncurried and infixed version of `append`.

```
append : * list -> * list -> * list
```

Description

`append` $[x_1; \ldots; x_m]$ $[y_1; \ldots; y_n] = [x_1; \ldots; x_m; y_1; \ldots; y_n]$

Definition

```
letrec append l1 l2 = if null l1 then l2 else hd l1.append (tl l1) l2
```

The function `el` extracts a specified element from a list. It fails if the integer argument is less than 1 or greater than the length of the list.

```
el : int -> * list -> *
```

Description

`el` i $[x_1; \ldots; x_n] = x_i$

Definition

```
letrec el i l =
 if null l or i < 1 then failwith `el`
 else if i = 1 then hd l
 else el (i-1) (tl l)
```

The functions `last` and `butlast` compute the last element of a list and all but the last element of a list. Both fail if the argument list is empty.

```
last    : * list -> *
butlast : * list -> * list
```

Description

`last` $[x_1; \ldots; x_n] = x_n$

`butlast` $[x_1; \ldots; x_n] = [x_1; \ldots; x_{n-1}]$

Definition

```
letrec last l = last (tl l) ? hd l ? failwith `last`

letrec butlast l =
 if null (tl l) then [] else (hd l).(butlast(tl l)) ? failwith `butlast`
```

The next function makes a list consisting of a value replicated a specified number of times. It fails if the specified number is less than zero.

```
replicate : * -> int -> * list
```

Description

`replicate` x n evaluates to $[x;\ldots;x]$, a list of length n.

Definition

```
letrec replicate x n =
 if n < 0 then failwith `replicate`
 else if n = 0 then []
 else x . (replicate x (n-1))
```

7.3 List mapping and iterating functions

```
map : (* -> **) -> * list -> ** list
```

Description

`map` f l returns the list obtained by applying f to the elements of l in turn.

Definition

```
letrec map f l = if null l then [] else f(hd l). map f (tl l)
```

The following three functions are versions of 'reduce'.

```
itlist     : (* -> ** -> **) -> * list -> ** -> **
rev_itlist : (* -> ** -> **) -> * list -> ** -> **
end_itlist : (* -> * -> *) -> * list -> *
```

Description

`itlist` f `[`x_1`;`x_2`;`$\ldots$`;`x_n`]` x $= f\ x_1\ (f\ x_2\ (\ \ldots\ (f\ x_n\ x)\ \ldots\ ))$
$= ((f\ x_1) \circ (f\ x_2) \circ\ \ldots\ \circ (f\ x_n))\ x$

`rev_itlist` f `[`x_1`;`$\ldots$`;`x_{n-1}`;`x_n`]` x $= f\ x_n\ (f\ x_{n-1}\ (\ \ldots\ (f\ x_1\ x)\ \ldots\ ))$
$= ((f\ x_n) \circ (f\ x_{n-1}) \circ\ \ldots\ \circ (f\ x_1))\ x$

`end_itlist` f `[`x_1`;`x_2`;`$\ldots$`;`x_{n-1}`;`x_n`]` $= f\ x_1\ (f\ x_2\ (\ \ldots\ (f\ x_{n-1}\ x_n)\ \ldots\ ))$
$= ((f\ x_1) \circ (f\ x_2) \circ\ \ldots\ \circ (f\ x_{n-1}))\ x_n$

Definition

```
letrec itlist f l x =
 if null l then x else f (hd l) (itlist f (tl l) x)

letrec rev_itlist f l x =
 if null l then x else rev_itlist f (tl l) (f (hd l) x)

let end_itlist ff l =
 if null l then failwith `end_itlist`
 else (let last.rest = rev l in
       rev_itlist ff rest last)
```

or, equivalently:

```
letrec itlist f     = fun [] . I | (y.l) . \x. f y (itlist f l x)

letrec rev_itlist f = fun [] . I | (y.l) . \x. rev_itlist f l (f y x)
```

7.4 List searching functions

The functions described in this section search lists for elements with various properties. Those functions that return elements fail if no such element is found; those that return booleans never fail (`false` is returned if the element is not found).

```
find    : (* -> bool) -> * list -> *
tryfind : (* -> **) -> * list -> **
```

Description

`find` p l returns the first element of l which satisfies the predicate p; `tryfind` f l returns the result of applying f to the first member of l for which the application of f succeeds.

Definition

```
letrec find p    = fun []    . failwith `find`
                   |  (x.l) . if p x then x else find p l


letrec tryfind f = fun []    . failwith `tryfind`
                   |  (x.l) . (f x ? tryfind f l)
```

The next two functions are analogous to the quantifiers $\exists$ and $\forall$.

```
exists : (* -> bool) -> * list -> bool
forall : (* -> bool) -> * list -> bool
```

Description

`exists` p l applies p to the elements of l in order until one is found which satisfies p, or until the list is exhausted, returning `true` or `false` accordingly; `forall` is the dual.

Definition

```
let exists p l = can (find p) l

let forall p l = not(exists ($not o p) l)
```

The next function tests for membership of a list.

```
mem : * -> * list -> bool
```

Description

`mem` x l returns `true` if some element of l is equal to x, otherwise it returns `false`.

Definition

```
let mem = exists o (curry $=)
```

The following two functions are ML versions of Lisp's `assoc`.

```
assoc     : * -> (* # **) list -> (* # **)
rev_assoc : * -> (** # *) list -> (** # *)
```

Description

`assoc` x l searches a list l of pairs for one whose first component is equal to x, returning the first pair found as result; similarly, `rev_assoc` y l searches for a pair whose second component is equal to y. For example:

1

```
#assoc 2 [(1,4);(3,2);(2,5);(2,6)];;
(2, 5) : (int # int)

#rev_assoc 2 [(1,4);(3,2);(2,5);(2,6)];;
(3, 2) : (int # int)
```

Definition

```
let assoc x     = find (\(x',y). x=x')

let rev_assoc y = find (\(x,y'). y=y')
```

7.5 List transforming functions

The next function reverses a list:

```
rev : * list -> * list
```

Description

`rev` $[x_1;x_2;\ldots;x_n] = [x_n;\ldots;x_2;x_1]$

Definition

```
let rev = rev1 []
 whererec rev1 l = fun [] . l | (x.l') . rev1 (x.l) l'
```

The following two functions filter a list to the sublist of elements satisfying a predicate.

```
filter    : (* -> bool) -> * list -> * list
mapfilter : (* -> **) -> * list -> * list
```

Description

`filter` p l applies p to every element of l, returning a list of those that satisfy p; evaluating `mapfilter` f l applies f to every element of l, returning a list of results for those elements for which application of f succeeds.

Definition

```
letrec filter p    = fun [] . []
                     |  (x.l) . if p x then (x.filter p l) else filter p l


letrec mapfilter f = fun [] . []
                     |  (x.l) . let l' = mapfilter f l in (f x).l' ? l'
```

The following three functions break-up lists.

```
remove    : (* -> bool) -> * list -> (* # * list)
partition : (* -> bool) -> * list -> (* list # * list)
chop_list : int -> * list -> (* list # * list)
```

Description

`remove` p l separates from the rest of l the first element that satisfies the predicate p; it fails if no element satisfies the predicate. `partition` p l returns a pair of lists. The first list contains the elements of l which satisfy p. The second list contains all the other elements of l.

`chop_list` i $[x_1; \ldots; x_n] = ([x_1; \ldots; x_i], [x_{i+1}; \ldots; x_n])$

`chop_list` fails if i is negative or greater than n.

Definition

```
letrec remove p l =
 if p (hd l) then (hd l, tl l)
 else (I # (\r. ((hd l) . r))) (remove p (tl l))

let partition p l =
 itlist (\a (yes,no). if p a then ((a.yes),no) else (yes,(a.no))) l ([],[])

letrec chop_list i l =
 if i = 0 then ([],l)
 else (let l1,l2 = chop_list (i-1) (tl l) in  hd l . l1 , l2)
 ? failwith `chop_list`
```

The next function flattens a list of lists:

```
flat : * list list -> * list
```

Description

flat $[[l_{11};\ldots;l_{1m_1}];[l_{21};\ldots;l_{2m_2}];\ \ldots\ ;[l_{n1};\ldots;l_{nm_n}]] =$
$[l_{11};\ldots;l_{1m_1};l_{21};\ldots;l_{2m_2};\ \ldots\ ;l_{n1};\ldots;l_{nm_n}]$

Definition

```
letrec flat = fun [] . [] | (x.l) . x@(flat l)
```

The next two functions 'zip' and 'unzip' between lists of pairs and pairs of lists.

```
combine : (* list # ** list) -> (* # **) list
split   : (* # **) list -> (* list # ** list)
```

Description

combine $([x_1;\ldots;x_n],[y_1;\ldots;y_n]) = [(x_1,y_1);\ \ldots\ ;(x_n,y_n)]$

split $[(x_1,y_1);\ \ldots\ ;(x_n,y_n)] = ([x_1;\ldots;x_n],[y_1;\ldots;y_n])$

Definition

```
letrec combine = fun ([],[])          . []
                   | ((x.lx),(y.ly)) . ((x,y).combine(lx,ly))
                   | _               . failwith 'combine'

letrec split   = fun []         . ([],[])
                   | ((x,y).l) . let lx,ly = split l in (x.lx,y.ly)
```

7.6 Functions for lists representing sets

The following functions behave like the corresponding set-theoretic operations on sets (represented as lists without repetitions).

```
intersect : * list -> * list -> * list
subtract  : * list -> * list -> * list
union     : * list -> * list -> * list
```

Description

intersect $l_1\ l_2 = l_1 \cap l_2$ subtract $l_1\ l_2 = l_1 - l_2$ union $l_1\ l_2 = l_1 \cup l_2$

Definition

```
let intersect l1 l2 = filter (\x. mem x l2) l1

let subtract l1 l2  = filter (\x. not(mem x l2)) l1

let union l1 l2     = l1 @ subtract l2 l1
```

There are also functions to test if a list is a set, remove duplicates from a list and test two lists for set equality.

```
distinct  : * list -> bool
setify    : * list -> * list
set_equal : * list -> * list -> bool
```

Description

`distinct` l returns `true` if all the elements of l are distinct; otherwise it returns `false`. `setify` l removes repeated elements from l, leaving the last occurrence of each duplicate in the list. `set_equal` l_1 l_2 returns `true` if every element of l_1 appears in l_2 and every element of l_2 appears in l_1; otherwise it returns `false`.

Definition

```
letrec distinct l =
 (null l) or (not (mem (hd l) (tl l)) & distinct (tl l))

let setify l = itlist (\a s. if mem a s then s else a.s) l []

let set_equal l1 l2 = (subtract l1 l2 = []) & (subtract l2 l1 = [])
```

7.7 Miscellaneous string processing functions

The following functions split strings into 'words'; `words2` uses a user supplied separator, while `words` uses space and carriage-return as separators.

```
words2 : string -> string -> string list
words  : string -> string list
```

Description

`words2` 'c' '$s_1 c s_2 c \ldots c s_n$' = ['s_1';'s_2'; ... ;'s_n']

`words` '$s_1\ s_2\ \ldots\ s_n$' = ['s_1';'s_2'; ... ;'s_n']

Definition

```
let words2 sep string =
 snd (itlist (\ch (chs,tokl).
                 if ch = sep then
                    if null chs then [],tokl
                    else [], (implode chs . tokl)
                 else (ch.chs), tokl)
             (sep . explode string)
             ([],[]))


let word_separators = [` `;`\L`]

let words string =
 snd (itlist (\ch (chs,tokl).
                 if mem ch word_separators then
                    if null chs then [],tokl
                    else [], (implode chs . tokl)
                 else (ch.chs), tokl)
             (` ` . explode string)
             ([],[]))
```

The next three functions (the second of which is an infixed version of the first) are string concatenation operators.

```
concat  : string -> string -> string
$^      : string -> string -> string
concatl : string list -> string
```

Description

`concat` concatenates two strings, `$^` is an infixed version of `concat`, and `concatl` concatenates all the strings in a list of strings.

Definition

```
let concat s1 s2 = implode(explode s1 @ explode s2)

ml_curried_infix `^`
let s1 ^ s2 = concat s1 s2

let concatl sl = implode(itlist append (map explode sl) [])
```

7.8 Failure handling functions

The failure handling functions described here are useful for writing code that fails with a backtrace.

```
set_fail_prefix : string -> (* -> **) -> * -> **
set_fail        : string -> (* -> **) -> * -> **
```

Description

`set_fail_prefix` s f x applies f to x and returns the result of the application if it is successful; if the application fails then the string s is concatenated to the failure string and the resulting string propagated as the new failure string.
`set_fail` s f x applies f to x and returns the result of the application if it is successful; if the application fails then the string s is propagated as the new failure string.

Definition

```
let set_fail_prefix s f x = f x ?\s' failwith(concatl[s;`--`;s'])

let set_fail s f x        = f x ? failwith s
```

Chapter 8

ML System Functions

This chapter describes the system functions provided by ML. Further system functions, for interfacing to the logic, can be found in Section 11.11.

8.1 Input and output

8.1.1 Reading and writing files

Input and output to files in ML is done via *ports*, as in Lisp. Ports are represented by the strings that are returned by one of the functions: `openi`, `openw` and `append_openw`.

An input port is created using the following function:

```
openi : string -> string
```

Executing `openi` '*file*' returns a string that can be used by the function `read` (see below) to read characters from the file *file*; and fails if the file does not exist on the current search path.

```
read : string -> string
```

Executing `read` '*port*' reads a character from a port *port* which has been produced with `openi`.

An output port is created using one of the two functions:

```
openw        : string -> string
append_openw : string -> string
```

Executing `openw` '*file*' creates a new file named *file* for output (overwriting an existing file with the same name) and returns a string that can be used by `write` to write characters to the file.

To evaluate `append_openw` '*file*': if no file named *file* exists, a new one is created and a port for writing to it opened. If a file named *file* exists, then a port for writing to its end is returned.

The function `write` is used to write characters to a port.

```
write : string # string -> void
```

Executing `write(`*port*`,`*chars*`)` writes *chars* to port *port* (which must have been produced by `openw` or `append_openw`).

The following function closes a port:

```
close : string -> void
```

8.1.2 Terminal input and output

The following function reads a character from the terminal:

```
tty_read : void -> string
```

while the next one writes a string to the terminal:

```
tty_write : string -> void
```

8.1.3 Loading ML files

It is possible to load ML code from a file, rather than to type interactively at the console. There is a user-programmable search path mechanism which determines how ML finds a file with a given name. This is used when:

(i) The system searches for files (via `load` and `compile`, see below).

(ii) The system loads or reads theories.

A file is searched for by first looking in the current working directory (i.e. the one where HOL was run). If the file is not found there, then the search path is invoked by successively concatenating its members to the file name (so members of the search path must end with `/`).

When the HOL system is built, the search path is initially set to contain: the empty string (for absolute path names), the string `` `~/` `` (i.e. the user's home directory) and the directory where the built-in theory files live. Its current value is given by the ML function:

```
search_path : void -> string list
```

The search path can be changed by:

```
set_search_path : string list -> void
```

The result of a search is given by:

```
find_file : string -> string
```

This function takes a file name (e.g. `foo`, `foo.ml` or `foo.th` etc.) and returns the result of searching for it using the search path. It fails if the file cannot be found.

The functions:

```
find_ml_file : string -> string
find_theory  : string -> string
```

find the file that would be loaded by `load` (and `loadt`, `loadf` etc.—see below for all of these), and the theory file that would be loaded, respectively. For example,

```
find_ml_file `file`
```

might return `` `~turkey/wonk/file.ml` `` or `` `~turkey/wonk/file_ml.o` ``.

The main file loading function is:

```
load : string # bool -> void
```

Evaluating `load(`*filename*`,`*prflag*`)` reads phrases from the first file named *filename* on the current search path and evaluates them exactly as if they had been typed by the user at top-level. If *prflag* is `true` then the result of each evaluation is printed exactly as is done interactively; if *prflag* is `false` then only a dot is printed for each top-level expression or declaration.

The file named *filename* is searched for as follows:

- If *filename* is a string not ending in `.ml`, then the system first tries to find a compiled file *filename*`_ml.o`; if that does not exist, then it looks for *filename*`.ml`.
- If *filename* does end in `.ml` then the system does not concatenate an extra `.ml` (as it would in older versions of HOL), but it looks for a *filename* (and ignores *filename*`_ml.o` if this exists).

All this is done in turn for each directory on the search path. If any failure occurs during loading, the call of `load` fails without the rest of the file being read. Calls of `load` can be nested, i.e. a file being input by `load` can itself contain calls of `load`.

The following special cases of `load` are provided for convenience:

```
loadt : string -> void
loadf : string -> void
```

These are defined by:

```
let loadt filename = load(filename,true)
and loadf filename = load(filename,false)
```

8.1.4 Compiling ML files

In Common Lisp versions of HOL, ML source code can be compiled on-the-fly by setting the flag `compile_on_the_fly` to `true` (this has no effect in Franz Lisp versions of HOL). In both Common Lisp and Franz Lisp versions of HOL, ML files can be compiled and the result saved in an object code file. An ML file can be compiled using the functions:

```
compile  : string # bool -> void

compilet : string -> void
compilef : string -> void
```

The conventions are the same as for `load`. The ML source file should always have a name of the form *filename*`.ml`; the object file will then have the name *filename*`_ml.o`. The effect of invoking `compile` (or `compilet` or `compilef`) is to load, execute, compile, and save the given file. The compiled code (placed on the current working directory) may be loaded by the `load` function. After successful compilation, an invocation of the `load` command is necessary in order to load the compiled code. It is not advisable to compile files containing the `compile` command, as this will not work in general. Object files must be loaded in the order in which they were compiled. If this is not the case the following kind of error will result:

```
#loadt`~/hol/ml/lis`;;                                                    1
Undefined function called from compiled code   FUN%6915%59
Lisp error during evaluation
evaluation failed
```

Functions with names like `FUN%6915%59` are generated automatically by the ML compiler as Lisp names for ML lambda expressions. In the session above, the compiled file `lis` was loaded before the object file defining `FUN%6915%59` was loaded.

8.2 Output

8.2.1 Pretty printing

The functions below have the side effect of printing their argument exactly as it would be printed at top-level, except that

(i) `print_string` does not print the quotes that delimit strings, but `print_tok` does;

(ii) `print_all_thm` prints out the assumptions of theorems, whereas `print_thm` just prints one dot (that is, period or full stop) per assumption, as at top-level (see Section 8.14.2 for how to change the top-level printing to use `print_all_thm`).

```
print_void    : void   -> void
print_bool    : bool   -> void
print_int     : int    -> void
print_string  : string -> void
print_tok     : string -> void
print_term    : term   -> void
print_type    : type   -> void
print_thm     : thm    -> void
print_all_thm : thm    -> void
```

All text printed by ML goes through a pretty-printer, controllable by the user using the commands below. The pretty-printer queues the text until it determines where line breaks will be needed; at any point, the queue may hold up to three lines of text. The text consists of nested blocks; a block that cannot fit on one line is broken at designated places.

```
print_begin  : int -> void
print_ibegin : int -> void
```

These functions begin printing blocks in which breaks are *consistent* or *inconsistent*, respectively. Consistent breaks cause a uniform indentation at each break; inconsistent breaks pack onto a line and break only when there is no more space available. For example, if there is a consistent break after each semicolon in `[1;2;3;4;5;6;7;8;9]` then that list is printed as

```
[1;
 2;
 3;
 4;
 5;
 6;
 7;
 8;
 9]
```

but if the breaks are inconsistent then the list might, for example, be printed as

```
[1; 2; 3 ;4 ;5; 6;
 7; 8; 9]
```

A call of the form `print_begin` $offset$ signals the beginning of a block that should be broken consistently (or inconsistently using `print_ibegin`), with $offset$ added to the indentation of any lines broken inside. The end of either kind of printing block is signalled by:

```
print_end : void -> void
```

Calls to `print_begin` (and `print_ibegin`) and `print_end` should nest like brackets. When writing a recursive printing routine, the user should include `print_begin 0` as its first command and `print_end ()` as its last.

Breaks are generated with the functions:

```
print_break   : int # int -> void
print_newline : void -> void
```

A call `print_break(`*width*`,`*offset*`)` tells the prettyprinter that the line may be broken at that location, if necessary. If the line is broken, then *offset* is added to the current indentation; otherwise, *width* determines the amount of blank space inserted horizontally. The function `print_newline` prints all the queued text, followed by a carriage return. The margin is taken by default to be at position 72, but can be changed with the function:

```
set_margin : int -> int
```

The value of the old margin is returned. As an example to illustrate the pretty printing primitives, the code for the following built-in ML function will be given.

```
print_list : bool -> string -> (* -> **) -> * list -> void
```

This function is used by the HOL system for printing theories; it prints out a list using a supplied printing function for its elements. The list is preceded by the supplied string and two dashes `--`. The argument of type `bool` determines whether the list is to be printed with consistent or inconsistent breaks. For example:

1

```
#set_margin 15;;
72 : int

#[1;2;3;4;5;6];;
[1;
 2;
 3;
 4;
 5;
 6]
: int list
```

The function `print_list` is defined after the next box.

```
#print_list true ‘Test1:‘ print_int [1;2;3;4;5;6];;                    2
Test1: --
  6     5
  4     3
  2     1
() : void

#print_list false ‘Test2:‘ print_int [1;2;3;4;5;6];;
Test2: --
  6
  5
  4
  3
  2
  1

() : void
```

The definition of `print_list` is:

```
let print_list incon name prfn l =
    if not (null l) then
    do (print_begin 2;
        print_string name;  print_string ‘ --‘;
        print_break (2,0);
        if incon then print_ibegin 0
        else print_begin 0;
        map (\x. prfn x; print_break (5,0)) (rev l);
        print_end(); print_end();
        print_newline())
```

The maximum depth of block nesting is initially 500, but can be changed by:

```
max_print_depth : int -> int
```

The old value of the maximum print depth is returned. When the current depth exceeds the maximum depth, the output is replaced by the holophrast `&`.

The following function first prints all remaining text in the print queue; then if the argument is `true`, the pretty printer is reinitialized.

```
set_pretty_mode : bool -> void
```

8.3 Exiting and re-entering the system

A executable Lisp core image of the current session is dumped by the function:

```
save : string -> void
```

The argument to `save` is the name of the executable file produced. Currently, an executable Franz Lisp HOL is about 3 megabytes and an executable Common Lisp HOL is about 7 megabytes (using the public domain Austin Kyoto Common Lisp).

A HOL session is terminated by executing `quit()`, where:

```
quit : void -> void
```

8.4 Autoloading

The ML function

```
autoload : string # string # string list -> void
```

can be used to mark a string x so that when it occurs as an ML variable, an action is done before the expression containing x is evaluated. A common action is to load a definition or theorem named x from a theory.

Evaluating:

`autoload(`‘x‘`, `‘fun‘`, [`‘s_1‘`; ... ;`‘s_n‘`])`

marks x so that whenever it is parsed as an ML variable in a declaration or expression E, the ML application fun[‘s_1‘; ... ;‘s_n‘] is evaluated before E is evaluated. In the session below, the various outputs are explained in the comments that follow them.

1

```
#let fn[x] =
# print_newline();print_string x;print_newline();print_newline();‘fn‘;;
fn = - : (string list -> string)

#autoload(‘Mike‘,‘fn‘,[‘Hello Mike‘]);;autoload(‘Mike‘,‘fn‘,[‘Hello Mike‘]);;
() : void

#let Mike = 1;;       % Typed at top level                          %

Hello Mike            % Printed by the autoloaded call of fn        %

‘fn‘ : string         % The value returned by the call of fn        %

Mike = 1 : int        % The result of evaluating ‘let Mike = 1;;‘ %
```

The effect of:

```
autoload(‘Mike‘,‘fn‘,[‘Hello Mike‘])
```

is that whenever `Mike` is parsed as a variable, the ML expression

```
fn[‘Hello Mike‘]
```

is put into the input stream as though it has been input before the expression containing `Mike` was input. Subsequent occurrences of `Mike` will also cause `fn[‘Hello Mike‘]` to be evaluated.

2

```
#Mike+1;;

Hello Mike

‘fn‘ : string

2 : int
```

The ML function

```
undo_autoload : string -> bool
```

removes the autoload action associated with a string; `true` is returned if such an action has already been set up, `false` if not.

3

```
#undo_autoload ‘Mike‘;;
true : bool

#Mike+1;;
2 : int
```

The intended use of this mechanism is to support predefined ML variables with their values only being loaded if they are actually used. This avoids, for example, the HOL system becoming cluttered up with a lot of predefined theorems; the space for them is only used if they are actually needed. To program this autoloading, the following functions are useful.

```
let_before : (string # string # string list) -> void
let_after  : (string # string # string list) -> void
```

Evaluating:

`let_before(‘`*MLname*`‘, ‘`*MLfn*`‘,[‘`s_1`‘;...;‘`s_n`‘])`

causes the declaration:

`let` *MLname* `=` *MLfn*`[`‘s_1‘`;` ... `;`‘s_n‘`]`

to be put at the beginning of the list of pending top-level evaluations. As soon as the current top-level evaluation is completed (i.e. the one that invoked the call to `let_before`), the declaration or expression at the front of this list of pending evaluations is executed. The ML function `let_after` is like `let_before`, except that the declaration above is put at the end of the list of pending evaluations.

To illustrate the use of `autoload` for autoloading theorems, suppose a theory `triv` has previously been created which contains theorems named `Th1`, `Th2` and `Th3`.

```
#load_theory‘triv‘;;                                                    4
Theory triv loaded
() : void

#Th1;;

unbound or non-assignable variable Th1
1 error in typing
typecheck failed
```

To arrange for `Th1`, `Th2` and `Th3` to be autoloaded, one first defines:

```
#let theorem_list_fn[thy;thname] = theorem thy thname;;                 5
theorem_list_fn = - : (string list -> thm)
```

This is a version of the ML function `theorem` that takes its arguments as a list. Then one defines:

```
#let load_thm[thy;thname] =                                             6
# let_before(thname, ‘theorem_list_fn‘, [thy;thname]);;
load_thm = - : (string list -> void)
```

This defines `load_thm` to be a function that will load and bind `thname` from the theory `thy`. Finally, one defines:

```
#let autoload_thm thy thname =                                          7
# autoload(thname,‘load_thm‘,[thy;thname]);;
autoload_thm = - : (string -> string -> void)
```

This takes a theory and a theorem name and sets up the autoloading of the theorem from the given theory. The three theorems can now be set up for autoloading as follows:

```
#map (autoload_thm ‘triv‘) (words ‘Th1 Th2 Th3‘);;                      8
[(); (); ()] : void list
```

Then if `Th1`, `Th2` or `Th3` occur, they will be autoloaded. This is illustrated below, with the output followed by explanatory comments.

```
#REWRITE_RULE[Th2;Th3]Th1;;      % Expression input to top level ML %
() : void                        % result of load_thm[`triv`;`Th1`] %
Th1 = . |- x = (y * 1) + 0       % Th1 autoloaded                    %

() : void                        % result of load_thm[`triv`;`Th3`] %

Th3 = |- m * 1 = m               % Th3 autoloaded                    %

() : void                        % result of load_thm[`triv`;`Th2`] %

Th2 = |- m + 0 = m               % Th2 autoloaded                    %

. |- x = y                       % Result of input expression       %
```

9

This implementation of autoloading illustrates the general principles, but has the disadvantage that theorems are always reloaded whenever they occur. As an exercise, the reader might wish to consider how to modify the definitions above (using `undo_autoload`) so that theorems are only loaded once per session.

A special purpose autoloading function, for axioms, definitions and theorems, is provided as standard. This is coded to provide a more pleasing output than given by the definitions above, and it only autoloads once. The function is:

```
autoload_theory  : (string # string # string) -> void
```

The first argument must be one of the strings `` `axiom` ``, `` `definition` `` or `` `theorem` ``; the second argument a theory; and the third argument the name of an axiom, definition or theorem from the theory. The result is a side-effect that sets up the autoloading of the axiom, definition or theorem from the given theory.

8.5 Interpreting lists of numbers as ML input

The ML function:

```
inject_input : int list -> void
```

concatenates its argument (considered as a list of ASCII characters) onto the front of the input character stream. The function:

```
ML_eval : string -> void
```

is defined in terms of `inject_input`; it converts its argument string to a list of ASCII characters and then injects these into the input stream. The implementation of `ML_eval` has not been finalised yet; see *REFERENCE* for more details.

Suppose, for example, that the following ML bindings are made:

```
#let space = ascii_code ` `;;                                                    1
space = 32 : int

#let ascii_ize tok =
#    itlist (\n l. map ascii_code (explode n)@[space]@l)
#           (words tok)
#           [];;
ascii_ize = - : (string -> int list)

#let ML_eval = inject_input o ascii_ize;;
ML_eval = - : (string -> void)
```

Then, for example:

```
#ML_eval `let x=1;;`;;                                                           2
() : void

#
x = 1 : int
```

8.6 Initialization

When HOL is first run, it looks to see if can find a file `hol-init.ml` (using the search path mechanism) and if it finds one, it is loaded (using `loadt`).

8.7 Operating system calls from ML

There is a miscellaneous collection of operating system calls available from ML. Some of these may not work in some systems (e.g. non Unix versions of HOL).

```
   system : string -> int
```

This executes the string argument in a separate process. The number returned by the process is returned by the ML function `system`. For example, if `foo14` exists on the current directory, then:

```
#system `rm foo14`;;                                                             1
0 : int

#system `rm foo14`;;
rm: foo14: No such file or directory
1 : int
```

```
host_name : void -> string
```

This returns the name of the host machine.

```
link : string # string -> void
```

`link(`‘*old*‘`,`‘*new*‘`)` links the file *old* to *new*; and fails if *old* does not exist.

```
unlink : string -> void
```

This unlinks (i.e. deletes) a file; and fails if the file doesn't exist.

8.8 Getting the version number

The ML function:

```
version : void -> int
```

returns the version number of the HOL system as an integer less than 1000 (i.e. in `HOL88 Version 1.`*mn*, evaluating `version()` returns `1`*mn*). For example, in version 1.10:

1
```
#version();;
110 : int
```

8.9 Ordering of ML values

A predefined ML infix

```
$<< : * -> ** -> int
```

is provided for comparing values; this is substitutive with respect to ML's equality (i.e. if t_1`<<`t_2 and t_1`=`t_1' and t_2`=`t_2' then t_1'`<<`t_2').

2
```
#"x+1" << "x+2";;
true : bool

#"x+2" << "x+1";;
false : bool
```

8.10 Printing defined types

The ML function:

```
print_defined_types : void -> void
```

prints out the definitions of all currently defined ML types (i.e. all ML types that have been defined using `lettype`). It also lists the abstract types in the system. The printing of `lettype`-defined types on output can be partially suppressed by setting the flag `print_lettypes` to `false` (the suppression is only partial, in that only the outer level of defined types is expanded). For example, if `print_lettypes` is `false`, then instead of printing:

```
··· : subgoals
```

ML will print:

```
··· : ((term list # term) list # proof)
```

The defined type `proof` (namely, `thm list -> thm`) is not expanded.

8.11 Lisp in ML

It is possible to connect to the Lisp world from ML by using the following function:

```
lisp : string -> void
```

This function interprets its argument string as an input to the Lisp interpreter. This permits arbitrary operating system operations. Beware: uncontrolled use of this feature may totally break ML.

8.12 Fast arithmetic

ML arithmetic is normally done in arbitrary precision. It is possible to speed up arithmetic, when the user is sure that integers will not exceed the word length of the machine, by using the function:

```
fast_arith : bool -> void
```

Applying `fast_arith` to `true` switches HOL to a mode in which subsequent arithmetic operations are done using finite precision fast arithmetic; applying it to `false` switches HOL back to infinite precision. Already defined functions are not affected by `fast_arith`. Here is a session that illustrates this:

```
#letrec exp m n = if n=0 then 1 else m*(exp m (n-1));;                    1
exp = - : (int -> int -> int)

#exp 2 30, exp 2 31, exp 2 32;;
(1073741824, 2147483648, 4294967296) : (int # int # int)

#fast_arith true;;
() : void

#exp 2 30, exp 2 31, exp 2 32;;
(1073741824, 2147483648, 4294967296) : (int # int # int)

#letrec exp m n = if n=0 then 1 else m*(exp m (n-1));;
exp = - : (int -> int -> int)

#exp 2 30, exp 2 31, exp 2 32;;
(1073741824, -2147483648, 0) : (int # int # int)

#fast_arith false;;
() : void

#exp 2 30, exp 2 31, exp 2 32;;
(1073741824, -2147483648, 0) : (int # int # int)

#letrec exp m n = if n=0 then 1 else m*(exp m (n-1));;
exp = - : (int -> int -> int)

#exp 2 30, exp 2 31, exp 2 32;;
(1073741824, 2147483648, 4294967296) : (int # int # int)
```

N.B. `fast_arith` is only supported in Franz Lisp implementations of HOL; it may or may not work in other implementations.

8.13 Flags

A *flag* is a boolean state variable of the HOL system. For example, the state of the flag `prompt` determines whether the ML prompt is printed. Many of the flags listed below affect the way that quotations are parsed and printed. These are described in detail in Chapter 11. For example, if the flag `print_let` is `true` (the default) then `let`-terms are printed in their surface syntax; if it is `false` then the 'deep structure' is printed (see Section 11.5.3).

Note that the flags `print_parse_trees`, `print_sexpr`, `pp_sexpr` and `read_sexpr` in the table below are intended to support HOL as an embedded system within Centaur, a tool from Inria at Sofia Antipolis (France) that is being used for experiments with a new interface (see Section 8.19). These flags are not intended for general use.

Settable system flags

Flag	*Function*	*Default value*
`timing`	Printing of evaluation times, garbage collect times and number of theorems proved	`false`
`show_types`	Prints types in quotations	`false`
`sticky`	Activates sticky types	`false`
`prompt`	ML prompt printed	`true`
`preterm`	Typecheck HOL with `preterm_handler`	`false`
`theory_pp`	Pretty printing of theory files	`false`
`type_error`	Verbose type checking errors in quotations	`true`
`interface_print`	Causes inverse of interface map to be used when printing	`true`
`abort_when_fail`	Modifies HOL so that any ML failure causes a quit	`false`
`print_top_types`	Determines whether ML prints types at top level	`true`
`print_top_val`	Determines whether ML prints value and type at top level	`true`
`print_cond`	Pretty print HOL conditionals	`true`
`print_all_subgoals`	Print all subgoals	`true`
`print_quant`	Pretty print HOL quantifications	`true`
`print_let`	Pretty print HOL `let`-expressions	`true`
`print_restrict`	Pretty print HOL binder restrictions	`true`
`print_list`	Pretty print HOL lists	`true`
`print_set`	Pretty print HOL sets	`false`
`print_uncurry`	Pretty print HOL paired abstractions	`true`
`print_infix`	Pretty print HOL infixes	`true`
`print_lettypes`	Print ML defined types	`true`
`print_load`	Flag set by the ML function `load` to determine printing when loading files	`true`
`print_fasl`	Print fasl messages from lisp when loading files	`false`
`file_load_msg`	Print message after loading file	`false`
`print_lib`	Verbose printing of library loading	`false`
`compile_on_the_fly`	Causes on-the-fly compilation in Common Lisp HOL (no effect in Franz Lisp HOL)	`false`
`print_parse_trees`	Print ML parse tree	`false`
`print_sexpr`	Print terms and theorems as Lisp	`false`
`pp_sexpr`	Pretty print Lisp output	`true`
`read_sexpr`	Enable direct parse tree input	`false`

The value of a flag is set with the ML function:

```
set_flag : (string # bool) -> bool
```

This function takes a pair (‘*flag*‘,*b*), where *b* is `true` or `false`, and sets the value of the flag *flag* to *b*. The previous value of the flag is returned. The settable flags in the system are those listed in the table above.

The current value of a flag is returned by the ML function:

```
get_flag_value : string -> bool
```

A new flag can be created with the ML function:

```
new_flag : (string # bool) -> void
```

The string is the flag's name and the boolean is its initial value; failure occurs if the flag already exists.

8.14 Modifying the ML read-eval-print loop

The top-level of ML can be customized in various ways. Some of these concern the representation of the HOL logic: see Section 8.13 above for various flags that control the parsing and printing of terms, see Section 11.11.4 for changing the concrete syntax for λ inside quotations, and see Section 11.11.5 for changing the string that the printer uses to represent $\vdash$.

8.14.1 Changing the ML prompt

The ML prompt string , default `#`, can be dynamically changed with the function:

```
set_prompt : string -> string
```

Executing `set_prompt` ‘*string*‘ changes the ML prompt to *string*; the previous prompt string is returned. For example:

```
#set_prompt ‘hol88> ‘;;                                                    1
‘#‘ : string

hol88> 1+2;;
3 : int

hol88>
```

Setting the flag `prompt` to `false` switches off the printing of prompts. In some contexts flag `prompt` being `false` has the same effect as setting the prompt to the empty string (‘‘), but in others (e.g. in some emacs modes) it is different.

8.14.2 Changing the top-level printing of values in ML

The printing of values in the ML read-eval-print loop can be modified to use a user-supplied printing function. This is mainly useful for defining the printing of abstract types.

If *foo* is an ML function with ML type *ty* `->` *ty'*, then evaluating: `top_print` *foo* at top-level will replace the default top-level printing of values of type *ty* by calls to *foo*. For example:

```
top_print print_all_thm
```

will make the assumptions of theorems be printed out in full, rather than just as a period (dot, full stop) (`print_all_thm` is a function that prints out the assumptions and conclusions of a theorem).

Note that `top_print` is a construct of ML, not an ordinary function. It should only be used at top-level; an error will result otherwise.

The following example illustrates how abstract type values can be made to print nicely.

1

```
#abstype two = void + void
# with left = abs_two(inl())
# and right = abs_two(inr())
# and print_two x = print_string(isl(rep_two x) => `left` | `right`);;
left = - : two
right = - : two
print_two = - : (two -> void)

#left;;
- : two

#right;;
- : two

#top_print print_two;;
- : (two -> void)

#left;;
left : two

#right;;
right : two
```

```
#top_print (\x. print_string`Int<`; print_int x; print_string`>`);;                2
- : (int -> void)

#1;;
Int<1> : int

#[1;2;3];;
[Int<1>; Int<2>; Int<3>] : int list
```

8.14.3 Suppressing the printing of ML types

The flag `print_top_types` determines whether ML prints out the types of values at top level. For example, continuing the session above:

```
#set_flag(`print_top_types`,false);;                                               3
true

#1;;
Int<1>

#[1;2;3];;
[Int<1>; Int<2>; Int<3>]
```

8.15 The help system

The help system accesses a database containing a description of every ML function predefined in HOL. This database is in the directory `help` and also provides the sources for *REFERENCE*. The help system is invoked with the function:

```
   help : string -> void
```

Evaluating `help` `` *foo*` concatenates the contents of the help file associated with *foo* into the current session. For example:

```
#help `length`;;                                                    1

length : * list -> int

SYNOPSIS

Computes the length of a list: length [x1;...;xn] returns n.

FAILURE CONDITIONS

Never fails.
```

The help system uses the Unix `cat` as the default for displaying help files. This default can be changed with the ML function:

```
set_help : string -> string
```

This installs a new user-supplied help function, and returns the previous one as result. The effect of `help` `` ``*file*`` is to pipe the appropriate help file derived from *file*`.doc` into the current help function, with the top level of ML being the standard output. For example,

```
set_help `lpr`
```

will cause the help file to be printed instead of being displayed and

```
set_help `lpr -Pbarf`
```

will cause it to be printed to the printer `barf`.

Two new functions:

```
set_help_search_path : string list -> void
help_search_path     : void -> string list
```

have been added to the system for accessing the internal search path used by HOL to find online help files. The help search path has precisely the same format as the regular search path, and these two functions are analogous to the ML functions `search_path` and `set_search_path`.

8.16 Syntax blocks

A syntax block starts with a keyword and ends with a terminator and is associated with a function on strings. When such a block is encountered in the input stream, all the characters between the start keyword and the terminator are made into a string to which the associated function is applied. Syntax blocks can be used with the parser generator library `parser` to parse user-specified languages into HOL terms. See the documentation of the library for details and examples.

The ML function:

```
new_syntax_block : string # string # string -> void
```

declares a new syntax block. The first argument is the start keyword of the block, the second argument is the terminator and the third argument is the name of a function having a type which is an instance of `string->*`. The effect of

```
new_syntax_block(‘XXX‘, ‘YYY‘, ‘f‘);;
```

is that if subsequently

```
XXX   ...   YYY
```

occurs in the input, then it is as though

```
f ‘   ...   ‘
```

were input instead. For example:

```
#let foo s = print_string ‘Hello: ‘; print_string s; print_newline();;
foo = - : (string -> void)

#new_syntax_block(‘<<‘,‘>>‘, ‘foo‘);;
() : void

#<< Mike >>;;
Hello: Mike
() : void
```

The function `set_string_escape` (see Section 3.3.2) is useful with syntax blocks, because it enables backslash (\) to be included in strings.

8.17 Relocating HOL

The built-in ML function:

```
install : string -> void
```

reconfigures a running HOL system to a new root directory. The string argument to `install` should be the absolute path name to the directory in which the HOL system is located. Chapter 1 of *TUTORIAL* ('Getting and Installing HOL') describes in detail how to configure HOL.

Executing `install ‘`*directory*`‘` does the following:

- Sets the search path to: `[‘‘; ‘~/‘; ‘`*directory*`/theories/‘]`
- Sets up the internal search path used by HOL to find online help files.
- Sets up the the internal pathname used by HOL to find libraries.

8.18 Libraries

The HOL system has a library mechanism which allows useful theories and tools to be loaded on demand in a uniform way. A library is loaded with the function:

```
load_library : string -> void
```

A call to `load_library` ‘*lib*‘ will do nothing if the library *lib* is already loaded. Otherwise it will load into HOL the *library load file* called *lib*`.ml`, which is expected to be found in the subdirectory *lib* of the HOL `Library` directory (see Section 8.18.2).

It is the responsibility of the author of a library to create an appropriate load file. This file must contain ML code that carries out all the actions necessary to make the contents of the library available in HOL. For example, for a library *lib* that contains both executable code and theory files, the standard sequence of events would be:

(i) add to the search path the absolute pathname to the library *lib*.

(ii) add to the help search path any pathnames to directories containing online help files specific to the library *lib*.

(iii) if the user is in draft mode, then

- make the theories of the library new parents of the current theory.
- activate autoloading from these theories.
- load any ML code that is part of the library.

(iv) if the user is *not* in draft mode, but the theories in *lib* are descendents of the current theory, then:

- make the main library theory the current theory.
- activate autoloading from library theories.
- load any ML code that is part of the library.

(v) if the user is not in draft mode and the theories in *lib* are not descendents of the current theory, then these theories cannot at this stage be loaded into HOL. In this case, the load file should cause a function to be defined (e.g. ‘`load_`*lib*’) which, when called, will activate autoloading for the library theories and load the library's ML code.

The output during the loading of a library is determined by the flag `print_lib`. If this is `true`, then the output is verbose. For consistency, libraries should be set up so that any loading of ML files that occurs when a library is loaded is done by:

```
load(‘file‘, get_flag_value ‘print-lib‘)
```

This makes verbose printing during the loading of *file* conditional on the current value of the flag `print_lib`.

A list of currently loaded libraries can be obtained using:

```
libraries : void -> string list
```

8.18.1 Libraries available in Version 2.0

The libraries in HOL88.2.0 are shown in the following table:

Libraries in HOL88.2.0

Library	*Brief description*	*Author*
`string`	`ASCII` characters & strings	T. Melham (Camb.)
`eval`	`word`n types from LCF_LSM	M. Gordon & T. Melham (Camb.)
`unwind`	Structure unwinding rules	M. Gordon & T. Melham (Camb.)
`taut`	Tautology checker	R. Boulton (Camb.)
`fixpoints`	General theory of fixed points	M. Gordon (Camb.)
`prog_logic88`	Hoare Logic & Dijkstra Wp	M. Gordon (Camb.)
`group`	Group theory	E. Gunter (Penn.)
`integer`	Integers derived from groups	E. Gunter (Penn.)
`finite_sets`	Theory of finite sets	P. Windley (UC Davis)
`sets`	Manna & Waldinger set theory	P. Leveilley (Ecole Polytechnique)
`bags`	Manna & Waldinger bag theory	P. Leveilley (Ecole Polytechnique)
`convert`	Conversion & unwinding tools	D. Shepherd (Inmos)
`auxiliary`	General rules, tactics & theorems	T. Kalker (Philips, ERL)
`set`	Set theory of predicates	T. Kalker (Philips, ERL)
`quotient`	Tools for quotient types	T. Kalker (Philips, ERL)
`parser`	Parser generator	J. Van Tassel (Camb.)
`prettyp`	Pretty-printer generator	R. Boulton (Camb.)
`trs`	Theorem retrieval system	R. Boulton (Camb.)

Four libraries from Version 1.11 of HOL have been temporarily withdrawn, because the Cambridge group have been unable to rebuild them in Version 2.0. These are:

```
card   well_order   zet   csp
```

The first three are difficult to rebuild because of changes to resolution. They rely very heavily on the order in which the assumptions of goals appear. The fourth library uses the `set` library, which has been substantially modified. These four libraries have been moved to `contrib` until they are updated by their authors.

The libraries `eval` and `unwind` are of historical interest only and will eventually be replaced.

8.18.2 Location of libraries

The function

```
library_pathname : void -> string
```

returns the internal pathname used by HOL to load libraries. This pathname, which is site-specific and is given an initial value when the system is built, should be the absolute pathname of the HOL system library directory. This pathname will typically have the form:

‘*directory*/hol/Library‘

where *directory* is the site-specific absolute pathname in which the HOL distribution directory (‘hol’) resides. The value returned by `library_pathname` can be changed by users only via the `install` function.

The string returned by `library_pathname` is primarily used in library load files to update the HOL search path and help search path. For example, suppose that in a library *lib* there is a directory `help` which contains online help files specific to this library. The load file *lib*`.ml` can then update the help search path as follows:

```
let path = library_pathname() ^ ‘/lib/help/‘
in
set_help_search_path (path . help_search_path())
```

This will make the help files of the library *lib* available for online help whenever the library is loaded.

8.18.3 Adding a new library

If *lib* is a library, then the directory `hol/Library/`*lib* needs to contain the following files (in addition to the various files making up the library):

(i) a file `READ-ME`, which gives details of the library.

(ii) a library load file *lib*`.ml`. This is the file loaded by `load_library`.

(iii) A `Makefile` which tells how to rebuild theories and compile ML. The `Makefile` should set up such that

- `make all` completely rebuilds the library.
- `make clean` deletes all object code in the library.
- `make clobber` deletes all object code and theory files in the library.

In addition, `lib` can optionally contain a directory `help` containing help files in the standard `.doc` format.

To install a new library *lib* in HOL, the directory *lib* should be added to `hol/Library` and the following files modified:

(i) `hol/Library/Makefile`: add *lib* to the list of libraries given by the makefile macro `Libraries`.

(ii) `hol/Versions.`*m*`.`*n*: add an entry to document the addition of *lib*.

The simplest way to make a library is to copy an existing one (e.g. `hol/Library/string`) and then to edit the various files. This will result in a uniform format.

8.19 Experimental support for a new interface

The material in this section supports a new and experimental interface to HOL being developed at Inria in Sofia Antipoles (France). For details, contact:

Gilles Kahn (`email:` Gilles.Kahn@fr.inria.mirsa)
INRIA — Sophia Antipolis
2004, route des Lucioles
B.P. 109
06561 Valbonne Cédex
France

This interface is implemented using a tool called Centaur and will support full colour graphic input including, for example, the selection of subterms, goal assumptions etc by mouse pointing. During 1991 experiments are being conducted to determine whether such an interface will actually improve proof productivity. Only when this is known will a production version be produced.

> *The facilities described in the rest of this section are not intended for general use. They describe the data formats being used to communicate with Centaur and are likely to change.*

Flags have been set up to enable HOL to interact by sending and receiving S-expressions from Centaur.

If the flag `print_parse_trees` is set to `true` then ML prints out the parse tree of the expression or declaration just input. Note that this is usually the value in `%pt`, but not always, because in some cases the read-eval-print loop of ML destructively modifies `%pt`. A variable `%pt1` holds the parse tree that is printed; this variable is only maintained (by `okpass` in `f-tml.l`) if the flag `print_parse_trees` is `true`.

If the flag `print_sexpr` is set to `true`, then an S-expression representing the value of a term or theorem is printed instead of the normal output. The printing of types is suppressed for all values.

The printing in the two cases above is pretty if the flag `pp_sexpr` is `true` (the default) and non-pretty otherwise. Here is an example session:

1

```
#let x = 1;;
x = 1 : int

#set_flag(`print_parse_trees`,true);;
false : bool

(mk-empty)

#let x = 1;;
x = 1 : int

(mk-let ((mk-var x) mk-intconst 1))
```

If `read_sexpr` is set to `true`, then parse trees may be input between `begin_parse_tree` and `end_parse_tree`. For example:

```
begin_parse_tree
 (mk-let ((mk-var x) mk-quot (MK=CONST T)))
end_parse_tree
```

is equivalent to:

```
let x = "T"
```

Continuing the session:

2

```
#set_flag(`read_sexpr`, true);;
false : bool

(mk-appn
 (mk-var set_flag)
 (mk-dupl (mk-tokconst read_sexpr) (mk-boolconst t)))

#begin_parse_tree (mk-let ((mk-var x) mk-intconst 1)) end_parse_tree;;
x = 1 : int

(mk-let ((mk-var x) mk-intconst 1))

#set_flag(`print_parse_trees`,false);;
true : bool

#begin_parse_tree
# (mk-let ((mk-var x) mk-quot (MK=CONST T)))
#end_parse_tree;;
x = "T" : term
```

Terms can also be input in the form printed out when `print_sexpr` is `true`, by enclosing them between `begin_term` and `end_term`. For example:

```
begin_term
(comb (comb (const + (fun (num) (fun (num) (num))))
            (var x (num))
            (fun (num) (num)))
      (const |1| (num))
      (num))
end_term
```

is equivalent to:

```
"x+1"
```

Continuing the session:

3

```
#begin_term
#  (comb (comb (const + (fun (num) (fun (num) (num))))
#               (var x (num))
#               (fun (num) (num)))
#        (const |1| (num))
#        (num))
#  end_term;;
"x + 1" : term

#set_flag(`print_sexpr`, true);;
false

#"x+1";;
(comb (comb (const + (fun (num) (fun (num) (num))))
            (var x (num))
            (fun (num) (num)))
      (const |1| (num))
      (num))

#[1;2;3];;
[1; 2; 3]
```

It is expected that the exact details of the format used to communicate between HOL and Centaur will change. For example, it is likely that in future releases all types, except those on variables, will be suppressed in S-expression output.

Part II

The HOL Logic

Chapter 9

Syntax and Semantics

9.1 Introduction

This chapter describes the syntax and set-theoretic semantics of the logic supported by the HOL system, which is a variant of Church's simple theory of types [3] and will henceforth be called the HOL logic, or just HOL. The meta-language for this description will be English, enhanced with various mathematical notations and conventions. The object language of this description is the HOL logic. Note that there is a 'meta-language', in a different sense, associated with the HOL logic, namely the programming language ML. This is the language used to manipulate the HOL logic by users of the system, and is described in detail in Part I of *DESCRIPTION*. It is hoped that because of context, no confusion results from these two uses of the word 'meta-language'. When ML is described in Part I, ML is the object language under consideration—and English is again the meta-language!

The HOL syntax contains syntactic categories of types and terms whose elements are intended to denote respectively certain sets and elements of sets. This set theoretic interpretation will be developed along side the description of the HOL syntax, and in the next chapter the HOL proof system will be shown to be sound for reasoning about properties of the set theoretic model.[1] This model is given in terms of a fixed set of sets $\mathcal{U}$, which will be called the *universe* and which is assumed to have the following properties.

Inhab Each element of $\mathcal{U}$ is a non-empty set.

Sub If $X \in \mathcal{U}$ and $\emptyset \neq Y \subseteq X$, then $Y \in \mathcal{U}$.

Prod If $X \in \mathcal{U}$ and $Y \in \mathcal{U}$, then $X \times Y \in \mathcal{U}$. The set $X \times Y$ is the cartesian product, consisting of ordered pairs (x, y) with $x \in X$ and $y \in Y$, with the usual set-theoretic coding of ordered pairs, viz. $(x, y) = \{\{x\}, \{x, y\}\}$.

Pow If $X \in \mathcal{U}$, then the powerset $P(X) = \{Y : Y \subseteq X\}$ is also an element of $\mathcal{U}$.

Infty $\mathcal{U}$ contains a distinguished infinite set I.

[1] There are other, 'non-standard' models of HOL, which will not concern us here.

Choice There is a distinguished element $\mathrm{ch} \in \prod_{X \in \mathcal{U}} X$. The elements of the product $\prod_{X \in \mathcal{U}} X$ are (dependently typed) functions: thus for all $X \in \mathcal{U}$, X is non-empty by **Inhab** and $\mathrm{ch}(X) \in X$ witnesses this.

There are some consequences of these assumptions which will be needed. In set theory functions are identified with their graphs, which are certain sets of ordered pairs. Thus the set $X \to Y$ of all functions from a set X to a set Y is a subset of $P(X \times Y)$; and it is a non-empty set when Y is non-empty. So **Sub**, **Prod** and **Pow** together imply that $\mathcal{U}$ also satisfies

Fun If $X \in \mathcal{U}$ and $Y \in \mathcal{U}$, then $X \to Y \in \mathcal{U}$.

By iterating **Prod**, one has that the cartesian product of any finite, non-zero number of sets in $\mathcal{U}$ is again in $\mathcal{U}$. $\mathcal{U}$ also contains the cartesian product of no sets, which is to say that it contains a one-element set (by virtue of **Sub** applied to any set in $\mathcal{U}$—**Infty** guarantees there is one); for definiteness, a particular one-element set will be singled out.

Unit $\mathcal{U}$ contains a distinguished one-element set $1 = \{0\}$.

Similarly, because of **Sub** and **Infty**, $\mathcal{U}$ contains two-element sets, one of which will be singled out.

Bool $\mathcal{U}$ contains a distinguished two-element set $2 = \{0, 1\}$.

The above assumptions on $\mathcal{U}$ are weaker than those imposed on a universe of sets by the axioms of Zermelo-Fraenkel set theory with the Axiom of Choice (ZFC), principally because $\mathcal{U}$ is not required to satisfy any form of the Axiom of Replacement. Indeed, it is possible to prove the existence of a set $\mathcal{U}$ with the above properties from the axioms of ZFC. (For example one could take $\mathcal{U}$ to consist of all non-empty sets in the von Neumann cumulative hierarchy formed before stage $\omega + \omega$.) Thus, as with many other pieces of mathematics, it is possible in principal to give a completely formal version within ZFC set theory of the semantics of the HOL logic to be given below.

9.2 Types

The types of the HOL logic are expressions that denote sets (in the universe $\mathcal{U}$). Following tradition, σ, possibly decorated with subscripts or primes, is used to range over arbitrary types.

There are four kinds of types in the HOL logic. These can be described informally by the following BNF grammar, in which α ranges over type variables, c ranges over atomic types and op ranges over type operators.

$$\sigma \quad ::= \quad \underbrace{\alpha}_{\text{type variables}} \quad | \quad \underbrace{c}_{\text{atomic types}} \quad | \quad \underbrace{(\sigma_1, \ldots, \sigma_n)op}_{\text{compound types}} \quad | \quad \underbrace{\sigma_1 \rightarrow \sigma_2}_{\substack{\text{function types} \\ \text{(domain } \sigma_1\text{, range } \sigma_2\text{)}}}$$

In more detail, the four kinds of types are as follows.

1. **Type variables:** these stand for arbitrary sets in the universe. In Church's original formulation of simple type theory, type variables are part of the meta-language and are used to range over object language types. Proofs containing type variables were understood as proof schemes (i.e. families of proofs). To support such proof schemes *within* the HOL logic, type variables have been added to the object language type system.[2]

2. **Atomic types:** these denote fixed sets in the universe. Each theory determines a particular collection of atomic types. For example, the standard atomic types *bool* and *ind* denote, respectively, the distinguished two-element set 2 and the distinguished infinite set I.

3. **Compound types:** These have the form $(\sigma_1, \ldots, \sigma_n)op$, where σ_1, ..., σ_n are the argument types and *op* is a *type operator* of arity n. Type operators denote operations for constructing sets. The type $(\sigma_1, \ldots, \sigma_n)op$ denotes the set resulting from applying the operation denoted by *op* to the sets denoted by σ_1, ..., σ_n. For example, *list* is a type operator with arity 1. It denotes the operation of forming all finite lists of elements from a given set. Another example is the type operator *prod* of arity 2 which denotes the cartesian product operation. The type $(\sigma_1, \sigma_2)prod$ is written as $\sigma_1 \times \sigma_2$.

4. **Function types:** If σ_1 and σ_2 are types, then $\sigma_1 \rightarrow \sigma_2$ is the function type with *domain* σ_1 and *range* σ_2. It denotes the set of all (total) functions from the set denoted by its domain to the set denoted by its range. (In the literature $\sigma_1 \rightarrow \sigma_2$ is written without the arrow and backwards—i.e. as $\sigma_2\sigma_1$.) Note that syntactically $\rightarrow$ is simply a distinguished type operator of arity 2 written with infix notation. It is singled out in the definition of HOL types because it will always denote the same operation in any model of a HOL theory—in contrast to the other type operators which may be interpreted differently in different models. (See Section 9.2.2.)

It turns out to be convenient to identify atomic types with compound types constructed with 0-ary type operators. For example, the atomic type *bool* of truth-values can be regarded as being an abbreviation for $()bool$. This identification will be made in the

[2] This technique was invented by Robin Milner for the object logic PPλ of his LCF system.

technical details that follow, but in the informal presentation atomic types will continue to be distinguished from compound types, and $()c$ will still be written as c.

9.2.1 Type structures

The term 'type constant' is used to cover both atomic types and type operators. It is assumed that an infinite set TyNames of the *names of type constants* is given. The greek letter ν is used to range over arbitrary members of TyNames, c will continue to be used to range over the names of atomic types (i.e. 0-ary type constants), and op is used to range over the names of type operators (i.e. n-ary type constants, where $n > 0$).

It is assumed that an infinite set TyVars of *type variables* is given. Greek letters $\alpha, \beta, \ldots$, possibly with subscripts or primes, are used to range over Tyvars. The sets TyNames and TyVars are assumed disjoint.

A *type structure* is a set Ω of type constants. A *type constant* is a pair (ν, n) where $\nu \in$ TyNames is the name of the constant and n is its arity. Thus $\Omega \subseteq$ TyNames $\times$ N (where N is the set of natural numbers). It is assumed that no two distinct type constants have the same name, i.e. whenever $(\nu, n_1) \in \Omega$ and $(\nu, n_2) \in \Omega$, then $n_1 = n_2$.

The set Types_Ω of types over a structure Ω can now be defined as the smallest set such that:

- TyVars $\subseteq \mathsf{Types}_\Omega$.
- If $(\nu, 0) \in \Omega$ then $()\nu \in \mathsf{Types}_\Omega$.
- If $(\nu, n) \in \Omega$ and $\sigma_i \in \mathsf{Types}_\Omega$ for $1 \leq i \leq n$, then $(\sigma_1, \ \ldots \ , \sigma_n)\nu \in \mathsf{Types}_\Omega$.
- If $\sigma_1 \in \mathsf{Types}_\Omega$ and $\sigma_2 \in \mathsf{Types}_\Omega$ then $\sigma_1 \rightarrow \sigma_2 \in \mathsf{Types}_\Omega$.

The type operator $\rightarrow$ is assumed to associate to the right, so that

$$\sigma_1 \rightarrow \sigma_2 \rightarrow \ldots \rightarrow \sigma_n \rightarrow \sigma$$

abbreviates

$$\sigma_1 \rightarrow (\sigma_2 \rightarrow \ldots \rightarrow (\sigma_n \rightarrow \sigma) \ldots)$$

The notation $tyvars(\sigma)$ is used to denote the set of type variables occurring in σ.

9.2.2 Semantics of types

A *model* M of a type structure Ω is specified by giving for each type constant (ν, n) an n-ary function

$$M(\nu) : \mathcal{U}^n \longrightarrow \mathcal{U}$$

Thus given sets $X_1, \ldots, X_n$ in the universe $\mathcal{U}$, $M(\nu)(X_1, \ldots, X_n)$ is also a set in the universe. In case $n = 0$, this amounts to specifying an element $M(\nu) \in \mathcal{U}$ for the atomic type ν.

Types containing no type variables are called *monomorphic*, whereas those that do contain type variables are called *polymorphic*. What is the meaning of a polymorphic type? One can only say what set a polymorphic type denotes once one has instantiated its type variables to particular sets. So its overall meaning is not a single set, but is rather a set-valued function, $\mathcal{U}^n \longrightarrow \mathcal{U}$, assigning a set for each particular assignment of sets to the relevant type variables. The arity n corresponds to the number of type variables involved. It is convenient in this connection to be able to consider a type variable to be involved in the semantics of a type σ whether or not it actually occurs in σ, leading to the notion of a type-in-context.

A *type context*, αs, is simply a finite (possibly empty) list of *distinct* type variables $\alpha_1, \ldots, \alpha_n$. A *type-in-context* is a pair, written $\alpha s.\sigma$, where αs is a type context, σ is a type (over some given type structure) and all the type variables occurring in σ appear somewhere in the list αs. The list αs may also contain type variables which do not occur in σ.

For each σ there are minimal contexts αs for which $\alpha s.\sigma$ is a type-in-context, which only differ by the order in which the type variables of σ are listed in αs. In order to select one such context, let us assume that TyVars comes with a fixed total order and define the *canonical* context of the type σ to consist of exactly the type variables it contains, listed in order.[3]

Let M be a model of a type structure Ω. For each type-in-context $\alpha s.\sigma$ over Ω, define a function

$$[\![\alpha s.\sigma]\!]_M : \mathcal{U}^n \longrightarrow \mathcal{U}$$

(where n is the length of the context) by induction on the structure of σ as follows.

- If σ is a type variable, it must be α_i for some unique $i = 1, \ldots, n$ and then $[\![\alpha s.\sigma]\!]_M$ is the ith projection function, which sends $(X_1, \ldots, X_n) \in \mathcal{U}^n$ to $X_i \in \mathcal{U}$.
- If σ is a function type $\sigma_1 \to \sigma_2$, then $[\![\alpha s.\sigma]\!]_M$ sends $Xs \in \mathcal{U}^n$ to the set of all functions from $[\![\alpha s.\sigma_1]\!]_M(Xs)$ to $[\![\alpha s.\sigma_2]\!]_M(Xs)$. (This makes use of the property **Fun** of $\mathcal{U}$.)
- If σ is a compound type $(\sigma_1, \ldots, \sigma_m)\nu$, then $[\![\alpha s.\sigma]\!]_M$ sends Xs to $M(\nu)(S_1, \ldots, S_m)$ where each S_j is $[\![\alpha s.\sigma_j]\!]_M(Xs)$.

One can now define the meaning of a type σ in a model M to be the function

$$[\![\sigma]\!]_M : \mathcal{U}^n \longrightarrow \mathcal{U}$$

[3] It is possible to work with unordered contexts, specified by finite sets rather than lists, but we choose not to do that since it mildly complicates the definition of the semantics to be given below.

given by $[\![\alpha s.\sigma]\!]_M$, where αs is the canonical context of σ. If σ is monomorphic, then $n = 0$ and $[\![\sigma]\!]_M$ can be identified with the element $[\![\sigma]\!]_M()$ of $\mathcal{U}$. When the particular model M is clear from the context, $[\![_]\!]_M$ will be written $[\![_]\!]$.

To summarize, given a model in $\mathcal{U}$ of a type structure Ω, the semantics interprets monomorphic types over Ω as sets in $\mathcal{U}$ and more generally, interprets polymorphic types involving n type variables as n-ary functions $\mathcal{U}^n \longrightarrow \mathcal{U}$ on the universe. Function types are interpreted by full function sets.

Examples Suppose that Ω contains a type constant $(b, 0)$ and that the model M assigns the set 2 to b. Then:

1. $[\![b{\to}b{\to}b]\!] = 2{\to}2{\to}2 \in \mathcal{U}$.
2. $[\![(\alpha{\to}b){\to}\alpha]\!] : \mathcal{U} \longrightarrow \mathcal{U}$ is the function sending $X \in \mathcal{U}$ to $(X{\to}2){\to}X \in \mathcal{U}$.
3. $[\![\alpha, \beta.(\alpha{\to}b){\to}\alpha]\!] : \mathcal{U}^2 \longrightarrow \mathcal{U}$ is the function sending $(X, Y) \in \mathcal{U}^2$ to $(X{\to}2){\to}X \in \mathcal{U}$.

Remark A more traditional approach to the semantics would involve giving meanings to types in the presence of 'environments' assigning sets in $\mathcal{U}$ to all type variables. The use of types-in-contexts is almost the same as using partial environments with finite domains—it is just that the context ties down the admissible domain to a particular finite (ordered) set of type variables. At the level of types there is not much to choose between the two approaches. However for the syntax and semantics of terms to be given below, where there is a dependency both on type variables and on individual variables, the approach used here seems best.

9.2.3 Instances and substitution

If σ and $\tau_1, \ldots, \tau_n$ are types over a type structure Ω,

$$\sigma[\tau_1, \ldots, \tau_p/\beta_1, \ldots, \beta_p]$$

will denote the type resulting from the simultaneous substitution for each $i = 1, \ldots, p$ of τ_i for the type variable β_i in σ. The resulting type is called an *instance* of σ. The following lemma about instances will be useful later; it is proved by induction on the structure of σ.

Lemma 1 *Suppose that σ is a type containing distinct type variables $\beta_1, \ldots, \beta_p$ and that $\sigma' = \sigma[\tau_1, \ldots, \tau_n/\beta_1, \ldots, \beta_p]$ is an instance of σ. Then the types $\tau_1, \ldots, \tau_p$ are uniquely determined by σ and σ'.*

We also need to know how the semantics of types behaves with respect to substitution:

Lemma 2 *Given types-in-context $\beta s.\sigma$ and $\alpha s.\tau_i$ ($i = 1, \ldots, p$, where p is the length of βs), let σ' be the instance $\sigma[\tau s/\beta s]$. Then $\alpha s.\sigma'$ is also a type-in-context and its meaning*

in any model M is related to that of $\beta s.\sigma$ as follows. For all $Xs \in \mathcal{U}^n$ (where n is the length of αs)

$$[\![\alpha s.\sigma']\!](Xs) = [\![\beta s.\sigma]\!]([\![\alpha s.\tau_1]\!](Xs), \ldots, [\![\alpha s.\tau_p]\!](Xs))$$

Once again, the lemma can be proved by induction on the structure of σ.

9.3 Terms

The terms of the HOL logic are expressions that denote elements of the sets denoted by types. The meta-variable t is used to range over arbitrary terms, possibly decorated with subscripts or primes.

There are four kinds of terms in the HOL logic. These can be described approximately by the following BNF grammar, in which x ranges over variables and c ranges over constants.

$$t \quad ::= \quad \underset{\text{variables}}{x} \quad | \quad \underset{\text{constants}}{\mathsf{c}} \quad | \quad \underbrace{t\ t'}_{\substack{\text{function applications}\\ (\text{function } t,\ \text{argument } t')}} \quad | \quad \underbrace{\lambda x.\ t}_{\lambda\text{-abstractions}}$$

Informally, a λ-term $\lambda x.\ t$ denotes a function $v \mapsto t[v/x]$, where $t[v/x]$ denotes the result of substituting v for x in t. An application $t\ t'$ denotes the result of applying the function denoted by t to the value denoted by t'. This will be made more precise below.

The BNF grammar just given omits mention of types. In fact, each term in the HOL logic is associated with a unique type. The notation t_σ is traditionally used to range over terms of type σ. A more accurate grammar of terms is:

$$t_\sigma \quad ::= \quad x_\sigma \quad | \quad \mathsf{c}_\sigma \quad | \quad (t_{\sigma' \to \sigma}\ t'_{\sigma'})_\sigma \quad | \quad (\lambda x_{\sigma_1}.\ t_{\sigma_2})_{\sigma_1 \to \sigma_2}$$

In fact, just as the definition of types was relative to a particular type structure Ω, the formal definition of terms is relative to a given collection of typed constants over Ω. Assume that an infinite set Names of names is given. A *constant* over Ω is a pair (c, σ), where $\mathsf{c} \in \mathsf{Names}$ and $\sigma \in \mathsf{Types}_\Omega$. A *signature* over Ω is just a set Σ_Ω of such constants.

The set $\mathsf{Terms}_{\Sigma_\Omega}$ of terms over Σ_Ω is defined to be the smallest set closed under the following rules of formation:

1. **Constants:** If $(\mathsf{c}, \sigma) \in \Sigma_\Omega$ and $\sigma' \in \mathsf{Types}_\Omega$ is an instance of σ, then $(\mathsf{c}, \sigma') \in \mathsf{Terms}_{\Sigma_\Omega}$. Terms formed in this way are called *constants* and are written $\mathsf{c}_{\sigma'}$.

2. **Variables:** If $x \in \mathsf{Names}$ and $\sigma \in \mathsf{Types}_\Omega$, then $\mathtt{var}\ x_\sigma \in \mathsf{Terms}_{\Sigma_\Omega}$. Terms formed in this way are called *variables*. The marker `var` is purely a device to distinguish

variables from constants with the same name. A variable $\mathtt{var}\ x_\sigma$ will usually be written as x_σ, if it is clear from the context that x is a variable rather than a constant.

3. **Function applications:** If $t_{\sigma'\to\sigma} \in \mathsf{Terms}_{\Sigma_\Omega}$ and $t'_{\sigma'} \in \mathsf{Terms}_{\Sigma_\Omega}$, then $(t_{\sigma'\to\sigma}\ t'_{\sigma'})_\sigma \in \mathsf{Terms}_{\Sigma_\Omega}$. (Terms formed in this way are sometimes called *combinations*.)

4. **λ-Abstractions:** If $\mathtt{var}\ x_{\sigma_1} \in \mathsf{Terms}_{\Sigma_\Omega}$ and $t_{\sigma_2} \in \mathsf{Terms}_{\Sigma_\Omega}$, then $(\lambda x_{\sigma_1}.\ t_{\sigma_2})_{\sigma_1\to\sigma_2} \in \mathsf{Terms}_{\Sigma_\Omega}$.

Note that it is possible for constants and variables to have the same name. It is also possible for different variables to have the same name, if they have different types.

The type subscript on a term may be omitted if it is clear from the structure of the term or the context in which it occurs what its type must be.

Function application is assumed to associate to the left, so that $t\ t_1\ t_2\ \ldots\ t_n$ abbreviates $(\ \ldots\ ((t\ t_1)\ t_2)\ \ldots\ t_n)$.

The notation $\lambda x_1\ x_2\ \cdots\ x_n.\ t$ abbreviates $\lambda x_1.\ (\lambda x_2.\ \cdots\ (\lambda x_n.\ t)\ \cdots\ )$.

A term is called polymorphic if it contains a type variable. Otherwise it is called monomorphic. Note that a term t_σ may be polymorphic even though σ is monomorphic—for example, $(f_{\alpha\to b}\ x_\alpha)_b$, where b is an atomic type. The expression $tyvars(t_\sigma)$ denotes the set of type variables occurring in t_σ.

An occurrence of a variable x_σ is called *bound* if it occurs within the scope of a textually enclosing λx_σ, otherwise the occurrence is called *free*. Note that λx_σ does not bind $x_{\sigma'}$ if $\sigma \neq \sigma'$. A term in which all occurrences of variables are bound is called *closed*.

9.3.1 Terms-in-context

A *context* αs,xs consists of a type context αs together with a list $xs = x_1, \ldots, x_m$ of distinct variables whose types only contain type variables from the list αs.

The condition that xs contains *distinct* variables needs some comment. Since a variable is specified by both a name and a type, it is permitted for xs to contain repeated names, so long as different types are attached to the names. This aspect of the syntax means that one has to proceed with caution when defining the meaning of type variable instantiation, since instantiation may cause variables to become equal 'accidentally': see Section 9.3.3.

A *term-in-context* αs,$xs.t$ consists of a context together with a term t satisfying the following conditions.

- αs contains any type variable that occurs in xs and t.
- xs contains any variable that occurs freely in t.
- xs does not contain any variable that occurs bound in t.

The context αs,xs may contain (type) variables which do not appear in t. Note that the combination of the second and third conditions implies that a variable cannot have both free and bound occurrences in t. For an arbitrary term, there is always an α-equivalent term which satisfies this condition, obtained by renaming the bound variables as necessary.[4] In the semantics of terms to be given below we will restrict attention to such terms. Then the meaning of an arbitrary term is taken to be the meaning of some α-variant of it having no variable both free and bound. (The semantics will equate α-variants, so it does not matter which is chosen.) Evidently for such a term there is a minimal context αs,xs, unique up to the order in which variables are listed, for which αs,$xs.t$ is a term-in-context. As for type variables, we will assume given a fixed total order on variables. Then the unique minimal context with variables listed in order will be called the *canonical* context of the term t.

9.3.2 Semantics of terms

Let Σ_Ω be a signature over a type structure Ω (see Section 9.3). A *model* M of Σ_Ω is specified by a model of the type structure plus for each constant $(\mathsf{c}, \sigma) \in \Sigma_\Omega$ an element

$$M(\mathsf{c}, \sigma) \in \prod_{Xs \in \mathcal{U}^n} [\![\sigma]\!]_M(Xs)$$

of the indicated cartesian product, where n is the number of type variables occurring in σ. In other words $M(\mathsf{c}, \sigma)$ is a (dependently typed) function assigning to each $Xs \in \mathcal{U}^n$ an element of $[\![\sigma]\!]_M(Xs)$. In the case that $n = 0$ (so that σ is monomorphic), $[\![\sigma]\!]_M$ was identified with a set in $\mathcal{U}$ and then $M(c, \sigma)$ can be identified with an element of that set.

The meaning of HOL terms in such a model will now be described. The semantics interprets closed terms involving no type variables as elements of sets in $\mathcal{U}$ (the particular set involved being derived from the type of the term as in Section 9.2.2). More generally, if the closed term involves n type variables then it is interpreted as an element of a product $\prod_{Xs \in \mathcal{U}^n} Y(Xs)$, where the function $Y : \mathcal{U}^n \longrightarrow \mathcal{U}$ is derived from the type of the term (in a type context derived from the term). Thus the meaning of the term is a (dependently typed) function which, when applied to any meanings chosen for the type variables in the term, yields a meaning for the term as an element of a set in $\mathcal{U}$. On the other hand, if the term involves m free variables but no type variables, then it is interpreted as a function $Y_1 \times \cdots \times Y_m \to Y$ where the sets $Y_1, \ldots, Y_m$ in $\mathcal{U}$ are the interpretations of the types of the free variables in the term and the set $Y \in \mathcal{U}$ is the interpretation of the type of the term; thus the meaning of the term is a function which, when applied to any meanings chosen for the free variables in the term, yields a meaning for the term. Finally, the most general case is of a term involving n type variables and m free variables: it is interpreted

[4] Recall that two terms are said to be α-equivalent if they differ only in the names of their bound variables.

as an element of a product

$$\prod_{Xs \in \mathcal{U}^n} Y_1(Xs) \times \cdots \times Y_m(Xs) \to Y(Xs)$$

where the functions $Y_1, \ldots, Y_m, Y : \mathcal{U}^n \longrightarrow \mathcal{U}$ are determined by the types of the free variables and the type of the term (in a type context derived from the term).

More precisely, given a term-in-context $\alpha s, xs.t$ over Σ_Ω suppose

- t has type τ
- $xs = x_1, \ldots, x_m$ and each x_j has type σ_j
- $\alpha s = \alpha_1, \ldots, \alpha_n$.

Then since $\alpha s, xs.t$ is a term-in-context, $\alpha s.\tau$ and $\alpha s.\sigma_j$ are types-in-context, and hence give rise to functions $[\![\alpha s.\tau]\!]_M$ and $[\![\alpha s.\sigma_j]\!]_M$ from $\mathcal{U}^n$ to $\mathcal{U}$ as in section 9.2.2. The meaning of $\alpha s, xs.t$ in the model M will be given by an element

$$[\![\alpha s, xs.t]\!]_M \in \prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!]_M(Xs) \right) \to [\![\alpha s.\tau]\!]_M(Xs).$$

In other words, given

$$\begin{aligned} Xs &= (X_1, \ldots, X_n) \in \mathcal{U}^n \\ ys &= (y_1, \ldots, y_m) \in [\![\alpha s.\sigma_1]\!]_M(Xs) \times \cdots \times [\![\alpha s.\sigma_m]\!]_M(Xs) \end{aligned}$$

one gets an element $[\![\alpha s, xs.t]\!]_M(Xs)(ys)$ of $[\![\alpha s.\tau]\!]_M(Xs)$. The definition of $[\![\alpha s, xs.t]\!]_M$ proceeds by induction on the structure of the term t, as follows. (As before, the subscript M will be dropped from the semantic brackets $[\![_]\!]$ when the particular model involved is clear from the context.)

- If t is a variable, it must be x_j for some unique $j = 1, \ldots, m$, so $\tau = \sigma_j$ and then $[\![\alpha s, xs.t]\!](Xs)(ys)$ is defined to be y_j.
- Suppose t is a constant $\mathsf{c}_{\sigma'}$, where $(\mathsf{c}, \sigma) \in \Sigma_\Omega$ and σ' is an instance of σ. Then by Lemma 1 of 9.2.3, $\sigma' = \sigma[\tau_1, \ldots, \tau_p / \beta_1, \ldots, \beta_p]$ for uniquely determined types $\tau_1, \ldots, \tau_p$ (where $\beta_1, \ldots, \beta_p$ are the type variables occurring in σ). Then define $[\![\alpha s, xs.t]\!](Xs)(ys)$ to be $M(\mathsf{c}, \sigma)([\![\alpha s.\tau_1]\!](Xs), \ldots, [\![\alpha s.\tau_p]\!](Xs))$, which is an element of $[\![\alpha s.\tau]\!](Xs)$ by Lemma 2 of 9.2.3 (since τ is σ').
- Suppose t is a function application term $(t_1\ t_2)$ where t_1 is of type $\tau' \to \tau$ and t_2 is of type τ'. Then $f = [\![\alpha s, xs.t_1]\!](Xs)(ys)$, being an element of $[\![\alpha s.\tau' \to \tau]\!](Xs)$, is a function from the set $[\![\alpha s.\tau']\!](Xs)$ to the set $[\![\alpha s.\tau]\!](Xs)$ which one can apply to the element $y = [\![\alpha s, xs.t_2]\!](Xs)(ys)$. Define $[\![\alpha s, xs.t]\!](Xs)(ys)$ to be $f(y)$.

- Suppose t is the abstraction term $\lambda x.t_2$where x is of type τ_1 and t_2 of type τ_2. Thus $\tau = \tau_1 \rightarrow \tau_2$ and $[\![\alpha s.\tau]\!](Xs)$ is the function set $[\![\alpha s.\tau_1]\!](Xs) \rightarrow [\![\alpha s.\tau_2]\!](Xs)$. Define $[\![\alpha s,xs.t]\!](Xs)(ys)$ to be the element of this set which is the function sending $y \in [\![\alpha s.\tau_1]\!](Xs)$ to $[\![\alpha s,xs,x.t_2]\!](Xs)(ys,y)$. (Note that since $\alpha s,xs.t$ is a term-in-context, by convention the bound variable x does not occur in xs and thus $\alpha s,xs,x.t_2$ is also a term-in-context.)

Now define the meaning of a term t_τ in a model M to be the dependently typed function

$$[\![t_\tau]\!] \in \prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!](Xs) \right) \rightarrow [\![\alpha s.\tau]\!](Xs)$$

given by $[\![\alpha s,xs.t_\tau]\!]$, where $\alpha s,xs$ is the canonical context of t_τ. So n is the number of type variables in t_τ, αs is a list of those type variables, m is the number of ordinary variables occurring freely in t_τ (assumed to be distinct from the bound variables of t_τ) and the σ_j are the types of those variables. (It is important to note that the list αs, which is part of the canonical context of t, may be strictly bigger than the canonical type contexts of σ_j or τ. So it would not make sense to write just $[\![\sigma_j]\!]$ or $[\![\tau]\!]$ in the above definition.)

If t_τ is a closed term, then $m = 0$ and for each $Xs \in \mathcal{U}^n$ one can identify $[\![t_\tau]\!]$ with the element $[\![t_\tau]\!](Xs)() \in [\![\alpha s.\tau]\!](Xs)$. So for closed terms one gets

$$[\![t_\tau]\!] \in \prod_{Xs \in \mathcal{U}^n} [\![\alpha s.\tau]\!](Xs)$$

where αs is the list of type variables occurring in t_τ and n is the length of that list. If moreover, no type variables occur in t_τ, then $n = 0$ and $[\![t_\tau]\!]$ can be identified with the element $[\![t_\tau]\!]()$ of the set $[\![\tau]\!] \in \mathcal{U}$.

The semantics of terms appears somewhat complicated because of the possible dependency of a term upon both type variables and ordinary variables. Examples of how the definition of the semantics works in practice can be found in Section 10.4.2, where the meaning of several terms denoting logical constants is given.

9.3.3 Substitution

Since terms may involve both type variables and ordinary variables, there are two different operations of substitution on terms which have to be considered—substitution of types for type variables and substitution of terms for variables.

Substituting types for type variables in terms

Suppose t is a term, with canonical context $\alpha s,xs$ say, where $\alpha s = \alpha_1, \ldots, \alpha_n$, $xs = x_1, \ldots, x_m$ and where for $j = 1, \ldots, m$ the type of the variable x_j is σ_j. If $\alpha s'.\tau_i$ $(i = 1, \ldots, n)$ are types-in-context, then substituting the types τ_i for the type variables α_i in the list xs, one obtains a new list of variables xs'. Thus the jth entry of xs has type $\sigma'_j = \sigma_j[\tau s/\alpha s]$. Only substitutions with the following property will be considered.

> In instantiating the type variables αs with the types τs, no two distinct variables in the list xs become equal in the list xs'.[5]

This condition ensures that $\alpha s, xs$ really is a context. Then one obtains a new term-in-context $\alpha s'$,$xs'.t'$ by substituting the types $\tau s = \tau_1, \ldots, \tau_n$ for the type variables αs in t (with suitable renaming of bound occurrences of variables to make them distinct from the variables in xs'). The notation

$$t[\tau s/\alpha s]$$

is used for the term t'.

Lemma 3 *The meaning of $\alpha s'$,$xs'.t'$ in a model is related to that of t as follows. For all $Xs' \in \mathcal{U}^{n'}$ (where n' is the length of $\alpha s'$)*

$$[\![\alpha s',xs'.t']\!](Xs') = [\![t]\!]([\![\alpha s'.\tau_1]\!](Xs'), \ldots, [\![\alpha s'.\tau_n]\!](Xs')).$$

Lemma 2 in 9.2.3 is needed to see that both sides of the above equation are elements of the same set of functions. The validity of the equation is proved by induction on the structure of the term t.

Substituting terms for variables in terms

Suppose t is a term, with canonical context αs,xs say, where $\alpha s = \alpha_1, \ldots, \alpha_n$, $xs = x_1, \ldots, x_m$ and where for $j = 1, \ldots, m$ the type of the variable x_j is σ_j. If one has terms-in-context αs,$xs'.t_j$ for $j = 1, \ldots, m$ with t_j of the same type as x_j, say σ_j, then one obtains a new term-in-context αs,$xs'.t''$ by substituting the terms $ts = t_1, \ldots, t_m$ for the variables xs in t (with suitable renaming of bound occurrences of variables to prevent the free variables of the t_j becoming bound after substitution). The notation

$$t[ts/xs]$$

is used for the term t''.

Lemma 4 *The meaning of αs,$xs'.t''$ in a model is related to that of t as follows. For all $Xs \in \mathcal{U}^n$ and all $ys' \in [\![\alpha s.\sigma'_1]\!] \times \cdots \times [\![\alpha s.\sigma'_{m'}]\!]$ (where σ'_j is the type of x'_j)*

$$[\![\alpha s,xs'.t'']\!](Xs)(ys') = [\![t]\!](Xs)([\![\alpha s,xs'.t_1]\!](Xs)(ys'), \ldots, [\![\alpha s,xs'.t_m]\!](Xs)(ys'))$$

Once again, this result is proved by induction on the structure of the term t.

[5] Such an identification of variables could occur if the variables had the same name component and their types became equal on instantiation.

9.4 Standard notions

Up to now the syntax of types and terms has been very general. To represent the standard formulas of logic it is necessary to impose some specific structure. In particular, every type structure must contain an atomic type *bool* which is intended to denote the distinguished two-element set $2 \in \mathcal{U}$, regarded as a set of truth-values. Logical formulas are then identified with terms of type *bool*. In addition, various logical constants are assumed to be in all signatures. These requirements are formalized by defining the notion of a standard signature.

9.4.1 Standard type structures

A type structure Ω is *standard* if it contains the atomic types *bool* (of booleans or truth-values) and *ind* (of individuals). (In the literature, the symbol o is often used instead of *bool* and ι instead of *ind*.)

A model M of Ω is *standard* if $M(bool)$ and $M(ind)$ are respectively the distinguished sets 2 and I in the universe $\mathcal{U}$.

It will be assumed from now on that type structures and their models are standard.

9.4.2 Standard signatures

A signature Σ_Ω is *standard* if it contains the following three primitive constants:

$\Rightarrow_{bool \to bool \to bool}$

$=_{\alpha \to \alpha \to bool}$

$\varepsilon_{(\alpha \to bool) \to \alpha}$

The intended interpretation of these constants is that $\Rightarrow$ denotes implication, $=_{\sigma \to \sigma \to bool}$ denotes equality on the set denoted by σ, and $\varepsilon_{(\sigma \to bool) \to \sigma}$ denotes a choice function on the set denoted by σ. More precisely, a model M of Σ_Ω will be called *standard* if

- $M(\Rightarrow, bool \to bool \to bool) \in (2 \to 2 \to 2)$ is the standard implication function, sending $b, b' \in 2$ to

$$(b \Rightarrow b') = \begin{cases} 0 & \text{if } b = 1 \text{ and } b' = 0 \\ 1 & \text{otherwise} \end{cases}$$

- $M(=, \alpha \to \alpha \to bool) \in \prod_{X \in \mathcal{U}} . X \to X \to 2$ is the function assigning to each $X \in \mathcal{U}$ the equality test function, sending $x, x' \in X$ to

$$(x =_X x') = \begin{cases} 1 & \text{if } x = x' \\ 0 & \text{otherwise} \end{cases}$$

- $M(\varepsilon, (\alpha \to bool) \to \alpha) \in \prod_{X \in \mathcal{U}} .(X \to 2) \to X$ is the function assigning to each $X \in \mathcal{U}$ the choice function sending $f \in (X \to 2)$ to

$$\mathrm{ch}_X(f) = \begin{cases} \mathrm{ch}(f^{-1}\{1\}) & \text{if } f^{-1}\{1\} \neq \emptyset \\ \mathrm{ch}(X) & \text{otherwise} \end{cases}$$

  where $f^{-1}\{1\} = \{x \in X : f(x) = 1\}$. (Note that $f^{-1}\{1\}$ is in $\mathcal{U}$ when it is non-empty, by the property **Sub** of the universe $\mathcal{U}$ given in Section 9.1. The function ch is given by property **Choice**.)

It will be assumed from now on that signatures and their models are standard.

Remark This particular choice of primitive constants is arbitrary. The standard collection of logical constants includes T ('true'), F ('false'), $\Rightarrow$ ('implies'), $\wedge$ ('and'), $\vee$ ('or'), $\neg$ ('not'), $\forall$ ('for all'), $\exists$ ('there exists'), $=$ ('equals'), ι ('the'), and ε ('a'). This set is redundant, since it can be defined (in a sense explained in Section 10.5.1) from various subsets. In practice, it is necessary to work with the full set of logical constants, and the particular subset taken as primitive is not important. The interested reader can explore this topic further by reading Andrews' book [1] and the references it contains.

Terms of type *bool* are called *formulas*.

The following notational abbreviations are used:

Notation	Meaning
$t_\sigma = t'_\sigma$	$=_{\sigma \to \sigma \to bool}\ t_\sigma\ t'_\sigma$
$t \Rightarrow t'$	$\Rightarrow_{bool \to bool \to bool}\ t_{bool}\ t'_{bool}$
$\varepsilon x_\sigma .\ t$	$\varepsilon_{(\sigma \to bool) \to \sigma}(\lambda x_\sigma .\ t)$

These notations are special cases of general abbreviatory conventions supported by the HOL system. The first two are infixes and the third is a binder (see Section 11.4.3).

Chapter 10

Theories

10.1 Introduction

The result, if any, of a session with the HOL system is an object called a *theory*. This object is closely related to what a logician would call a theory, but there are some differences arising from the needs of mechanical proof. A HOL theory, like a logician's theory, contains sets of types, constants, definitions and axioms. In addition, however, a HOL theory, at any point in time, contains an explicit list of theorems that have already been proved from the axioms and definitions. Logicians have no need to distinguish theorems actually proved from those merely provable; hence they do not normally consider sets of proven theorems as part of a theory; rather, they take the theorems of a theory to be the (often infinite) set of all consequences of the axioms and definitions. A related difference between logicians' theories and HOL theories is that for logicians, theories are static objects, but in HOL they can be thought of as potentially extendable. For example, the HOL system provides tools for adding to theories and combining theories. A typical interaction with HOL consists in combining some existing theories, making some definitions, proving some theorems and then saving the new results.

The purpose of the HOL system is to provide tools to enable well-formed theories to be constructed. The HOL logic is typed: each theory specifies a signature of type and individual constants; these then determine the sets of types and terms as in the previous chapter. All the theorems of such theories are logical consequences of the definitions and axioms of the theory. The HOL system ensures that only well-formed theories can be constructed by allowing theorems to be created only by *formal proof*. Explicating this involves defining what it means to be a theorem, which leads to the description of the proof system of HOL, to be given below. It is shown to be *sound* for the set theoretic semantics of HOL described in the previous chapter. This means that a theorem is satisfied by a model if it has a formal proof from axioms which are themselves satisfied by the model. Since a logical contradiction is not satisfied by any model, this guarantees in particular that a theory possessing a model is necessarily consistent, i.e. a logical contradiction cannot be formally proved from its axioms.

This chapter also describes the various mechanisms by which HOL theories can be extended to new theories. Each mechanism is shown to preserve the property of possessing a model. Thus theories built up from the initial HOL theory (which does possess a model) using these mechanisms are guaranteed to be consistent.

10.2 Sequents

The HOL logic is phrased in terms of hypothetical assertions called *sequents*. Fixing a (standard) signature Σ_Ω, a sequent is a pair (Γ, t) where Γ is a finite set of formulas over Σ_Ω and t is a single formula over Σ_Ω.[1] The set of formulas Γ forming the first component of a sequent is called its set of *assumptions* and the term t forming the second component is called its *conclusion*. When it is not ambiguous to do so, a sequent $(\{\}, t)$ is written as just t.

Intuitively, a model M of Σ_Ω *satisfies* a sequent (Γ, t) if any interpretation of relevant free variables as elements of M making the formulas in Γ true, also makes the formula t true. To make this more precise, suppose $\Gamma = \{t_1, \ldots, t_p\}$ and let αs,xs be a minimal context containing all the type variables and all the free variables occurring in the formulas $t, t_1, \ldots, t_p$. Suppose that αs has length n, that $xs = x_1, \ldots, x_m$ and that the type of x_j is σ_j. Since formulas are terms of type *bool*, the semantics of terms defined in the previous chapter gives rise to elements $[\![\alpha s,xs.t]\!]_M$ and $[\![\alpha s,xs.t_k]\!]_M$ $(k = 1, \ldots, p)$ in

$$\prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!]_M(Xs) \right) \to 2$$

Say that the model M *satisfies* the sequent (Γ, t) and write

$$\Gamma \models_M t$$

if for all $Xs \in \mathcal{U}^n$ and all $ys \in [\![\alpha s.\sigma_1]\!]_M(Xs) \times \cdots \times [\![\alpha s.\sigma_m]\!]_M(Xs)$ with

$$[\![\alpha s,xs.t_k]\!]_M(Xs)(ys) = 1$$

for all $k = 1, \ldots, p$, it is also the case that

$$[\![\alpha s,xs.t]\!]_M(Xs)(ys) = 1.$$

(Recall that 2 is the set $\{0, 1\}$.)

In the case $p = 0$, the satisfaction of $(\{\}, t)$ by M will be written $\models_M t$. Thus $\models_M t$ means that the dependently typed function

$$[\![t]\!]_M \in \prod_{Xs \in \mathcal{U}^n} \left(\prod_{j=1}^{m} [\![\alpha s.\sigma_j]\!]_M(Xs) \right) \to 2$$

is constant with value $1 \in 2$.

[1] Note that the type subscript is omitted from terms when it is clear from the context that they are formulas, i.e. have type *bool*.

10.3 Logic

A deductive system $\mathcal{D}$ is a set of pairs $(L, (\Gamma, t))$ where L is a (possibly empty) list of sequents and (Γ, t) is a sequent.

A sequent (Γ, t) follows from a set of sequents Δ by a deductive system $\mathcal{D}$ if and only if there exist sequents (Γ_1, t_1), ... , (Γ_n, t_n) such that:

1. $(\Gamma, t) = (\Gamma_n, t_n)$, and
2. for all i such that $1 \leq i \leq n$
   (a) either $(\Gamma_i, t_i) \in \Delta$ or
   (b) $(L_i, (\Gamma_i, t_i)) \in \mathcal{D}$ for some list L_i of members of $\Delta \cup \{(\Gamma_1, t_1), \ldots, (\Gamma_{i-1}, t_{i-1})\}$.

The sequence $(\Gamma_1, t_1), \cdots, (\Gamma_n, t_n)$ is called a *proof* of (Γ, t) from Δ with respect to $\mathcal{D}$.

Note that if (Γ, t) follows from Δ, then (Γ, t) also follows from any Δ' such that $\Delta \subseteq \Delta'$. This property is called *monotonicity*.

The notation $t_1, \ldots, t_n \vdash_{\mathcal{D},\Delta} t$ means that the sequent $(\{t_1, \ldots, t_n\},\ t)$ follows from Δ by $\mathcal{D}$. If either $\mathcal{D}$ or Δ is clear from the context then it may be omitted. In the case that there are no hypotheses (i.e. $n = 0$), just $\vdash t$ is written.

In practice, a particular deductive system is usually specified by a number of (schematic) *rules of inference*, which take the form

$$\frac{\Gamma_1 \vdash t_1 \qquad \cdots \qquad \Gamma_n \vdash t_n}{\Gamma \vdash t}$$

The sequents above the line are called the *hypotheses* of the rule and the sequent below the line is called its *conclusion*. Such a rule is schematic because it may contain metavariables standing for arbitrary terms of the appropriate types. Instantiating these metavariables with actual terms, one gets a list of sequents above the line and a single sequent below the line which together constitute a particular element of the deductive system. The instantiations allowed for a particular rule may be restricted by imposing a *side condition* on the rule.

10.3.1 The HOL deductive system

The deductive system of the HOL logic is specified by eight rules of inference, given below. The first three rules have no hypotheses; their conclusions can always be deduced. The identifiers in square brackets are the names of the ML functions in the HOL system that implement the corresponding inference rules (See Section 11.8.5). Any side conditions restricting the scope of a rule are given immediately below it.

Assumption introduction [ASSUME]

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

Reflexivity [REFL]

$$\frac{}{\vdash t = t}$$

Beta-conversion [BETA_CONV]

$$\frac{}{\vdash (\lambda x.\ t_1)t_2 = t_1[t_2/x]}$$

- Where $t_1[t_2/x]$ is the result of substituting t_2 for x in t_1, with suitable renaming of variables to prevent free variables in t_2 becoming bound after substitution.

Substitution [SUBST]

$$\frac{\Gamma_1 \vdash t_1 = t'_1 \quad \cdots \quad \Gamma_n \vdash t_n = t'_n \qquad \Gamma \vdash t[t_1, \ldots, t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \vdash t[t'_1, \ldots, t'_n]}$$

- Where $t[t_1, \ldots, t_n]$ denotes a term t with some free occurrences of subterms $t_1, \ldots, t_n$ singled out and $t[t'_1, \ldots, t'_n]$ denotes the result of replacing each selected occurrence of t_i by t'_i (for $1 \leq i \leq n$), with suitable renaming of variables to prevent free variables in t'_i becoming bound after substitution.

Abstraction [ABS]

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\ t_1) = (\lambda x.\ t_2)}$$

- Provided x is not free in Γ.

Type instantiation [INST_TYPE]

$$\frac{\Gamma \vdash t}{\Gamma \vdash t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

- Where $t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]$ is the result of substituting, in parallel, the types σ_1, ..., σ_n for type variables α_1, ..., α_n in t, with the restrictions:

  (i) none of the type variables $\alpha_1, \ldots, \alpha_n$ occur in Γ;

  (ii) no distinct variables in t become identified after the instantiation.[2]

[2] The ML function implementing INST_TYPE in the HOL system fails if side condition (i) is violated, but instead of failing if (ii) is violated, it automatically renames any variable whose type is instantiated if the variable is preceded in t by a different variable with the same name.

Discharging an assumption [DISCH]

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

- Where $\Gamma - \{t_1\}$ is the set subtraction of $\{t_1\}$ from Γ.

Modus Ponens [MP]

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

In addition to these eight rules, there are also five *axioms* which could have been regarded as rules of inference without hypotheses. This is not done, however, since it is most natural to state the axioms using some defined logical constants and the principle of constant definition has not yet been described. The axioms are given in Section 10.4.3 and the definitions of the extra logical constants they involve are given in Section 10.4.2.

The particular set of rules and axioms chosen to axiomatize the HOL logic is rather arbitrary. It is partly based on the rules that were used in the LCF logic PPλ, since HOL was implemented by modifying the LCF system. In particular, the substitution rule SUBST is exactly the same as the corresponding rule in LCF; the code implementing this was written by Robin Milner and is highly optimized. Because substitution is such a pervasive activity in proof, it was felt to be important that the system primitive be as fast as possible. From a logical point of view it would be better to have a simpler substitution primitive, such as 'Rule R' of Andrews' logic $\mathcal{Q}_0$, and then to derive more complex rules from it.

10.3.2 Soundness theorem

The rules of the HOL *deductive system are* sound *for the notion of satisfaction defined in Section 10.2: for any instance of the rules of inference, if a (standard) model satisfies the hypotheses of the rule it also satisfies the conclusion.*

Proof The verification of the soundness of the rules is straightforward. The properties of the semantics with respect to substitution given by Lemmas 3 and 4 in Section 9.3.3 are needed for rules BETA_CONV, SUBST and INST_TYPE.[3] The fact that $=$ and $\Rightarrow$ are interpreted standardly (as in Section 9.4.2) is needed for rules REFL, BETA_CONV, SUBST, ABS, DISCH and MP.

[3]Note in particular that the second restriction on INST_TYPE enables the result on the semantics of substituting types for type variables in terms to be applied.

10.4 HOL Theories

A HOL *theory* $\mathcal{T}$ is a 4-tuple:

$$\mathcal{T} = \langle \mathsf{Struc}_{\mathcal{T}}, \mathsf{Sig}_{\mathcal{T}}, \mathsf{Axioms}_{\mathcal{T}}, \mathsf{Theorems}_{\mathcal{T}} \rangle$$

where

(i) $\mathsf{Struc}_{\mathcal{T}}$ is a type structure called the type structure of $\mathcal{T}$;

(ii) $\mathsf{Sig}_{\mathcal{T}}$ is a signature over $\mathsf{Struc}_{\mathcal{T}}$ called the signature of $\mathcal{T}$;

(iii) $\mathsf{Axioms}_{\mathcal{T}}$ is a set of sequents over $\mathsf{Sig}_{\mathcal{T}}$ called the axioms of $\mathcal{T}$;

(iv) $\mathsf{Theorems}_{\mathcal{T}}$ is a set of sequents over $\mathsf{Sig}_{\mathcal{T}}$ called the theorems of $\mathcal{T}$, with the property that every member follows from $\mathsf{Axioms}_{\mathcal{T}}$ by the HOL deductive system.

The sets $\mathsf{Types}_{\mathcal{T}}$ and $\mathsf{Terms}_{\mathcal{T}}$ of types and terms of a theory $\mathcal{T}$ are, respectively, the sets of types and terms constructable from the type structure and signature of $\mathcal{T}$, i.e.:

$$\begin{aligned} \mathsf{Types}_{\mathcal{T}} &= \mathsf{Types}_{\mathsf{Struc}_{\mathcal{T}}} \\ \mathsf{Terms}_{\mathcal{T}} &= \mathsf{Terms}_{\mathsf{Sig}_{\mathcal{T}}} \end{aligned}$$

A model of a theory $\mathcal{T}$ is specified by giving a (standard) model M of the underlying signature of the theory with the property that M satisfies all the sequents which are axioms of $\mathcal{T}$. Because of the Soundness Theorem 10.3.2, it follows that M also satisfies any sequents in the set of given theorems, $\mathsf{Theorems}_{\mathcal{T}}$.

10.4.1 The theory MIN

The *minimal theory* `MIN` is defined by:

$$\texttt{MIN} = \langle \{(bool, 0),\ (ind, 0)\},\ \{\Rightarrow_{bool \to bool \to bool}, =_{\alpha \to \alpha \to bool}, \varepsilon_{(\alpha \to bool) \to \alpha}\},\ \{\},\ \{\} \rangle$$

Since the theory `MIN` has a signature consisting only of standard items and has no axioms, it possesses a unique standard model, which will be denoted *Min*.

Although the theory `MIN` contains only the minimal standard syntax, by exploiting the higher order constructs of HOL one can construct a rather rich collection of terms over it. The following theory introduces names for some of these terms that denote useful logical operations in the model *Min*.

10.4.2 The theory LOG

The theory LOG has the same type structure as MIN. Its signature contains the constants in MIN and the following constants:

T_{bool}

$\forall_{(\alpha\to bool)\to bool}$

$\exists_{(\alpha\to bool)\to bool}$

F_{bool}

$\neg_{bool\to bool}$

$\wedge_{bool\to bool\to bool}$

$\vee_{bool\to bool\to bool}$

$\mathsf{One_One}_{(\alpha\to\beta)\to bool}$

$\mathsf{Onto}_{(\alpha\to\beta)\to bool}$

$\mathsf{Type_Definition}_{(\alpha\to bool)\to(\beta\to\alpha)\to bool}$

The following special notation is used in connection with these constants:

Notation	Meaning
$\forall x_\sigma.\ t$	$\forall(\lambda x_\sigma.\ t)$
$\forall x_1\ x_2\ \cdots\ x_n.\ t$	$\forall x_1.\ (\forall x_2.\ \cdots\ (\forall x_n.\ t)\ \cdots)$
$\exists x_\sigma.\ t$	$\exists(\lambda x_\sigma.\ t)$
$\exists x_1\ x_2\ \cdots\ x_n.\ t$	$\exists x_1.\ (\exists x_2.\ \cdots\ (\exists x_n.\ t)\ \cdots)$
$t_1\ \wedge\ t_2$	$\wedge\ t_1\ t_2$
$t_1\ \vee\ t_2$	$\vee\ t_1\ t_2$

The axioms of the theory LOG consist of the following sequents:

$\vdash\ \mathsf{T} = ((\lambda x_{bool}.\ x) = (\lambda x_{bool}.\ x))$
$\vdash\ \forall = \lambda P_{\alpha\to bool}.\ \ P = (\lambda x.\ \mathsf{T})$
$\vdash\ \exists = \lambda P_{\alpha\to bool}.\ \ P(\varepsilon\ P)$
$\vdash\ \mathsf{F} = \forall b_{bool}.\ \ b$
$\vdash\ \neg = \lambda b.\ \ b \Rightarrow \mathsf{F}$
$\vdash\ \wedge = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$
$\vdash\ \vee = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b)$
$\vdash\ \mathsf{One_One} = \lambda f_{\alpha\to\beta}.\ \forall x_1\ x_2.\ (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2)$
$\vdash\ \mathsf{Onto} = \lambda f_{\alpha\to\beta}.\ \forall y.\ \exists x.\ y = f\ x$
$\vdash\ \mathsf{Type_Definition} = \ \lambda P_{\alpha\to bool}\ rep_{\beta\to\alpha}.\mathsf{One_One}\ rep\ \ \wedge$
$(\forall x.\ P\ x\ \ =\ \ (\exists y.\ x = rep\ y))$

Finally, as for the theory MIN, the set $\mathsf{Theorems}_{\mathtt{LOG}}$ is taken to be empty.

Note that the axioms of the theory LOG are essentially *definitions* of the new constants of LOG as terms in the original theory MIN. (The mechanism for making such extensions of theories by definitions of new constants will be set out in general in Section 10.5.1.) The first seven axioms define the logical constants for truth, universal quantification, existential quantification, falsity, negation, conjunction and disjunction. Although these definitions may be obscure to some readers, they are in fact standard definitions of these logical constants in terms of implication, equality and choice within higher order logic. The next two axioms define the properties of a function being one-one and onto; they will be used to express the axiom of infinity (see Section 10.4.3), amongst other things. The last axiom defines a constant used for type definitions (see Section 10.5.4).

The unique standard model *Min* of MIN gives rise to a unique standard model of LOG. This is because, given the semantics of terms set out in Section 9.3.2, to satisfy the above equations one is forced to interpret the new constants in the following way:

- $[\![\mathsf{T}_{bool}]\!] = 1 \in 2$
- $[\![\forall_{(\alpha\to bool)\to bool}]\!] \in \prod_{X\in\mathcal{U}}(X\to 2)\to 2$ sends $X \in \mathcal{U}$ and $f \in X\to 2$ to

$$[\![\forall]\!](X)(f) = \begin{cases} 1 & \text{if } f^{-1}\{1\} = X \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\exists_{(\alpha\to bool)\to bool}]\!] \in \prod_{X\in\mathcal{U}}(X\to 2)\to 2$ sends $X \in \mathcal{U}$ and $f \in X\to 2$ to

$$[\![\exists]\!](X)(f) = \begin{cases} 1 & \text{if } f^{-1}\{1\} \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{F}_{bool}]\!] = 0 \in 2$
- $[\![\neg_{bool\to bool}]\!] \in 2\to 2$ sends $b \in 2$ to

$$[\![\neg]\!](b) = \begin{cases} 1 & \text{if } b = 0 \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\wedge_{bool\to bool\to bool}]\!] \in 2\to 2\to 2$ sends $b, b' \in 2$ to

$$[\![\wedge]\!](b)(b') = \begin{cases} 1 & \text{if } b = 1 = b' \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\vee_{bool\to bool\to bool}]\!] \in 2\to 2\to 2$ sends $b, b' \in 2$ to

$$[\![\vee]\!](b)(b') = \begin{cases} 0 & \text{if } b = 0 = b' \\ 1 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{One_One}_{(\alpha\to\beta)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to Y)\to 2$ sends $(X,Y)\in\mathcal{U}^2$ and $f\in(X\to Y)$ to

$$[\![\mathsf{One_One}]\!](X,Y)(f) = \begin{cases} 0 & \text{if } f(x)=f(x') \text{ for some } x\neq x' \text{ in } X \\ 1 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{Onto}_{(\alpha\to\beta)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to Y)\to 2$ sends $(X,Y)\in\mathcal{U}^2$ and $f\in(X\to Y)$ to

$$[\![\mathsf{Onto}]\!](X,Y)(f) = \begin{cases} 1 & \text{if } \{f(x) : x\in X\} = Y \\ 0 & \text{otherwise} \end{cases}$$

- $[\![\mathsf{Type_Definition}_{(\alpha\to bool)\to(\beta\to\alpha)\to bool}]\!] \in \prod_{(X,Y)\in\mathcal{U}^2}(X\to 2)\to(Y\to X)\to 2$ sends $(X,Y)\in\mathcal{U}^2$, $f\in(X\to 2)$ and $g\in(Y\to X)$ to

$$[\![\mathsf{Type_Definition}]\!](X,Y)(f)(g) = \begin{cases} 1 & \text{if } [\![\mathsf{One_One}]\!](Y,X)(g)=1 \\ & \text{and } f^{-1}\{1\} = \{g(y) : y\in Y\} \\ 0 & \text{otherwise.} \end{cases}$$

Since these definitions were obtained by applying the semantics of terms to the left hand sides of the equations which form the axioms of LOG, these axioms are satisfied and one obtains a model of the theory LOG.

10.4.3 The theory INIT

The theory INIT is obtained by adding the following five axioms to the theory LOG.

BOOL_CASES_AX	$\vdash \forall b.\ (b=\mathsf{T})\vee(b=\mathsf{F})$
IMP_ANTISYM_AX	$\vdash \forall b_1\ b_2.\ (b_1\Rightarrow b_2)\Rightarrow(b_2\Rightarrow b_1)\Rightarrow(b_1=b_2)$
ETA_AX	$\vdash \forall f_{\alpha\to\beta}.\ (\lambda x.\ f\ x)=f$
SELECT_AX	$\vdash \forall P_{\alpha\to bool}\ x.\ P\ x\Rightarrow P(\varepsilon\ P)$
INFINITY_AX	$\vdash \exists f_{ind\to ind}.\ \mathsf{One_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$

The unique standard model of LOG satisfies these five axioms and hence is the unique standard model of the theory INIT. (For axiom SELECT_AX one needs to use the definition of $[\![\varepsilon]\!]$ given in Section 9.4.2; for axiom INFINITY_AX one needs the fact that $[\![ind]\!] = \mathrm{I}$ is an infinite set.)

The theory INIT is the initial theory of the HOL logic. A theory which extends INIT will be called a *standard theory*.

10.4.4 Consistency

A (standard) theory is *consistent* if it is not the case that every sequent over its signature can be derived from the theory's axioms using the HOL logic, or equivalently, if the particular sequent $\vdash \mathsf{F}$ cannot be so derived.

The existence of a (standard) model of a theory is sufficient to establish its consistency. For by the Soundness Theorem 10.3.2, any sequent that can be derived from the theory's axioms will be satisfied by the model, whereas the sequent $\vdash \mathsf{F}$ is never satisfied in any standard model. So in particular, the initial theory `INIT` is consistent.

However, it is possible for a theory to be consistent but not to possess a standard model. This is because the notion of a *standard* model is quite restrictive—in particular there is no choice how to interpret the integers and their arithmetic in such a model. The famous incompleteness theorem of Gödel ensures that there are sequents which are satisfied in all standard models (i.e. which are 'true'), but which are not provable in the HOL logic.

10.5 Extensions of theories

A theory $\mathcal{T}'$ is said to be an *extension* of a theory $\mathcal{T}$ if:

(i) $\mathsf{Struc}_{\mathcal{T}} \subseteq \mathsf{Struc}_{\mathcal{T}'}$.

(ii) $\mathsf{Sig}_{\mathcal{T}} \subseteq \mathsf{Sig}_{\mathcal{T}'}$.

(iii) $\mathsf{Axioms}_{\mathcal{T}} \subseteq \mathsf{Axioms}_{\mathcal{T}'}$.

(iv) $\mathsf{Theorems}_{\mathcal{T}} \subseteq \mathsf{Theorems}_{\mathcal{T}'}$.

In this case, any model M' of the larger theory $\mathcal{T}'$ can be restricted to a model of the smaller theory $\mathcal{T}$ in the following way. First, M' gives rise to a model of the structure and signature of $\mathcal{T}$ simply by forgetting the values of M' at constants not in $\mathsf{Struc}_{\mathcal{T}}$ or $\mathsf{Sig}_{\mathcal{T}}$. Denoting this model by M, one has for all $\sigma \in \mathsf{Types}_{\mathcal{T}}$, $t \in \mathsf{Terms}_{\mathcal{T}}$ and for all suitable contexts that

$$\begin{aligned} [\![\alpha s.\sigma]\!]_M &= [\![\alpha s.\sigma]\!]_{M'} \\ [\![\alpha s,xs.t]\!]_M &= [\![\alpha s,xs.t]\!]_{M'}. \end{aligned}$$

Consequently if (Γ, t) is a sequent over $\mathsf{Sig}_{\mathcal{T}}$ (and hence also over $\mathsf{Sig}_{\mathcal{T}'}$), then $\Gamma \models_M t$ if and only if $\Gamma \models_{M'} t$. Since $\mathsf{Axioms}_{\mathcal{T}} \subseteq \mathsf{Axioms}_{\mathcal{T}'}$ and M' is a model of $\mathcal{T}'$, it follows that M is a model of $\mathcal{T}$. M will be called the *restriction* of the model M' of the theory $\mathcal{T}'$ to the subtheory $\mathcal{T}$.

There are two main mechanisms for making extensions of theories in HOL:

- Extension by a constant specification (see Section 10.5.2).

- Extension by a type specification (see Section 10.5.5).[4]

The first mechanism allows 'loose specifications' of constants as in the **Z** notation [18]; the latter allows new types and type-operators to be introduced. As special cases (when the thing being specified is uniquely determined) one also has:

- Extension by a constant definition (see Section 10.5.1).
- Extension by a type definition (see Section 10.5.4).

These mechanisms are described in the following sections. They all produce *definitional extensions* in the sense that they extend a theory by adding new constants and types which are defined in terms of properties of existing ones. Their key property is that the extended theory possesses a (standard) model if the original theory does. So a series of these extensions starting from the theory `INIT` is guaranteed to result in a theory with a standard model, and hence in a consistent theory. It is also possible to extend theories simply by adding new uninterpreted constants and types. This preserves consistency, but is unlikely to be useful without additional axioms. However, when adding arbitrary new axioms, there is no guarantee that consistency is preserved. The advantages of postulation over definition have been likened by Bertrand Russell to the advantages of theft over honest toil.[5] As it is all too easy to introduce inconsistent axiomatizations, users of the HOL system are strongly advised to resist the temptation to add axioms, but to toil through definitional theories honestly.

10.5.1 Extension by constant definition

A *constant definition* over a signature Σ_Ω is a formula of the form $\mathsf{c}_\sigma = t_\sigma$, such that:

(i) c is not the name of any constant in Σ_Ω;

(ii) t_σ a closed term in $\mathsf{Terms}_{\Sigma_\Omega}$.

(iii) all the type variables occurring in t_σ also occur in σ

Given a theory $\mathcal{T}$ and such a constant definition over $\mathsf{Sig}_\mathcal{T}$, then the *definitional extension* of $\mathcal{T}$ by $\mathsf{c}_\sigma = t_\sigma$ is the theory $\mathcal{T} +_{def} \langle \mathsf{c}_\sigma = t_\sigma \rangle$ defined by:

$$\begin{aligned} \mathcal{T} +_{def} \langle \mathsf{c}_\sigma = t_\sigma \rangle \;=\; \langle\; & \mathsf{Struc}_\mathcal{T},\ \mathsf{Sig}_\mathcal{T} \cup \{(\mathsf{c}, \sigma)\}, \\ & \mathsf{Axioms}_\mathcal{T} \cup \{\mathsf{c}_\sigma = t_\sigma\},\ \mathsf{Theorems}_\mathcal{T} \rangle \end{aligned}$$

[4] This theory extension mechanism is not implemented in Version 2.0 of the HOL system.

[5] See page 71 of Russell's book *Introduction to Mathematical Philosophy*.

Note that the mechanism of extension by constant definition has already been used implicitly in forming the theory LOG from the theory MIN in Section 10.4.2. Thus with the notation of this section one has

$$\begin{aligned}
\mathtt{LOG} \;=\; \mathtt{MIN} &+_{def} \langle \mathsf{T} \;=\; ((\lambda x_{bool}.\ x) = (\lambda x_{bool}.\ x)) \rangle \\
&+_{def} \langle \forall \;=\; \lambda P_{\alpha\to bool}.\ \ P = (\lambda x.\ \mathsf{T}) \rangle \\
&+_{def} \langle \exists \;=\; \lambda P_{\alpha\to bool}.\ \ P(\varepsilon\ P) \rangle \\
&+_{def} \langle \mathsf{F} \;=\; \forall b_{bool}.\ \ b \rangle \\
&+_{def} \langle \neg \;=\; \lambda b.\ \ b \Rightarrow \mathsf{F} \rangle \\
&+_{def} \langle \wedge \;=\; \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b \rangle \\
&+_{def} \langle \vee \;=\; \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow ((b_2 \Rightarrow b) \Rightarrow b) \rangle \\
&+_{def} \langle \mathsf{One_One} \;=\; \lambda f_{\alpha\to\beta}.\ \forall x_1\ x_2.\ (f\ x_1 = f\ x_2) \Rightarrow (x_1 = x_2) \rangle \\
&+_{def} \langle \mathsf{Onto} \;=\; \lambda f_{\alpha\to\beta}.\ \forall y.\ \exists x.\ y = f\ x \rangle \\
&+_{def} \langle \mathsf{Type_Definition} \;=\; \lambda P_{\alpha\to bool}\ rep_{\beta\to\alpha}. \\
&\qquad\qquad \mathsf{One_One}\ rep\ \wedge \\
&\qquad\qquad (\forall x.\ P\ x \;=\; (\exists y.\ x = rep\ y)) \rangle
\end{aligned}$$

If $\mathcal{T}$ possesses a standard model then so does the extension $\mathcal{T} +_{def} \langle \mathsf{c}_\sigma \;=\; t_\sigma \rangle$. This will be proved as a corollary of the corresponding result in Section 10.5.2 by showing that extension by constant definition is in fact a special case of extension by constant specification. (This reduction requires that one is dealing with *standard* theories in the sense of section 10.4.3, since although existential quantification is not needed for constant definitions, it is needed to state the mechanism of constant specification.)

Remark Condition (iii) in the definition of what constitutes a correct constant definition is an important restriction without which consistency could not be guaranteed. To see this, consider the term $\exists f_{\alpha\to\alpha}.\ \mathsf{One_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$, which expresses the proposition that (the set of elements denoted by the) type α is infinite. The term contains the type variable α, whereas the type of the term, *bool*, does not. Thus by (iii)

$$\mathsf{c}_{bool} = \exists f_{\alpha\to\alpha}.\ \mathsf{One_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$$

is not allowed as a constant definition. The problem is that the meaning of the right hand side of the definition varies with α, whereas the meaning of the constant on the left hand side is fixed, since it does not contain α. Indeed, if we were allowed to extend the consistent theory INIT by this definition, the result would be an inconsistent theory. For instantiating α to *ind* in the right hand side results in a term that is provable from the axioms of INIT, and hence $\mathsf{c}_{bool} = \mathsf{T}$ is provable in the extended theory. But equally, instantiating α to *bool* makes the negation of the right hand side provable from the axioms of INIT, and hence $\mathsf{c}_{bool} = \mathsf{F}$ is also provable in the extended theory. Combining these theorems, one has that $\mathsf{T} = \mathsf{F}$, i.e. F is provable in the extended theory.

10.5.2 Extension by constant specification

Constant specifications introduce constants (or sets of constants) that satisfy arbitrary given (consistent) properties. For example, a theory could be extended by a constant specification to have two new constants b_1 and b_2 of type *bool* such that $\neg(\mathsf{b}_1 = \mathsf{b}_2)$. This specification does not uniquely define b_1 and b_2, since it is satisfied by either $\mathsf{b}_1 = \mathsf{T}$ and $\mathsf{b}_2 = \mathsf{F}$, or $\mathsf{b}_1 = \mathsf{F}$ and $\mathsf{b}_2 = \mathsf{T}$. To ensure that such specifications are consistent, they can only be made if it has already been proved that the properties which the new constants are to have are consistent. This rules out, for example, introducing three boolean constants b_1, b_2 and b_3 such that $\mathsf{b}_1 \neq \mathsf{b}_2$, $\mathsf{b}_1 \neq \mathsf{b}_3$ and $\mathsf{b}_2 \neq \mathsf{b}_3$.

Suppose $\exists x_1 \cdots x_n.\ t$ is a formula, with $x_1, \ldots, x_n$ distinct variables. If $\vdash\ \exists x_1 \cdots x_n.\ t$, then a constant specification allows new constants $\mathsf{c}_1, \ldots, \mathsf{c}_n$ to be introduced satisfying:

$$\vdash\ t[\mathsf{c}_1, \cdots, \mathsf{c}_n / x_1, \cdots, x_n]$$

where $t[\mathsf{c}_1, \cdots, \mathsf{c}_n / x_1, \cdots, x_n]$ denotes the result of simultaneously substituting $\mathsf{c}_1, \ldots, \mathsf{c}_n$ for $x_1, \ldots, x_n$ respectively. Of course the type of each constant c_i must be the same as the type of the corresponding variable x_i. To ensure that this extension mechanism preserves the property of possessing a model, a further more technical requirement is imposed on these types: they must each contain all the type variables occurring in t. This condition is discussed further in Section 10.5.3 below.

Formally, a *constant specification* for a theory $\mathcal{T}$ is given by

Data

$$\langle(\mathsf{c}_1, \ldots, \mathsf{c}_n), \lambda x_{1\sigma_1}, \ldots, x_{n\sigma_n}.\ t_{bool}\rangle$$

Conditions

(i) $\mathsf{c}_1, \ldots, \mathsf{c}_n$ are distinct names that are not the names of any constants in $\mathsf{Sig}_{\mathcal{T}}$.

(ii) $\lambda x_{1\sigma_1} \cdots x_{n\sigma_n}.\ t_{bool}\ \in\ \mathsf{Terms}_{\mathcal{T}}$.

(iii) $tyvars(t_{bool})\ =\ tyvars(\sigma_i)$ for $1 \leq i \leq n$.

(iv) $\exists x_{1\sigma_1}\ \cdots\ x_{n\sigma_n}.\ t\ \in\ \mathsf{Theorems}_{\mathcal{T}}$.

The extension of a standard theory $\mathcal{T}$ by such a constant specification is denoted by

$$\mathcal{T} +_{spec} \langle(\mathsf{c}_1, \ldots, \mathsf{c}_n), \lambda x_{1\sigma_1}, \ldots, x_{n\sigma_n}.\ t_{bool}\rangle$$

and is defined to be the theory:

$$\begin{array}{l}\langle \mathsf{Struc}_{\mathcal{T}},\\ \ \mathsf{Sig}_{\mathcal{T}} \cup \{\mathsf{c}_{1\sigma_1}, \ldots, \mathsf{c}_{n\sigma_n}\},\\ \ \mathsf{Axioms}_{\mathcal{T}} \cup \{t[\mathsf{c}_1, \ldots, \mathsf{c}_n / x_1, \ldots, x_n]\},\\ \ \mathsf{Theorems}_{\mathcal{T}}\rangle\end{array}$$

Proposition *The theory* $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$ *has a standard model if the theory* $\mathcal{T}$ *does.*

Proof Suppose M is a standard model of $\mathcal{T}$. Let $\alpha s = \alpha_1,\ldots,\alpha_m$ be the list of distinct type variables occurring in the formula t. Then $\alpha s,xs.t$ is a term-in-context, where $xs = x_1,\ldots,x_n$. (Change any bound variables in t to make them distinct from xs if necessary.) Interpreting this term-in-context in the model M yields

$$[\![\alpha s,xs.t]\!]_M \in \prod_{Xs\in\mathcal{U}^m}\left(\prod_{i=1}^{n}[\![\alpha s.\sigma_i]\!]_M(Xs)\right)\to 2$$

Now $\exists xs.\ t$ is in $\mathsf{Theorems}_{\mathcal{T}}$ and hence by the Soundness Theorem 10.3.2 this sequent is satisfied by M. Using the semantics of $\exists$ given in Section 10.4.2, this means that for all $Xs\in\mathcal{U}^m$ the set

$$S(Xs) = \{ys\in[\![\alpha s.\sigma_1]\!]_M(Xs)\times\cdots\times[\![\alpha s.\sigma_n]\!]_M(Xs)\ :\ [\![\alpha s,xs.t]\!]_M(Xs)(ys)=1\}$$

is non-empty. Since it is also a subset of a finite product of sets in $\mathcal{U}$, it follows that it is an element of $\mathcal{U}$ (using properties **Sub** and **Prod** of the universe). So one can apply the global choice function $\mathrm{ch}\in\prod_{X\in\mathcal{U}}X$ to select a specific element

$$(s_1(Xs),\ldots,s_n(Xs)) = \mathrm{ch}(S(Xs))\in\prod_{i=1}^{n}[\![\alpha s.\sigma_i]\!]_M(Xs)$$

at which $[\![\alpha s,xs.t]\!]_M(Xs)$ takes the value 1. Extend M to a model M' of the signature of $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$ by defining its value at each new constant (c_i,σ_i) to be

$$M'(\mathsf{c}_i,\sigma_i) = s_i\in\prod_{Xs\in\mathcal{U}^m}[\![\sigma_i]\!]_M(Xs).$$

Note that the Condition (iii) in the definition of a constant specification ensures that αs is the canonical context of each type σ_i, so that $[\![\sigma_i]\!] = [\![\alpha s.\sigma_i]\!]$ and thus s_i is indeed an element of the above product.

Since t is a term of the subtheory $\mathcal{T}$ of $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$, as remarked at the beginning of Section 10.5, one has that $[\![\alpha s,xs.t]\!]_{M'} = [\![\alpha s,xs.t]\!]_M$. Hence by definition of the s_i, for all $Xs\in\mathcal{U}^m$

$$[\![\alpha s,xs.t]\!]_{M'}(Xs)(s_1(Xs),\ldots,s_n(Xs)) = 1$$

Then using Lemma 4 in Section 9.3.3 on the semantics of substitution together with the definition of $[\![\mathsf{c}_i]\!]_{M'}$, one finally obtains that for all $Xs\in\mathcal{U}^m$

$$[\![t[\mathsf{c}_1,\ldots,\mathsf{c}_n/x_1,\ldots,x_n]]\!]_{M'}(Xs) = 1$$

or in other words that M' satisfies $t[\mathsf{c}_1,\ldots,\mathsf{c}_n/x_1,\ldots,x_n]$. Hence M' is a model of $\mathcal{T}+_{spec}\langle(\mathsf{c}_1,\ldots,\mathsf{c}_n),\lambda x_{1\sigma_1},\ldots,x_{n\sigma_n}.\ t_{bool}\rangle$, as required.

The constants which are asserted to exist in a constant specification are not necessarily uniquely determined. Correspondingly, there may be many different models of $\mathcal{T} +_{spec} \langle (\mathsf{c}_1, \ldots, \mathsf{c}_n), \lambda x_{1\sigma_1}, \ldots, x_{n\sigma_n}.\ t_{bool} \rangle$ whose restriction to $\mathcal{T}$ is M; the above construction produces such a model in a uniform manner by making use of the global choice function on the universe.

Extension by a constant definition, $\mathsf{c}_\sigma = t_\sigma$, is a special case of extension by constant specification. For let t' be the formula $x_\sigma = t_\sigma$, where x_σ is a variable not occurring in t_σ. Then clearly $\vdash\ \exists x_\sigma.\ t'$ and one can apply the method of constant specification to obtain the theory

$$\mathcal{T} +_{spec} \langle \mathsf{c}, \lambda x_\sigma.\ t' \rangle$$

But since $t'[\mathsf{c}_\sigma / x_\sigma]$ is just $\mathsf{c}_\sigma = t_\sigma$, this extension yields exactly $\mathcal{T} +_{def} \langle \mathsf{c}_\sigma = t_\sigma \rangle$. So as a corollary of the Proposition, one has that for each standard model M of $\mathcal{T}$, there is a standard model M' of $\mathcal{T} +_{def} \langle \mathsf{c}_\sigma = t_\sigma \rangle$ whose restriction to $\mathcal{T}$ is M. In contrast with the case of constant specifications, M' is uniquely determined by M and the constant definition.

10.5.3 Remarks about constants in HOL

Note how Condition (iii) in the definition of a constant specification was needed in the proof that the extension mechanism preserves the property of possessing a standard model. Its role is to ensure that the introduced constants have, via their types, the same dependency on type variables as does the formula loosely specifying them. The situation is the same as that discussed in the Remark in Section 10.5.1. In a sense, what is causing the problem in the example given in that Remark is not so much the method of extension by introducing constants, but rather the syntax of HOL which does not allow constants to depend explicitly on type variables (in the way that type operators can). Thus in the example one would like to introduce a 'polymorphic' constant $\mathsf{c}_{bool}(\alpha)$ explicitly depending upon α, and define it to be $\exists f_{\alpha\to\alpha}.\ \mathsf{One_One}\ f\ \wedge\ \neg(\mathsf{Onto}\ f)$. Then in the extended theory one could derive $\mathsf{c}_{bool}(ind) = \mathsf{T}$ and $\mathsf{c}_{bool}(bool) = \mathsf{F}$, but now no contradiction results since $\mathsf{c}_{bool}(ind)$ and $\mathsf{c}_{bool}(bool)$ are different.

In the current version of HOL, constants are (name,type)-pairs. One can envision a slight extension of the HOL syntax with 'polymorphic' constants, specified by pairs $(\mathsf{c}, \alpha s.\sigma)$ where now $\alpha s.\sigma$ is a type-in-context and the list αs may well contain extra type variables not occurring in σ. Such a pair would give rise to the particular constant term $\mathsf{c}_\sigma(\alpha s)$, and more generally to constant terms $\mathsf{c}_{\sigma'}(\tau s)$ obtained from this one by instantiating the type variables α_i with types τ_i (so σ' is the instance of σ obtained by substituting τs for αs). This new syntax of polymorphic constants is comparable to the existing syntax of compound types (see section 9.2): an n-ary type operator op gives rise to a compound type $(\alpha_1, \ldots, \alpha_n)op$ depending upon n type variables. Similarly, the above

syntax of polymorphic constants records how they depend upon type variables (as well as which generic type the constant has).

However, explicitly recording dependency of constants on type variables makes for a rather cumbersome syntax which in practice one would like to avoid where possible. It is possible to avoid it if the type context αs in $(\mathsf{c}, \alpha s.\sigma)$ is actually the *canonical* context of σ, i.e. contains exactly the type variables of σ. For then one can apply Lemma 1 of Section 9.2.3 to deduce that the polymorphic constant $\mathsf{c}_{\sigma'}(\tau s)$ can be abbreviated to the ordinary constant $\mathsf{c}_{\sigma'}$ without ambiguity—the missing information τs can be reconstructed from σ' and the information about the constant c given in the signature. From this perspective, the rather technical side Conditions (iii) in Sections 10.5.1 and 10.5.2 become rather less mysterious: they precisely ensure that in introducing new constants one is always dealing just with canonical contexts, and so can use ordinary constants rather than polymorphic ones without ambiguity. In this way one avoids complicating the existing syntax at the expense of restricting somewhat the applicability of these theory extension mechanisms.

10.5.4 Extension by type definition

Every (monomorphic) type σ in the initial theory `INIT` determines a set $[\![\sigma]\!]$ in the universe $\mathcal{U}$. However, there are many more sets in $\mathcal{U}$ than there are types in `INIT`. In particular, whilst $\mathcal{U}$ is closed under the operation of taking a non-empty subset of $[\![\sigma]\!]$, there is no corresponding mechanism for forming a 'subtype' of σ. Instead, subsets are denoted indirectly via characteristic functions, whereby a closed term p of type $\sigma \to bool$ determines the subset $\{x \in [\![\sigma]\!] : [\![p]\!](x) = 1\}$ (which is a set in the universe provided it is non-empty). However, it is useful to have a mechanism for introducing new types which are subtypes of existing ones. Such types are defined in HOL by introducing a new type constant and asserting an axiom that characterizes it as denoting a set in bijection (i.e. one-to-one correspondence) with a non-empty subset of an existing type (called the *representing type*). For example, the type `num` is defined to be equal to a countable subset of the type `ind`, which is guaranteed to exist by the axiom `INFINITY_AX` (see Section 10.4.3).

As well as defining types, it is also convenient to be able to define type operators. An example would be a type operator *inj* which mapped a set to the set of one-to-one (i.e. injective) functions on it. The subset of $\sigma \to \sigma$ representing $(\sigma)inj$ would be defined by the predicate One_One. Another example would be a binary cartesian product type operator *prod*. This is defined by choosing a representing type containing two type variables, say $\sigma[\alpha_1; \alpha_2]$, such that for any types σ_1 and σ_2, a subset of $\sigma[\sigma_1; \sigma_2]$ represents the cartesian product of σ_1 and σ_2. The details of such a definition are given in Section 11.9.2.1.

Types in HOL must denote non-empty sets. Thus it is only consistent to define a new type isomorphic to a subset specified by a predicate p, if there is at least one thing for which p holds, i.e. $\vdash \exists x.\ p\ x$. For example, it would be inconsistent to define a binary

type operator *iso* such that $(\sigma_1, \sigma_2)iso$ denoted the set of one-to-one functions from σ_1 *onto* σ_2 because for some values of σ_1 and σ_2 the set would be empty; for example $(ind, bool)iso$ would denote the empty set. To avoid this, a precondition of defining a new type is that the representing subset is non-empty.

To summarize, a new type is defined by:

1. Specifying an existing type.
2. Specifying a subset of this type.
3. Proving that this subset is non-empty.
4. Specifying that the new type is isomorphic to this subset.

In more detail, defining a new type $(\alpha_1, \ldots, \alpha_n)op$ consists in:

1. Specifying a type-in-context, $\alpha_1, \ldots, \alpha_n.\sigma$ say. The type σ is called the *representing type*, and the type $(\alpha_1, \ldots, \alpha_n)op$ is intended to be isomorphic to a subset of σ.
2. Specifying a closed term-in-context, $\alpha_1, \ldots, \alpha_n, .p$ say, of type $\sigma \rightarrow bool$. The term p is called the *characteristic function*. This defines the subset of σ to which $(\alpha_1, \ldots, \alpha_n)op$ is to be isomorphic.[6]
3. Proving $\vdash \exists x_\sigma.\ p\ x$.
4. Asserting an axiom saying that $(\alpha_1, \ldots, \alpha_n)op$ is isomorphic to the subset of σ selected by p.

To make this formal, the theory LOG provides the polymorphic constant Type_Definition defined in Section 10.4.2. The formula $\exists f_{(\alpha_1,\ldots,\alpha_n)op \rightarrow \sigma}.$ Type_Definition $p\ f$ asserts that there exists a one-to-one map f from $(\alpha_1, \ldots, \alpha_n)op$ onto the subset of elements of σ for which p is true. Hence, the axiom that characterizes $(\alpha_1, \ldots, \alpha_n)op$ is:

$$\vdash \exists f_{(\alpha_1,\ldots,\alpha_n)op \rightarrow \sigma}.\ \text{Type_Definition}\ p\ f$$

Defining a new type $(\alpha_1, \ldots, \alpha_n)op$ in a theory $\mathcal{T}$ thus consists of introducing op as a new n-ary type operator and the above axiom as a new axiom. Formally, a *type definition* for a theory $\mathcal{T}$ is given by

Data

$$\langle (\alpha_1, \ldots, \alpha_n)op,\ \sigma,\ p_{\sigma \rightarrow bool} \rangle$$

Conditions

[6] The reason for restricting p to be closed, i.e. to have no free variables, is that otherwise for consistency the defined type operator would have to *depend* upon (i.e. be a function of) those variables. Such dependent types are not (yet!) a part of the HOL system.

(i) (op, n) is not the name of a type constant in $\mathsf{Struc}_{\mathcal{T}}$.

(ii) $\alpha_1, \ldots, \alpha_n.\sigma$ is a type-in-context with $\sigma \in \mathsf{Types}_{\mathcal{T}}$.

(iii) $p_{\sigma \to bool}$ is a closed term in $\mathsf{Terms}_{\mathcal{T}}$ whose type variables occur in $\alpha_1, \ldots, \alpha_n$.

(iv) $\exists x_\sigma.\ p\ x\ \in\ \mathsf{Theorems}_{\mathcal{T}}$.

The extension of a standard theory $\mathcal{T}$ by a such a type definition is denoted by

$$\mathcal{T} +_{tydef} \langle (\alpha_1, \ldots, \alpha_n) op, \sigma, p \rangle$$

and defined to be the theory

$$\begin{array}{l} \langle \mathsf{Struc}_{\mathcal{T}} \cup \{(op, n)\}, \\ \ \mathsf{Sig}_{\mathcal{T}}, \\ \ \mathsf{Axioms}_{\mathcal{T}} \cup \{\exists f_{(\alpha_1,\ldots,\alpha_n) op \to \sigma}.\ \mathsf{Type_Definition}\ p\ f\}, \\ \ \mathsf{Theorems}_{\mathcal{T}} \rangle \end{array}$$

Proposition *The theory $\mathcal{T} +_{tydef} \langle (\alpha_1, \ldots, \alpha_n) op, \sigma, p \rangle$ has a standard model if the theory $\mathcal{T}$ does.*

Instead of giving a direct proof of this result, it will be deduced as a corollary of the corresponding proposition in the next section.

10.5.5 Extension by type specification[7]

The type definition mechanism allows one to introduce new types by giving a concrete representation of the type as a 'subtype' of an existing type. One might instead wish to introduce a new type satisfying some property without having to give an explicit representation for the type. For example, one might want to extend `INIT` with an atomic type *one* satisfying $\vdash\ \forall f_{\alpha \to one}\ g_{\alpha \to one}.\ f = g$ without choosing a specific type in `INIT` and saying that *one* is in bijection with a one-element subset of it. (The idea being that the choice of representing type is irrelevant to the properties of *one* that can be expressed in HOL.) The mechanism described in this section provides one way of achieving this while at the same time preserving the all-important property of possessing a standard model and hence maintaining consistency.

Each closed formula q involving a single type variable α can be thought of as specifying a property $q[\tau/\alpha]$ of types τ. Its interpretation in a model is of the form

$$[\![\alpha, .q]\!] \in \prod_{X \in \mathcal{U}} [\![\alpha.bool]\!](X)\ =\ \prod_{X \in \mathcal{U}} 2\ =\ \mathcal{U} \to 2$$

[7] This theory extension mechanism is not implemented in Version 2.0 of the HOL system. It was proposed by T. Melham and refines a suggestion from R. Jones and R. Arthan.

which is a characteristic function on the universe, determining a subset $\{X \in \mathcal{U} : [\![\alpha, .q]\!](X) = 1\}$ consisting of those sets in the universe for which the property q holds. The most general way of ensuring the consistency of introducing a new atomic type ν satisfying $q[\nu/\alpha]$ would be to prove '$\exists\alpha.\ q$'. However, such a formula with quantification over types is not[8] a part of the HOL logic and one must proceed indirectly—replacing the formula by (a logically weaker) one that can be expressed formally with HOL syntax. The formula used is

$$(\exists f_{\alpha\to\sigma}.\ \mathsf{Type_Definition}\ p\ f)\ \Rightarrow\ q$$

where σ is a type, $p_{\sigma\to bool}$ is a closed term and neither involve the type variable α. This formula says 'q holds of any type which is in bijection with the subtype of σ determined by p'. If this formula is provable and if the subtype is non-empty, i.e. if

$$\exists x_\sigma.\ p\ x$$

is provable, then it is consistent to introduce an extension with a new atomic type ν satisfying $q[\nu/\alpha]$.

In giving the formal definition of this extension mechanism, two refinements will be made. Firstly, σ is allowed to be polymorphic and hence a new type constant of appropriate arity is introduced, rather than just an atomic type. Secondly, the above existential formulas are permitted to be proved (in the theory to be extended) from some hypotheses.[9] Thus a *type specification* for a theory $\mathcal{T}$ is given by

Data

$$\langle(\alpha_1, \ldots, \alpha_n)op, \sigma, p, \alpha, \Gamma, q\rangle$$

Conditions

(i) (op, n) is a type constant that is not in $\mathsf{Struc}_\mathcal{T}$.

(ii) $\alpha_1, \ldots, \alpha_n.\sigma$ is a type-in-context with $\sigma \in \mathsf{Types}_\mathcal{T}$.

(iii) $p_{\sigma\to bool}$ is a closed term in $\mathsf{Terms}_\mathcal{T}$ whose type variables occur in $\alpha s = \alpha_1, \ldots, \alpha_n$.

(iv) α is a type variable distinct from those in αs.

(v) Γ is a list of closed formulas in $\mathsf{Terms}_\mathcal{T}$ not involving the type variable α.

(vi) q is a closed formula in $\mathsf{Terms}_\mathcal{T}$.

[8] yet!

[9] This refinement increases the applicability of the extension mechanism without increasing its expressive power. A similar refinement could have be made to the other theory extension mechanisms.

(vii) The sequents

$$(\Gamma \quad , \quad \exists x_\sigma.\ p\ x)$$
$$(\Gamma \quad , \quad (\exists f_{\alpha\to\sigma}.\ \mathsf{Type_Definition}\ p\ f)\ \Rightarrow\ q)$$

are in $\mathsf{Theorems}_{\mathcal{T}}$.

The extension of a standard theory $\mathcal{T}$ by such a type specification is denoted

$$\mathcal{T}+_{tyspec}\langle(\alpha_1,\ldots,\alpha_n)op,\sigma,p,\alpha,\Gamma,q\rangle$$

and is defined to be the theory

$$\begin{array}{l}\langle \mathsf{Struc}_{\mathcal{T}} \cup \{(op,n)\},\\ \ \mathsf{Sig}_{\mathcal{T}},\\ \ \mathsf{Axioms}_{\mathcal{T}} \cup \{(\Gamma, q[(\alpha_1,\ldots,\alpha_n)op/\alpha])\},\\ \ \mathsf{Theorems}_{\mathcal{T}}\rangle\end{array}$$

Example To carry out the extension of `INIT` mentioned at the start of this section, one forms

$$\texttt{INIT}+_{tyspec}\langle()one, bool, p, \alpha, \emptyset, q\rangle$$

where p is the term $\lambda b_{bool}.\ b$ and q is the formula $\forall f_{\beta\to\alpha}\ g_{\beta\to\alpha}.\ f = g$. Thus the result is a theory extending `INIT` with a new type constant *one* satisfying the axiom $\forall f_{\beta\to one}\ g_{\beta\to one}.\ f = g$.

To verify that this is a correct application of the extension mechanism, one has to check Conditions (i) to (vii) above. Only the last one is non-trivial: it imposes the obligation of proving two sequents from the axioms of `INIT`. The first sequent says that p defines an inhabited subset of *bool*, which is certainly the case since T witnesses this fact. The second sequent says in effect that any type α that is in bijection with the subset of *bool* defined by p has the property that there is at most one function to it from any given type β; the proof of this from the axioms of `INIT` is left as an exercise.

Proposition *The theory* $\mathcal{T}+_{tyspec}\langle(\alpha_1,\ldots,\alpha_n)op,\sigma,p,\alpha,\Gamma,q\rangle$ *has a standard model if the theory* $\mathcal{T}$ *does.*

Proof Write αs for $\alpha_1,\ldots,\alpha_n$, and suppose that $\alpha s' = \alpha'_1,\ldots,\alpha'_m$ is the list of type variables occurring in Γ and q, but not already in the list $\alpha s, \alpha$.

Suppose M is a standard model of $\mathcal{T}$. Since $\alpha s, .p$ is a term-in-context of type $\sigma\to bool$, interpreting it in M yields

$$[\![\alpha s, .p]\!]_M \in \prod_{Xs\in\mathcal{U}^n} [\![\alpha s.\sigma\to bool]\!]_M(Xs) = \prod_{Xs\in\mathcal{U}^n} [\![\alpha s.\sigma]\!]_M(Xs)\to 2.$$

There is no loss of generality in assuming that Γ consists of a single formula γ. (Just replace Γ by the conjunction of the formulas it contains, with the convention that this

conjunction is T if Γ is empty.) By assumption on $\alpha s'$ and by Condition (iv), $\alpha s, \alpha s', .\gamma$ is a term-in-context. Interpreting it in M yields

$$[\![\alpha s, \alpha s'.\gamma]\!]_M \in \prod_{(Xs,Xs')\in\mathcal{U}^{n+m}} [\![\alpha s, \alpha s'.bool]\!]_M(Xs, Xs') = \mathcal{U}^{n+m}{\to}2$$

Now $(\gamma, \exists x_\sigma.\ p\ x)$ is in $\mathsf{Theorems}_T$ and hence by the Soundness Theorem 10.3.2 this sequent is satisfied by M. Using the semantics of $\exists$ given in Section 10.4.2 and the definition of satisfaction of a sequent from Section 10.2, this means that for all $(Xs, Xs') \in \mathcal{U}^{n+m}$ if $[\![\alpha s, \alpha s'.\gamma]\!]_M(Xs, Xs') = 1$, then the set

$$\{y \in [\![\alpha s.\sigma]\!]_M \ :\ [\![\alpha s, .p]\!](Xs)(y) = 1\}$$

is non-empty. (This uses the fact that p does not involve the type variables $\alpha s'$, so that by Lemma 4 in Section 9.3.3 $[\![\alpha s, \alpha s'.p]\!]_M(Xs, Xs') = [\![\alpha s, .p]\!]_M(Xs)$.) Since it is also a subset of a set in $\mathcal{U}$, it follows by property **Sub** of the universe that this set is an element of $\mathcal{U}$. So defining

$$S(Xs) = \begin{cases} \{y \in [\![\alpha s.\sigma]\!]_M \ :\ [\![\alpha s, .p]\!](Xs)(y) = 1\} & \text{if } [\![\alpha s, .\gamma]\!]_M(Xs, Xs') = 1, \text{ some } Xs' \\ 1 & \text{otherwise} \end{cases}$$

one has that S is a function $\mathcal{U}^n{\to}\mathcal{U}$. Extend M to a model of the signature of $\mathcal{T}'$ by defining its value at the new n-ary type constant op to be this function S. Note that the values of σ, p, γ and q in M' are the same as in M, since these expressions do not involve the new type constant op.

For each $Xs \in \mathcal{U}^n$ define i_{Xs} to be the inclusion function for the subset $S(Xs) \subseteq [\![\alpha s.\sigma]\!]_M$ if $[\![\alpha s, \alpha s'.\gamma]\!]_M(Xs, Xs') = 1$ for some Xs', and otherwise to be the function $1{\to}[\![\alpha s.\sigma]\!]_M$ sending $0 \in 1$ to $\mathrm{ch}([\![\alpha s.\sigma]\!]_M)$. Then $i_{Xs} \in (S(Xs){\to}[\![\alpha s.\sigma]\!]_{M'}(Xs))$ because $[\![\alpha s.\sigma]\!]_{M'} = [\![\alpha s.\sigma]\!]_M$. Using the semantics of Type_Definition given in Section 10.4.2, one has that for any $(Xs, Xs') \in \mathcal{U}^{n+m}$, if $[\![\alpha s, \alpha s'.\gamma]\!]_{M'}(Xs, Xs') = 1$ then

$$[\![\mathsf{Type_Definition}]\!]_{M'}([\![\alpha s.\sigma]\!]_{M'}, S(Xs))([\![\alpha s, .p]\!]_{M'})(i_{Xs}) = 1.$$

Thus M' satisfies the sequent

$$(\gamma\ ,\ \exists f_{(\alpha s)op\to\sigma}.\ \mathsf{Type_Definition}\ p\ f).$$

But since the sequent $(\gamma, (\exists f_{\alpha\to\sigma}.\ \mathsf{Type_Definition}\ p\ f)\ \Rightarrow\ q)$ is in $\mathsf{Theorems}_T$, it is satisfied by the model M and hence also by the model M' (since the sequent does not involve the new type constant op). Instantiating α to $(\alpha s)op$ in this sequent (which is permissible since by Condition (iv) α does not occur in γ), one thus has that M' satisfies the sequent

$$(\gamma\ ,\ (\exists f_{(\alpha s)op\to\sigma}.\ \mathsf{Type_Definition}\ p\ f) \Rightarrow q[(\alpha s)op/\alpha]).$$

Applying Modus Ponens, one concludes that M' satisfies $(\gamma\ ,\ q[(\alpha s)op/\alpha])$ and therefore M' is a model of $\mathcal{T}'$, as required.

An extension by type definition is in fact a special case of extension by type specification. To see this, suppose $\langle(\alpha_1,\ldots,\alpha_n)op,\ \sigma,\ p_{\sigma\to bool}\rangle$ is a type definition for a theory $\mathcal{T}$. Choosing a type variable α different from $\alpha_1,\ldots,\alpha_n$, let q denote the formula

$$\exists f_{\alpha\to\sigma}.\ \mathsf{Type_Definition}\ p\ f$$

Then $\langle(\alpha_1,\ldots,\alpha_n)op,\sigma,p,\alpha,\emptyset,q\rangle$ satisfies all the conditions necessary to be a type specification for $\mathcal{T}$. Since $q[(\alpha_1,\ldots,\alpha_n)op/\alpha]$ is just $\exists f_{(\alpha_1,\ldots,\alpha_n)op\to\sigma}.\ \mathsf{Type_Definition}\ p\ f$, one has that

$$\mathcal{T}+_{tydef}\langle(\alpha_1,\ldots,\alpha_n)op,\sigma,p\rangle = \mathcal{T}+_{tyspec}\langle(\alpha_1,\ldots,\alpha_n)op,\sigma,p,\alpha,\emptyset,q\rangle$$

Thus the Proposition in Section 10.5.4 is a special case of the above Proposition.

In an extension by type specification, the property q which is asserted of the newly introduced type constant need not determine the type constant uniquely (even up to bijection). Correspondingly there may be many different standard models of the extended theory whose restriction to $\mathcal{T}$ is a given model M. By contrast, a type definition determines the new type constant uniquely up to bijection, and any two models of the extended theory which restrict to the same model of the original theory will be isomorphic.

Part III

The HOL System

Chapter 11

The HOL Logic in ML

In this chapter, the concrete representation of the HOL logic is described. This involves describing the ML functions that comprise the interface to the logic (up to and including Section 11.3); the quotation, printing and parsing of logical terms (Section 11.4); the representation of theorems (Section 11.7); the representation of theories (Section 11.8); the basic HOL theories that are built into the HOL system (Sections 11.8.4 and 11.8.5); the methods for extending theories (throughout Section 11.9 and in Section 11.10); and the ML system functions concerning the logic (Section 11.11.1). It is assumed that the reader is familiar with ML. If not, the introduction to ML in *Getting Started with HOL* in *TUTORIAL* should be read first.

The HOL system provides ML types `type` and `term` to represent types and terms of the HOL logic, as defined in Sections 9.2 and 9.3, respectively.[1] It also provides primitive ML functions for creating and manipulating values of these types. The key idea of the HOL system, due to Robin Milner, and discussed in this chapter, is that theorems are represented as an abstract ML type whose only pre-defined values are axioms, and whose only operations are rules of inference. This means that the only way to construct theorems in HOL is to apply rules of inference to axioms or existing theorems; hence the consistency of the logic is preserved.

The purpose of the meta-language ML is to provide a programming environment in which to build theorem proving tools to assist in the construction of proofs. When the HOL system is built, a range of useful theorems are pre-proved and a set of tools pre-defined. The basic system thus offers a rich initial environment; users can further enrich it by implementing their own application specific tools and building their own application specific theories.

11.1 Lexical matters

The name of a HOL variable can be any ML string, but the quotation mechanism will parse only names that are identifiers (see Section 11.1.1 below). The use of non-identifiers as variable names is discouraged except in special circumstances (for example, when writing derived rules that generate variables with names that are guaranteed to be different from

[1] Care must be taken to avoid confusion between ML types and types of the HOL logic: the ML type `type` represents types of the HOL logic in ML.

existing names). The names of type variables in the HOL logic are strings of *s optionally followed by a number or an identifier (see Section 11.1.2 for examples). The name of a type constant or a term constant of the HOL logic can be any identifier. ML identifiers are described in Part I; the lexical structure of HOL identifiers is identical, however the description is repeated here for convenience.

11.1.1 Identifiers

The structure of HOL identifiers is partly programmable. The default is that an identifier can be of two forms:

(i) A sequence of alphanumerics starting with a letter, where the default structure of an alphanumeric is that it is a letter, a digit, a prime (') or an underbar (_).

(ii) A special symbol chosen from the following list

```
**    ++    <--   <->   -->   ---   ><    >>
>=    <==   <=>   ===   ==>   \/    //    /\
!?    !!    !\    ?!    ??    ?\    :=    <>
<-    <<    <=    --    ->    =>    ==
```

A letter is a member of the list:

```
a b c d e f g h i j k l m n o p q r s t u v w x y z
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
```

HOL is case-sensitive: upper and lower case letters are considered to be different.

The function

```
new_letter : string -> void
```

makes a new character behave like a letter. For example:

```
new_letter `+`;;
```

will make `+int` and `foo+bar` become allowable identifiers. Failure occurs if the argument string is longer than one character.

The function

```
is_letter : string -> bool
```

tests if a unit string is a letter; it fails if the string has more than one character.

Alphanumerics are, by default, letters or digits. A digit is one of `0`, `1`, `2`, `3`, `4`, `5`, `6`, `7`, `8`, or `9`. A *number* is a string of one or more digits. The function

```
new_alphanum : string -> void
```

makes a new character behave like an alphanumeric. For example:

```
new_alphanum `+`;;
```

will make `foo+bar` and `foo+` become allowable names (but not `+bar`). Failure occurs if the argument string is longer than one character.

The function

```
is_alphanum : string -> bool
```

tests if a unit string is an alphanumeric; it fails if the string has more than one character.

It is a consequence of the way lexical analysis is implemented that any initial subsequence of a special symbol is also a special symbol (e.g. since `==>` is a special symbol, so are `==` and `=`). The function

```
new_special_symbol : string -> bool
```

makes the argument string, and all its substrings, special symbols; it fails if

(i) the argument string is a single character, or

(ii) it starts with a letter or an alphanumeric.

For example:

```
new_special_symbol `===>>>`;;
```

makes `===>>>`, `===>>`, `===>` and `===` into new special symbols (`==` is already a special symbol because `==>` is).

The function

```
special_symbols : void -> string list
```

gives the list of currently declared special symbols .

11.1.1.1 Separators

The separators used by the HOL lexical analyser are (with ascii codes in brackets):

space (32), carriage return (13), line feed (10), tab (^I, 9), form feed (^L, 12)

11.1.2 Type variable names

The names of type variables in the HOL logic are strings of *s optionally followed by a number or by an identifier; for example:

```
*   **   ***   *'   **'   **25   ***thing
```

11.2 Types

The allowed types depend on which type constants have been declared in the current theory. See Section 11.8 for details of how such declarations are made.

There are two primitive constructor functions for values of type `type`:

```
mk_vartype : string -> type
mk_type    : (string # type list) -> type
```

The function `mk_vartype` constructs a type variable with a given name; it fails if the name is not an allowable type variable name (i.e. not a string of *s followed by a number or identifier).

The function `mk_type` constructs a compound type from a string representing the name of the type operator and a list of types representing the arguments to the operator. Function types $\sigma_1 \rightarrow \sigma_2$ of the logic are represented in ML as though they were compound types (σ_1, σ_2)`fun` (in Section 9.2, however, function types were not regarded as compound types).

The evaluation of `mk_type(`‘*name*‘`, [`σ_1`;`$\cdots$`;`σ_n`])` fails if

(i) *name* is not an identifier;

(ii) *name* is not a type operator of the current theory;

(iii) *name* is a type operator of the current theory, but its arity is not n.

For example, `mk_type(‘bool‘,[])` evaluates to an ML value of type term representing the type *bool* and `mk_type(‘fun‘, [mk_type(‘ind‘,[]); mk_type(‘bool‘,[])])` evaluates to a value representing *ind*$\rightarrow$*bool*. (These types are introduced in Section 11.8.4.1).

There are two primitive destructor functions for values of type `type`:

```
dest_vartype : type -> string
dest_type    : type -> (string # type list)
```

The function `dest_vartype` extracts the name of a type variable. The function `dest_type` destructs a compound type into the name of the type operator and a list of the argument types; `dest_vartype` and `dest_type` are thus the inverses of `mk_vartype` and `mk_type`, respectively. The destructors fail on arguments of the wrong form.

Types are printed in the form `":` $\cdots$ `"` using the quotation syntax described in Section 11.4. For example, the ML value of type `type` representing *ind*$\rightarrow$(*ind*$\rightarrow$*bool*) would be printed as `":ind -> ind -> bool"`.

11.3 Terms

The four primitive kinds of terms of the logic are described in Section 9.3. The ML functions for manipulating these are described in this section. There are also various derived terms that are described in Section 11.4.3.

The allowed terms depend on which constants have been declared in the current theory. See Section 11.8 for details of how such declarations are made.

There are four primitive constructor functions for values of type `term`:

```
mk_var : (string # type) -> term
```

`mk_var`(x,σ) evaluates to a variable with name x and type σ; it always succeeds.

```
mk_const : (string # type) -> term
```

`mk_const`(c,σ) evaluates to a term representing the constant with name c and type σ; it fails if:

(i) c is not an allowable constant name;

(ii) c is not the name of a constant in the current theory;

(iii) σ is not an instance of the generic type of c (the generic type of a constant is established when the constant is defined; see Section 11.8).

```
mk_comb : (term # term) -> term
```

`mk_comb`(t_1,t_2) evaluates to a term representing the combination t_1 t_2. It fails if:

(i) the type of t_1 does not have the form σ'`->`σ;

(ii) the type of t_1 has the form σ'`->`σ, but the type of t_2 is not equal to σ'.

```
mk_abs : (term # term) -> term
```

`mk_abs`(x,t) evaluates to a term representing the abstraction $\lambda x.\ t$; it fails if x is not a variable.

There are four primitive destructor functions on terms:

```
dest_var   : term -> (string # type)
dest_const : term -> (string # type)
dest_comb  : term -> (term # term)
dest_abs   : term -> (term # term)
```

These are the inverses of `mk_var`, `mk_const`, `mk_comb` and `mk_abs`, respectively. They fail when applied to terms of the wrong form. Other useful destructor functions are `rator`, `rand`, `bndvar`, `body`, `lhs` and `rhs`. See *REFERENCE* for details.

The function

```
type_of : term -> type
```

returns the type of a term. It could be defined (recursively) in terms of the destructors but is predefined for convenience.

Terms are printed in the form " ··· " using the quotation syntax described in Section 11.4. For example, the term representing

$$\forall x\ y.\ x < y \Rightarrow \exists z.\ x + z = y$$

would be printed as:

```
"!x y.  x < y ==> ?z.  x + z = y"
```

Note that a colon is used to distinguish type quotation from term quotation; the former have the form ": ··· " and the latter have the form " ··· ".

11.4 Quotation

HOL types and terms can be input to the system in two ways: by using constructor functions, or by using *quotation*. The former allows some terms to be built which cannot be constructed using quotation. For example, a term containing two variables with the same name but different types, e.g. the term $x_{bool} = (x_{num} = 1)$, can be built only by using constructors.

It would be tedious, however, to always have to input types and terms using the constructor functions. The HOL system, following LCF, has a special quotation parser and type checker that enables terms to be input using a fairly standard syntax. The HOL printer also outputs types and terms using this syntax.

For example, the ML expression `":bool->bool"` denotes exactly the same value (of ML type `type`) as

```
mk_type(`fun`,[mk_type(`bool`,[]);mk_type(`bool`,[])])
```

and `"\x.x+1"` can be used instead of

```
mk_abs
 (mk_var(`x`,mk_type(`num`,[])),
  mk_comb
  (mk_comb
   (mk_const
    (`+`,
     mk_type(`fun`,[mk_type(`num`,[]);
                    mk_type(`fun`,[mk_type(`num`,[]);
                                   mk_type(`num`,[])])])),
    mk_var(`x`, mk_type(`num`,[]))),
   mk_const(`1`, mk_type(`num`,[]))))
```

It should be noted that there is no explicit type information in `"\x.x+1"`. The HOL type checker knows that `1` has type `num` and `+` has type `num->(num->num)`. From this information it can infer that both occurrences of `x` in `"\x.x+1"` could have type `num`. This is not the only possible type assignment; for example, the first occurrence of `x` could have type `bool` and the second one have type `num`. In that case there would be two *different* variables with name `x`, namely $\mathtt{x}_{\mathtt{bool}}$ and $\mathtt{x}_{\mathtt{num}}$, the second of which is free. In fact, as mentioned, the only way to construct a term with this second type assignment is by using constructors, since the type checker uses the heuristic that all variables in a term with the same name have the same type. This is illustrated in the following session.

1

```
#"x = (x = 1)";;
Badly typed application of:  "$= x"
   which has type:           ":num -> bool"
to the argument term:        "x = 1"
   which has type:           ":bool"

evaluation failed     mk_comb in quotation

#mk_eq
# (mk_var(`x`,mk_type(`bool`,[])),
#  mk_eq
#   (mk_var(`x`,mk_type(`num`,[])),
#    mk_const(`1`,mk_type(`num`,[]))));;
"x = (x = 1)" : term
```

The quotation type checker was designed and implemented by Robin Milner. It employs heuristics like the one above to infer a sensible type for all variables occurring in a term. If there are not enough clues, then the system will complain with an error message.

To give the system a hint, types can be explicitly indicated by following any subterm with a colon and then a type. For example, `"f(x:num):bool"` will type check with `f` and `x` getting types `num->bool` and `num` respectively. If there are polymorphic constants in a term, there must be enough type information to uniquely identify a type instance for

each such constant. There is also a feature called *sticky types* that enables variables to be given default types; this is described in Section 11.11.2.

The type checking algorithm used for the HOL logic differs from that used for ML. For example, the ML expression `\x.x` will get ML type `*->*`, but the HOL term `"\x.x"` will fail to type check, as shown in the session below.

```
#"\x. x";;                                                              1
Indeterminate types:  "x:?"

evaluation failed     types indeterminate in quotation
```

To get the term `\x.x` to type check, the type of the variable `x` must be given explicitly, for example by writing `"\x:*.x"`.

```
#"\x:*. x";;                                                            2
"\x. x" : term
```

This treatment of types within quotations is inherited from LCF.

11.4.1 Type quotation

The table below shows ML expressions for various kinds of type quotations. The expressions in the same row are equivalent.

Types

Kind of type	ML *quotation*	*Constructor expression*
Type variable	`":*...."`	`mk_vartype(`‘`*...`‘`)`
Type constant	`":`*op*`"`	`mk_type(`‘*op*‘`,[])`
Function type	`":`σ_1`->`σ_2`"`	`mk_type(‘fun‘, [":`σ_1`";":`σ_2`"])`
Compound type	`":(`σ_1`, ... , `σ_n`)`*op*`"`	`mk_type(`‘*op*‘`, [":`σ_1`"; ... ;":`σ_n`"])`

11.4.2 Term quotation

Equivalent ways of inputting the four primitive kinds of term are shown in the next table.

Primitive terms		
Kind of term	ML *quotation*	*Constructor expression*
Variable	"var:σ"	mk_var(‘var‘,":σ")
Constant	"$const$:σ"	mk_const(‘$const$‘,":σ")
Combination	"t_1 t_2"	mk_comb("t_1","t_2")
Abstraction	"\x.t"	mk_abs("x","t")

11.4.3 Special syntactic forms

The HOL quotation parser can translate various standard logical notations into primitive terms. For example, if + has been declared an infix (as explained in Section 11.8), then "x+1" is translated to "$+ x 1". The escape character $ suppresses the infix behaviour of + and prevents the quotation parser getting confused. In general, $ can be used to suppress any special syntactic behaviour a constant name might have. This is illustrated in the table below, in which the terms in the column headed 'ML *quotation*' are translated by the quotation parser to the corresponding terms in the column headed '*Primitive term*'. Conversely, the terms in the latter column are always printed in the form shown in the former one. The ML constructor expressions in the rightmost column evaluate to the same values (of type term) as the other quotations in the same row.

Non-primitive terms			
Kind of term	ML *quotation*	*Primitive term*	*Constructor expression*
Negation	"~t"	"$~ t"	mk_neg("t")
Disjunction	"t_1\/t_2"	"$\/ t_1 t_2"	mk_disj("t_1","t_2")
Conjunction	"t_1/\t_2"	"$/\ t_1 t_2"	mk_conj("t_1","t_2")
Implication	"t_1==>t_2"	"$==> t_1 t_2"	mk_imp("t_1","t_2")
Equality	"t_1=t_2"	"$= t_1 t_2"	mk_eq("t_1","t_2")
$\forall$-quantification	"!x.t"	"$!(\$x$.$t$)"	mk_forall("x","t")
$\exists$-quantification	"?x.t"	"$?(\$x$.$t$)"	mk_exists("x","t")
ε-term	"@x.t"	"$@(\$x$.$t$)"	mk_select("x","t")
Conditional	"(t=>t_1|t_2)"	"COND t t_1 t_2"	mk_cond("t","t_1","t_2")
let-expression	"let x=t_1 in t_2"	"LET(\x.t_2)t_1"	mk_let("\x.t_1","t_2")

There are constructors, destructors and indicators for all the obvious constructs. (Indicators, e.g. is_neg, return truth values indicating whether or not a term belongs to the

syntax class in question.) In addition to the constructors listed in the table there are constructors for pairs and lists, namely `mk_pair`, `mk_cons` and `mk_list` (see *REFERENCE*). The constants `COND` and `LET` are explained in Sections 11.5.1 and 11.5.3, respectively. The constants `\/`, `/\`, `==>` and `=` are examples of *infixes* and represent $\vee$, $\wedge$, $\Rightarrow$ and equality, respectively. If c is declared to be an infix, then the HOL parser will translate "$t_1\ c\ t_2$" to "$\$c\ t_1\ t_2$".

The constants `!`, `?` and `@` are examples of *binders* and represent $\forall$, $\exists$ and ε, respectively. If c is declared to be a binder, then the HOL parser will translate "$c\ x.t$" to the combination "$\$c(\backslash x.t)$" (i.e. the application of the constant c to the representation of the abstraction $\lambda x.\ t$).

In addition to the kinds of terms in the tables above, the parser also supports the following syntactic abbreviations.

Syntactic abbreviations

Abbreviated term	*Meaning*	*Constructor expression*
"$t\ t_1 \cdots t_n$"	"$(\cdots(t\ t_1)\cdots t_n)$"	`list_mk_comb("`t`",["`t_1`"; ... ;"`t_n`"])`
"$\backslash x_1 \cdots x_n.t$"	"$\backslash x_1.\ \cdots\ \backslash x_n.t$"	`list_mk_abs(["`x_1`"; ... ;"`x_n`"],"`t`")`
"$!x_1 \cdots x_n.t$"	"$!x_1.\ \cdots\ !x_n.t$"	`list_mk_forall(["`x_1`"; ... ;"`x_n`"],"`t`")`
"$?x_1 \cdots x_n.t$"	"$?x_1.\ \cdots\ ?x_n.t$"	`list_mk_exists(["`x_1`"; ... ;"`x_n`"],"`t`")`

There are also constructors `list_mk_conj`, `list_mk_disj`, `list_mk_imp` and `list_mk_pair` for conjunctions, disjunctions, implications and tuples respectively. The corresponding destructor functions are called `strip_comb`, etc., except that there is no `strip_conj` or `strip_disj`. Instead there are functions called `conjuncts` and `disjuncts`, but it should be noted that these are not proper inverses of `list_mk_conj` and `list_mk_disj`.

11.5 Syntax for restricted quantification

Syntactic support for restricted quantification and abstraction is provided. This follows a suggestion discussed at the Second HOL Users Meeting and implements a method of simulating subtypes and dependent types with predicates.

Currently no derived rules are provided to support this notation, so any inferences will need to work on the underlying semantic representation.

The new syntax automatically translates as follows:

```
\v::P.B    <---->   RES_ABSTRACT P (\v.B)
!v::P.B    <---->   RES_FORALL   P (\v.B)
?v::P.B    <---->   RES_EXISTS   P (\v.B)
@v::P.B    <---->   RES_SELECT   P (\v.B)
```

Anything can be written between the binder and '::' that can be written between the binder and '.'. See the examples below.

The flag `print_restrict` has default `true`, but if set to `false` will disable the pretty printing. This is useful for seeing what the semantics of particular restricted abstractions are.

The constants `RES_ABSTRACT`, `RES_FORALL`, `RES_EXISTS` and `RES_SELECT` are defined in the theory `bool` by:

```
RES_ABSTRACT P B  =  \x:*. (P x => B x | ARB:**)

RES_FORALL   P B  =  !x:*. P x ==> B x

RES_EXISTS   P B  =  ?x:*. P x /\ B x

RES_SELECT   P B  =  @x:*. P x /\ B x
```

where the constant `ARB`, which is also defined in the theory `bool`, has the definition:

```
ARB  =  @x:*. T
```

User-defined binders can also have restricted forms, which are set up with the function:

```
associate_restriction : (string # string) -> *
```

If B is the name of a binder and `RES_`B is the name of a suitable constant (which must be explicitly defined), then executing:

```
associate_restriction('B', 'RES_B')
```

will cause the parser and pretty-printer to support:

$B\ v$`::`P`.`$\ B$ `<---->` `RES_`$B\ P$ `(\`v`.`$\ B$`)`

Note that associations between user defined binders and their restrictions are not stored in theory files, so they have to be set up for each HOL session (e.g. with a `hol-init.ml` initialization file – see Section 8.6).

Here is an example session:

```
#"!x y::P. x<y";;                                                        1
"!x y :: P. x < y" : term

#set_flag(`print_restrict`, false);;
true : bool

#"!x y::P. x<y";;
"RES_FORALL P(\x. RES_FORALL P(\y. x < y))" : term

#"?(x,y) p::(\(m,n).m<n). p=(x,y)";;
"RES_EXISTS
 (\(m,n). m < n)
 (\(x,y). RES_EXISTS(\(m,n). m < n)(\p. p = x,y))"
: term

#"\x y z::P.[0;x;y;z]";;
"RES_ABSTRACT P(\x. RES_ABSTRACT P(\y. RES_ABSTRACT P(\z. [0;x;y;z])))"
: term
```

A conversion that rewrites away the constants `RES_ABSTRACT`, `RES_FORALL`, `RES_EXISTS` and `RES_SELECT` is:

```
let RESTRICT_CONV =
 DEPTH_CONV
  (REWRITE_CONV (definition `bool` `RES_ABSTRACT`) ORELSEC
   REWRITE_CONV (definition `bool` `RES_FORALL`)   ORELSEC
   REWRITE_CONV (definition `bool` `RES_EXISTS`)   ORELSEC
   REWRITE_CONV (definition `bool` `RES_SELECT`))
 THENC DEPTH_CONV BETA_CONV
```

This is a bit unsatisfactory, as is shown by the artificial example below:

```
#let t = "!x y::P.?f:num->num::Q. f(@n::R.T) = (x+y)";;                  1
t = "!x y :: P. ?f :: Q. f(@n :: R. T) = x + y" : term

#RESTRICT_CONV t;;
|- (!x y :: P. ?f :: Q. f(@n :: R. T) = x + y) =
   (!x. P x ==> (!x'. P x' ==> (?x. Q x /\ (x(@x. R x /\ T) = x + x'))))
```

The variable x in the definitions of the constants `RES_ABSTRACT`, `RES_FORALL`, `RES_EXISTS` and `RES_SELECT` gets confused with the variable in the supplied term. This can be avoided by changing `RESTRICT_CONV` to perform explicit alpha-conversion. For example, by implementing a conversion:

```
RES_FORALL P (\v.B[v]) ----> !v.  P v ==> B[v]
```

Dealing with the case when v is a variable structure is also desirable. For example:

```
#let t1 = "!(m,n)::P. m<n";;                                              2
t1 = "!(m,n) :: P. m < n" : term

#RESTRICT_CONV t1;;
|- (!(m,n) :: P. m < n) = (!x. P x ==> (\(m,n). m < n)x)
```

If anyone writes the desired conversions please let us know!

Here is an example of a user-defined restriction:

```
#new_binder_definition(`DURING`, "DURING(p:num#num->bool) = $!p");;       3
|- !p. $DURING p = $! p

#"DURING x::(m,n). p x";;
no restriction constant associated with DURING
skipping: x " ;; parse failed

#new_definition
# (`RES_DURING`, "RES_DURING(m,n)p = !x. m<=x /\ x<=n ==> p x");;
|- !m n p. RES_DURING(m,n)p = (!x. m <= x /\ x <= n ==> p x)

#associate_restriction(`DURING`,`RES_DURING`);;
() : void

#"DURING x::(m,n). p x";;
"DURING x :: (m,n). p x" : term

#set_flag(`print_restrict`,false);;
true : bool

#"DURING x::(m,n). p x";;
"RES_DURING(m,n)(\x. p x)" : term
```

11.5.1 Conditionals

A conditional term "t_1 `=>` t_2 `|` t_3" means 'if t_1 then t_2 else t_3' and abbreviates the application "`COND` t_1 t_2 t_3", where `COND` is a predefined constant of the theory `bool` (see Section 11.8.4.1). An iterated conditional

$$t_{11}\ \texttt{=>}\ t_{12}\ \texttt{|}\ t_{21}\ \texttt{=>}\ t_{22}\ \texttt{|}\ \ \ldots\ \texttt{|}\ t_{n1}\ \texttt{=>}\ t_{n2}\ \texttt{|}\ t_{n3}$$

translates to:

$$t_{11}\ \texttt{=>}\ t_{12}\ \texttt{|}\ (t_{21}\ \texttt{=>}\ t_{22}\ \texttt{|}\ \ \ldots\ \texttt{|}\ (t_{n1}\ \texttt{=>}\ t_{n2}\ \texttt{|}\ t_{n3})\ \ldots\ )$$

which, in turn, abbreviates:

$$\texttt{COND}\ t_{11}\ t_{12}\ (\texttt{COND}\ t_{21}\ t_{22}\ \ldots\ (\texttt{COND}\ t_{n1}\ t_{n2}\ t_{n3})\ \ldots\ )$$

11.5.2 Paired abstractions

The quotation parser will convert "\(x_1,x_2).t" to "UNCURRY(\x_1 x_2.t)", where the constant UNCURRY is defined by:

```
UNCURRY f (x,y)  =  f x y
```

See Section 11.9.2.1 for more details and an explanation of pair terms (t_1,t_2). The transformation is done recursively so that, for example,

"\(x_1,x_2,x_3).t"

is converted to

"UNCURRY \$x_1$.UNCURRY(\$x_2$,$x_3$.$t$))"

More generally, the quotation parser repeatedly applies the transformation:

"\(v_1,v_2).t" $\leadsto$ "UNCURRY(\$v_1$.\$v_2$.$t$)"

until no more variable structures remain. For example:

"\(x,y).t"	$\leadsto$	"UNCURRY(\x y.t)"
"\(x_1,x_2,...,x_n).t"	$\leadsto$	"UNCURRY(\x_1.\(x_2,...,x_n).t)"
"\((x_1,...,x_n),y_1,...,y_m).t"	$\leadsto$	"UNCURRY(\(x_1,...,x_n).\(y_1,...,y_m).t)"

The HOL top-level printer inverts these transformations if the flag[2] print_uncurry is true. For example:

```
#"\((a,b),(x,y)). (a+b) < (x*y)";;                                        1
"\((a,b),x,y). (a + b) < (x * y)" : term

#set_flag(`print_uncurry`,false);;
true : bool

#"\((a,b),(x,y)). (a+b) < (x*y)";;
"UNCURRY(UNCURRY(\a b. UNCURRY(\x y. (a + b) < (x * y))))" : term
```

Note that a variable structure like "(x,y)" in "\(x,y).x+y" is not a subterm of the abstraction in which it occurs; it disappears on parsing. This can lead to unexpected errors (accompanied by obscure error messages). For example:

[2] See Section 8.13 for details of ML flags.

```
#"\(x,y).x+y";;                                                    2
"\(x,y). x + y" : term

#let p = "(x:num,y:num)";;
p = "x,y" : term

#"\^p.x+y";;
evaluation failed     mk_abs in quotation
```

Furthermore, if the type checker complains, it may print out diagnostic messages referring to the transformed term:

```
#"\(x,y). x+1";;                                                   3
Indeterminate types:
    "UNCURRY:(num -> (?1 -> num)) -> (num # ?2 -> num)"

evaluation failed     types indeterminate in quotation
```

If b is a binder, then "b(\$x_1$,$x_2$).$t$" is parsed as "\$$b$(\(\$x_1$,$x_2$).$t$)", and hence transformed as above. For example, "!(x,y).x>y" parses to "$!(UNCURRY(\x.\y.$> x y))" (where > is an infixed constant of the theory num meaning 'is greater than').

Applications of paired abstraction to tuples can be β-reduced using PAIRED_BETA_CONV (see Section 13.3.1).

11.5.3 let-terms

The quotation parser accepts let-terms superficially similar to those in ML. For example, the following terms are allowed:

```
"let x = 1 and y = 2 in x+y"

"let f(x,y) = (x*x)+(y*y) and a = 20*20 and b = 50*49 in f(a,b)"
```

As with paired abstractions, let-terms are actually abbreviations for ordinary terms which are specially supported by the parser and pretty printer. The constant LET is defined (in the theory bool) by:

```
LET = (\f x. f x)
```

and is used to encode let-terms in the logic. The parser repeatedly applies the transformations:

"let f v_1 ... v_n = t_1 in t_2" $\leadsto$ "LET(\$f$.$t_2$)(\$v_1$ ... v_n.t_1)"
"let (v_1,...,v_n) = t_1 in t_2" $\leadsto$ "LET(\(v_1,...,v_n).t_2)t_1"
"let v_1=t_1 and ... and v_n=t_n in t" $\leadsto$ "LET(...(LET(LET(\v_1 ... v_n.t)t_1)t_2)...)t_n"

The printer inverts these transformations if the flag print_let is true. For example:

```
#"let x = 1 and y = 2 in x+y";;                                        1
"let x = 1 and y = 2 in x + y" : term

#set_flag(`print_let`,false);;
true : bool

#"let x = 1 and y = 2 in x+y";;
"LET(LET(\x y. x + y)1)2"

#"let (x,y) = (1,2) in x+y";;
"LET(UNCURRY(\x y. x + y))(1,2)" : term
```

Note that, as with uncurried terms, the underlying representation in HOL can manifest itself in error messages. For example:

```
#"let x = y in z";;                                                    1
Indeterminate types:  "LET:(?1 -> ?2) -> (?3 -> ?4)"

evaluation failed     types indeterminate in quotation
```

The reader is recommended to convince himself or herself that the translations of `let`-terms represent the intuitive meaning suggested by the surface syntax.

`let`-terms can be simplified with `let_CONV` – see Section 13.3.4.

11.6 Syntax for sets

The special purpose set-theoretic notations "$\{t_1, t_2, \ldots, t_n\}$" and "$\{t \mid p\}$" are available. The normal interpretation of the former is the finite set containing $t_1, t_2, \ldots, t_n$ and the normal interpretation of the latter is the set of all ts such that p. These interpretations are predefined for the library `sets`, but the user can use the syntax for other purposes if he or she wishes, using the functions:

```
define_finite_set_syntax      : (string # string) -> void
define_set_abstraction_syntax : string -> void
```

Executing:

`define_finite_set_syntax(`c_1`,`c_2`)`

causes "$\{t_1, \ldots, t_n\}$" to parse to:

"$c_1\ t_1\ (c_1\ t_2\ \cdots\ (c_1\ t_n\ c_2)\ \cdots\ ))$"

with failure if either c_1 or c_2 is not the name of a constant.

In the library `sets`, the empty set is `EMPTY` and the infixed function `INSERT` adds an element to a set. Executing:

```
define_finite_set_syntax(‘EMPTY‘,‘INSERT‘)
```

will cause

```
"{1,2,3,4}"
```

to parse to

```
"1 INSERT (2 INSERT (3 INSERT (4 INSERT EMPTY)))"
```

Executing:

```
define_set_abstraction_syntax ‘c‘
```

causes "{t|p}" to parse to:

"c(\(x_1,...,x_n).(t,p))"

where x_1, ..., x_n are the free variables occurring in both t and p. If there are no such free variables then an error results. The order in which the variables are listed in the variable structure of the paired abstraction is an unspecified function of the structure of t (it is approximately left to right). Failure if c is not the name of a constant.

For example, if the library `sets` (i.e. what used to be `all_sets`) is loaded, then

```
define_set_abstraction_syntax ‘GSPEC‘
```

will cause

```
"{x+y | (x < y) /\ (y < z)}"
```

to parse to:

```
"GSPEC(\(x,y). ((x+y), (x < y) /\ (y < z)))"
```

where `GSPEC` is defined by:

```
|- !f. GSPEC f = SPEC(\x. ?y. x,T = f y)
```

and `SPEC` abstracts a predicate to a set (it is the abstraction bijection used in the definition of the type operator `set`). Other examples are:

```
"{x+y+z | (x < y) /\ (y < z)}"
```

will parse to:

```
"GSPEC(\(x,y,z). (x+(y+z), (x < y /\ y < z)))"
```

and

```
"{x+y+w | (x < y) /\ (y < z)}"
```

will parse to:

```
"GSPEC(\(x,y). (x+(y+w), (x < y /\ y < z)))"
```

Note that the precedence of comma is increased in the contexts "`{`$\cdots$`}`" and "`{`$\cdots$`|`". Terms will be printed in set notation if the flag `print_set` is `true`. Note that

"c`(\(`x_1`,...,`x_n`).(`t`,`p`))`"

will only print as "`{`t`|`p`}`" if the variables x_1, ... , x_n occur free in both t and p (and `print_set` is `true`) .

11.6.1 Antiquotation

Within a quotation, expressions of the form `^(`t`)` (where t is an ML expression of type `term` or `type`) are called *antiquotations*. An antiquotation `^(`t`)` evaluates to the ML value of t. For example, `"x \/ ^(mk_conj("y:bool","z:bool"))"` evaluates to the same term as `"x \/ (y /\ z)"`. The most common use of antiquotation is when the term t is just an ML variable x. In this case `^(`x`)` can be abbreviated by `^`x.

The following session illustrates antiquotation.

```
#let y = "x+1";;                                                    1
y = "x + 1" : term

#let z = "y = ^y";;
z = "y = x + 1" : term

#"!x:num.?y:num.^z";;
"!x. ?y. y = x + 1" : term
```

11.6.2 User-programmable quotation typechecker

The typechecker for HOL quotations contains a number of arbitrary design decisions. Several people have suggested changes, e.g. that full Hindley/Milner type inference be performed. Rather than try to create a single new improved typechecker, a facility is now provided that enables the user to write his or her own one and then to install it in the system.

The ML abstract type `preterm` represents the parse trees of HOL terms. A typechecker is a function of type `preterm->term`. If the flag `preterm` is set to `true` (the default is `false`), then HOL will use whatever ML function is currently bound to the name `preterm_handler` as the quotation typechecker. The way this works is that when `preterm` is true the parser

produces a preterm rather than a term, and then wraps a call of `preterm_handler` around the quotation. Other uses of preterms are possible, for example as patterns for describing terms.

The definition of the ML type `preterm` is:

```
rectype preterm =
  preterm_var      of string                    Variables
| preterm_const    of string                    Constants
| preterm_comb     of preterm # preterm         Combinations
| preterm_abs      of preterm # preterm         Abstractions
| preterm_typed    of preterm # type            Explicit typing
| preterm_antiquot of term                      Antiquotation
```

The function:

```
preterm_to_term : preterm -> term
```

invokes the standard HOL typechecker on a preterm and returns the resulting typechecked term (or causes the standard error message).

The following is a rather contrived example:

1
```
#letref p_reg = preterm_var `x`;;
p_reg = preterm_var `x` : preterm

#let preterm_handler p = p_reg:=p;
                         print_string `Typechecking ... `;
                         print_newline();
                         preterm_to_term p;;
preterm_handler = - : (preterm -> term)
```

2
```
#set_flag(`preterm`,true);;
false : bool

#"x+y";;
Typechecking ...
"x + y" : term

#p_reg;;
preterm_comb((preterm_comb((preterm_const `+`), preterm_var `x`)),
             preterm_var `y`)
: preterm
```

Different top-level typechecking can be defined by using a different definition of the function `preterm_handler`. Note that quotations and their typechecking is purely a 'user interface' feature, so changing the typechecker does not compromise the logical soundness of HOL.

11.7 Theorems

In Chapter 9, the notion of deduction was introduced in terms of *sequents*, where a sequent is a pair whose second component is a formula being asserted (a conclusion), and whose first component is a set of formulas (hypotheses). Based on this was the notion of a *deductive system*: a set of pairs, whose second component is a sequent, and whose first component is a sequent list[3]. The concept of a sequent *following from* a set of sequents via a deductive system was then defined: a sequent follows from a set of sequents if the sequent is the last element of some chain of sequents, each of whose elements is either in the set, or itself follows from the set along with earlier elements of the chain, via the deductive system.

A notation for 'follows from' was then introduced. That a sequent $(\{t_1, \ldots, t_n\},\ t)$ follows from a set of sequents Δ, via a deductive system $\mathcal{D}$, is denoted by: $t_1, \ldots, t_n \vdash_{\mathcal{D},\Delta} t$. (It was noted that where either $\mathcal{D}$ or Δ were clear by context, their mention could be omitted; and where the set of hypotheses was empty, its mention could be omitted.)

A sequent that follows from the empty set of sequents via a deductive system is called a *theorem* of that deductive system. That is, a theorem is the last element of a *proof* (in the sense of Chapter 9) from the empty set of sequents. When a pair $(L, (\Gamma, t))$ belongs to a deductive system, and the list L is empty, then the sequent (Γ, t) is called an *axiom*. Any pair $(L, (\Gamma, t))$ belonging to a deductive system is called a *primitive inference* of the system, with hypotheses[4] L and conclusion (Γ, t).

A formula in the abstract is represented concretely in HOL by a term whose HOL type is `":bool"`. Therefore, a term of type `":bool"` is used to represent a member of the set of hypotheses of a sequent; and likewise to represent the conclusion of a sequent. Sets in this context are represented by lists, so the set of hypotheses of a sequent is represented by a list of `":bool"`-typed terms.

A theorem in the abstract is represented concretely in the HOL system by a value with the ML abstract type `thm`. The type `thm` has a primitive destructor function

```
dest_thm : thm -> (term list # term)
```

which returns a pair consisting of the hypothesis list and the conclusion, respectively, of a theorem. From this, two destructor functions are derived

```
hyp   : thm -> term list
concl : thm -> term
```

for extracting the hypothesis list and the conclusion, respectively, of a theorem. The ML type `thm` does not have a primitive constructor function. In this way, the ML type system

[3]Note that these sequents form a list, not a set; that is, are ordered.

[4]Note that 'hypotheses' and 'conclusion' are also used for the components of sequents.

protects the HOL logic from the arbitrary and unrecorded construction of theorems, which would compromise the consistency of the logic. (Functions which return theorems as values, e.g. functions representing primitive inferences, are discussed first in Section 11.8.5, and further in Chapter12.)

It was mentioned in Chapter 9 that the deductive system of HOL includes five axioms[5]. In that Chapter, the axioms were presented in abstract form. The concrete representation of the axioms in HOL is given in Section 11.8.4. To anticipate, the axiom `BOOL_CASES_AX` mentioned in Chapter 9 is printed in HOL as follows (where `"T"` and `"F"` are the HOL logic's constants representing truth and falsity, respectively):

```
|- !t. (t = T) \/ (t = F)
```

Note the special print format, with the approximation to the abstract $\vdash$ notation, `|-`, used to indicate ML type `thm` status; the absence of HOL quotation marks in the `|-` context; and the absence of type information following the printed term. The session below illustrates the use of the destructor functions:

```
#let th = BOOL_CASES_AX;;                                              1
th = |- !t. (t = T) \/ (t = F)

#hyp th;;
[] : term list

#concl th;;
"!t. (t = T) \/ (t = F)" : term

#type_of it;;
":bool" : type
```

In addition to the print conventions mentioned above, the printing of theorems prints hypotheses as periods (i.e. full stops or dots). The function `print_all_thm` prints theorems with hypotheses shown in full. These points are illustrated with a theorem inferred, for example purposes, from another axiom mentioned in Chapter 9: `SELECT_AX`.

```
th = . |- P($@ P)                                                      1

#print_all_thm th;;
P x |- P($@ P)() : void
```

[5]This is a simplification: the axioms are an extension of the basic logic. See Sections 11.8.4.1 and 11.8.4.2.

11.8 Theories

It was mentioned in Chapter 9 that a theory is a 4-tuple

$$\mathcal{T} = \langle \mathsf{Struc}_{\mathcal{T}},\ \mathsf{Sig}_{\mathcal{T}},\ \mathsf{Axioms}_{\mathcal{T}},\ \mathsf{Theorems}_{\mathcal{T}} \rangle$$

where

(i) $\mathsf{Struc}_{\mathcal{T}}$ is the type structure of $\mathcal{T}$;

(ii) $\mathsf{Sig}_{\mathcal{T}}$ is the signature of $\mathcal{T}$;

(iii) $\mathsf{Axioms}_{\mathcal{T}}$ is the set of axioms of $\mathcal{T}$;

(iv) $\mathsf{Theorems}_{\mathcal{T}}$ is the set of theorems of $\mathcal{T}$.

Such a theory is represented in the HOL system as a collection of files, called theory files. Each file has a name of the form *name*`.th`, where *name* is a string supplied by the user. Various additional pieces of information are stored in the theory files, including the parsing status of the constants (i.e. whether they are infixes or binders), as well as several items involving concepts not yet introduced: type abbreviations (see Section 11.8.6 for this); and which axioms are definitions, type definitions or specifications (see Section 11.8.2 for the last three).

Theory files are structured hierarchically to represent sequences of extensions of an initial theory (see Section 10.5) called `HOL`. Each theory file making up a theory records some types, constants, axioms and theorems, together with pointers to other theory files called its *parents*. The theory represented by such a theory file is obtained by taking the union of all the types, constants, axioms and theorems in the file, together with the types, constants, axioms and theorems in all the theory files reachable by following pointers to parents. This collection of reachable files is called the *ancestry* of the theory file. Axioms (including definitions and specifications) and theorems are named in the HOL system by two strings: the name of the theory file in which they are stored, together with a name supplied by the user.

The data stored in a single theory file is called a *theory segment*; this is not really a logical concept, but rather a concept of the representation of theories in the HOL system. It is necessary to distinguish theories from their constituent theory segments because the naming of data in theories is based on the names given to segments. Specifically, axioms, definitions, specifications and theorems are named by a pair of strings $\langle thy, name \rangle$, where thy is the name of the theory segment current when the item was declared and *name* is a specific name supplied by the user (see the functions `new_axiom`, `new_definition`, below). Different items can have the same specific name if the associated segment is different.

A typical piece of work with the HOL system consists in a number of sessions. In the first of these, a new theory, $\mathcal{T}$ say, is created by extending existing theories with a number

of definitions. The concrete result of the session will be a theory file $\mathcal{T}$`.th` whose contents is the segment created during the session and whose ancestry represents the desired logical theory. In subsequent sessions this theory is extended by proving new theorems that will be stored in the file $\mathcal{T}$`.th`. The logical meaning of these sessions is that a new extension to $\mathcal{T}$ is created which replaces the old version. Subsequent pieces of work can build on (i.e. extend) the definitions and theorems of $\mathcal{T}$ by making it a parent of new theories.

There are two modes of working with HOL: *draft mode* and *proof mode*. In the former, the sets $\mathsf{Struc}_{\mathcal{T}}$, $\mathsf{Sig}_{\mathcal{T}}$ and $\mathsf{Axioms}_{\mathcal{T}}$ can be extended; in the latter only $\mathsf{Theorems}_{\mathcal{T}}$ can be changed. In draft mode, inconsistencies can be introduced by asserting inconsistent axioms, but in proof mode only consistency-preserving actions (namely valid proof) can be evaluated. Draft mode is analogous to 'super user mode' in Unix, in that it gives access to dangerous facilities. Everything that can be done in proof mode can be done in draft mode, but not vice versa.

There is always a *current theory* which is the theory represented by the current theory segment together with its ancestry. The name of the current theory is returned by the ML function:

```
current_theory : void -> string
```

Initially HOL is in proof mode with current theory called `HOL`, which is described in Section 11.8.4.

11.8.1 Primitive ML functions for creating theories

The ML functions for creating theories are listed below.

```
new_theory : string -> void
```

`new_theory` '*thy*' can be done in both proof and draft modes. It switches into draft mode for a new theory with name *thy* and it fails if there already exists a file *thy*`.th` in the current search path. The current theory becomes a new parent of *thy*.

```
new_parent : string -> void
```

Executing `new_parent` *thy* makes *thy* into a parent of the current theory. Failure if:

(i) not in draft mode;

(ii) *thy* is not a theory on the current search path;

(iii) there is a type in *thy* with the same name as a type in the current theory;

(iv) there is a constant in *thy* with the same name as a constant in the current theory.

```
new_type : int -> string -> void
```

Executing `new_type` n `‘`op`‘` makes op a new n-ary type operator in the current theory. Failure if:

(i) not in draft mode;

(ii) there already exists a type operator named op in the current theory.

```
new_constant : (string # type) -> void
```

Executing `new_constant(‘`c`‘,`σ`)` makes $c_{\sigma'}$ a new constant of the current theory, for all $c_{\sigma'}$ where σ' is an instance of σ. The type σ is called the *generic type* of c. Failure if:

(i) not in draft mode;

(ii) there already exists a constant named c in the current theory.

```
new_infix : (string # type) -> void
```

Executing `new_infix(‘`ix`‘,`σ`)` declares ix to be a new constant with generic type σ and infix status. Failure if:

(i) not in draft mode;

(ii) there already exists a constant named ix in the current theory;

(iii) σ not of the form σ_1`->`σ_2`->`σ_3.

```
new_binder : (string # type) -> void
```

Executing `new_binder(‘`b`‘,`σ`)` declares b to be a new constant with generic type σ and binder status. Failure if:

(i) not in draft mode;

(ii) there already exists a constant named b in the current theory;

(iii) σ not of the form `(`σ_1`->`σ_2`)->`σ_3.

```
new_axiom : (string # term) -> thm
```

Executing `new_axiom(‘`$name$`‘,`t`)` declares the sequent `({},`t`)` to be an axiom of the current theory with name $name$. Failure if:

(i) not in draft mode;

(ii) there is already an axiom, definition or specification named *name* in the current theory segment.

Once a theorem has been proved, it can be saved with the function

```
save_thm : (string # thm) -> thm
```

Evaluating `save_thm(`*name*`,`*th*`)` will save the theorem *th* with name *name* in the current theory segment.

The following function is used to finish a session in draft mode and save the current theory segment in a theory file on disk.

```
close_theory : void -> void
```

Warning: quitting HOL without closing the session with `close_theory` may result in the theory segment created during the session being lost (i.e. not saved in a theory file)!

11.8.2 Functions for creating definitional extensions

There are three kinds of definitional extensions: constant definitions, constant specifications and type definitions.

11.8.2.1 Constant definitions

In Section 10.5.1 a constant definition over a signature Σ_Ω is defined to be an equation, i.e. a formula of the form $c_\sigma = t_\sigma$, such that:

(i) c is not the name of any constant in Σ_Ω;

(ii) t_σ is a closed term in $\mathsf{Terms}_{\Sigma_\Omega}$;

(iii) all the type variables occurring in t_σ occur in σ.

In HOL, definitions can be slightly more general than this, in that an equation:

$$c\ v_1\ \cdots\ v_n\ =\ t$$

is allowed to be a definition where v_1, ..., v_n are variable structures (i.e. tuples of distinct variables). Such an equation is logically equivalent to:

$$c\ =\ \lambda v_1\ \cdots\ v_n.\ t$$

which is a definition in the sense of Section 10.5.1 if (i), (ii) and (iii) hold.

The following ML function creates a new definition in the current theory.

```
new_definition : (string # term) -> thm
```

Evaluating `new_definition(`*name*`, "`$c\ v_1\ \cdots\ v_n\ =\ t$`")`, where c is not already a constant, declares the sequent `({},`$\lambda v_1\ \cdots\ v_n.\ t$`)` to be a constant definition of the current theory. The name associated with the definition in this theory is *name*. Failure if:

(i) not in draft mode;

(ii) there is already an axiom, definition or specification named *name* in the current theory segment;

(iii) c is already a constant in the current theory;

(iv) t contains free variables that are not in any of the variable structures v_1, ..., v_n (this is equivalent to requiring $\lambda v_1\ \cdots\ v_n.\ t$ to be a closed term);

(v) there is a type variable in v_1, ..., v_n or t that does not occur in the type of c.

The following two functions declare new constants that have infix or binder status. The failure conditions are the same as for `new_definition`, with the additional conditions that the constants being declared must be types of the appropriate form (σ_1`->`σ_2`->`σ_3 for infixes and `(`σ_1`->`σ_2`)->`σ_3 for binders).

```
new_infix_definition  : (string # term) -> thm
new_binder_definition : (string # term) -> thm
```

Note that until an infix or binder declaration has been processed the constant being defined will not have its special status. It is therefore necessary to use the constant in normal prefix position when making the definition. For example,

`new_infix_definition(`*ix*`_DEF", "`$m\ ix\ n$` = ... ")`

will not work, as at the time when the quotation is parsed, `ix` does not have infix status and hence `m ix n` will parse with `m` as the function. The definition must thus have the form:

`new_infix_definition(`*ix*`_DEF", "`$ix\ m\ n$` = ... ")`

Only after this has been processed will *ix* be an infix. It is a common practice among HOL users to write a `$` before the constant being defined as an infix or binder to indicate that after the definition is made, it will have a special syntactic status; i.e. to write:

`new_infix_definition(`*ix*`_DEF", "$`$ix\ m\ n$` = ... ")`

and similarly with `new_binder_definition`. This use of `$` is not necessary; but after the definition has been made `$` must, of course, be used if the syntactic status needs to be suppressed.

11.8.2.2 Constant specifications

In Section 10.5.2 a constant specification for a theory $\mathcal{T}$ is defined to be a pair:

$$\langle (c_1, \ldots, c_n),\ \lambda x_{1\sigma_1} \cdots x_{n\sigma_n}.\ t_{bool} \rangle$$

such that:

(i) c_1, ..., c_n are distinct names that are not the names of any constants in $\mathsf{Sig}_{\mathcal{T}}$.

(ii) $\lambda x_{1\sigma_1} \cdots x_{n\sigma_n}.\ t_{bool}\ \in\ \mathsf{Terms}_{\mathcal{T}}$.

(iii) $tyvars(\lambda x_{1\sigma_1} \cdots x_{n\sigma_n}.\ t_{bool})\ \subseteq\ tyvars(\sigma_i)$ for $1 \leq i \leq n$.

(iv) $\exists x_{1\sigma_1}\ \cdots\ x_{n\sigma_n}.\ t\ \in\ \mathsf{Theorems}_{\mathcal{T}}$.

The following ML function is used to make constant specifications in the HOL system.

```
new_specification : string -> ((string#string)list) -> thm -> thm
```

Evaluating:

```
new_specification
 ‘name‘
 [‘flag1‘,‘c1‘; ... ; ‘flagn‘,‘cn‘]
 |- ?x1 ··· xn. t[x1, ... ,xn]
```

simultaneously introduces new constants named c_1, ..., c_n satisfying the property:

```
|- t[c1, ... ,cn]
```

If ‘$flag_i$‘ is ‘`constant`‘ then c_i is declared an ordinary constant, if it is ‘`infix`‘ then c_i is declared an infix and if it is ‘`binder`‘ then c_i is declared a binder. Any other value of flag causes an error. This theorem is stored, with name *name*, as a definition in the current theory segment. A call to `new_specification` fails if:

(i) not in draft mode;

(ii) there is already an axiom, definition or specification named *name* in the current theory segment;

(iii) the theorem argument has a non-empty assumption list;

(iv) there are free variables in the theorem argument;

(v) c_1, ..., c_n are not distinct variables;

(vi) some c_i is already a constant in the current theory;

(vii) some c_i is not an allowed name for a constant;

(viii) some $flag_i$ is not either `constant`, `infix` or `binder`;

(ix) the type of c_i is not suitable for a constant with the syntactic status specified by $flag_i$;

(x) the type of some c_i does not contain all the type variables which occur in the term `\`x_1 $\cdots$ x_n`. `t`[`x_1`, ... ,`x_n`]`.

11.8.2.3 Type definitions

In Section 10.5.4 it is explained that defining a new type $(\alpha_1, \ldots, \alpha_n)op$ in a theory $\mathcal{T}$ consists of introducing op as a new n-ary type operator and

$$\vdash \exists f_{(\alpha_1,\ldots,\alpha_n)op\rightarrow\sigma}.\ \mathsf{Type_Definition}\ p\ f$$

as a new axiom, where p is a predicate characterizing a non-empty subset of an existing type σ. Formally, a type definition for a theory $\mathcal{T}$ is a 3-tuple

$$\langle \sigma,\ (\alpha_1, \ldots, \alpha_n)op,\ p_{\sigma\rightarrow bool}\rangle$$

where:

(i) $\sigma \in \mathsf{Types}_{\mathcal{T}}$ and $tyvars(\sigma) \in \{\alpha_1, \ldots, \alpha_n\}$.

(ii) op is not the name of a type constant in $\mathsf{Struc}_{\mathcal{T}}$.

(iii) $p \in \mathsf{Terms}_{\mathcal{T}}$ is a closed term of type $\sigma\rightarrow bool$ and $tyvars(p) \subseteq \{\alpha_1, \ldots, \alpha_n\}$.

(iv) $\exists x_\sigma.\ p\ x\ \subseteq\ \mathsf{Theorems}_{\mathcal{T}}$.

The following ML function makes a type definition in the HOL system.

```
new_type_definition : (string # term # thm) -> thm
```

If t is a term of type σ`->bool` containing n distinct type variables, then evaluating:

`new_type_definition(`‘op‘`,` t`, |- ?`x`.` t x`)`

results in op being declared as a new n-ary type operator characterized by the definitional axiom:

`|- ?rep. TYPE_DEFINITION` t `rep`

which is stored as a definition with the automatically generated name *op*_TY_DEF.[6] The constant TYPE_DEFINITION is defined in the theory bool by:

```
|- TYPE_DEFINITION (P:*->bool) (rep:**->*) =
    (!x' x''. (rep x' = rep x'') ==> (x' = x'')) /\
    (!x. P x = (?x'. x = rep x'))
```

Executing new_type_definition(‘*op*‘, *t*, |- ?*x*. *t x*) fails if:

(i) not in draft mode;

(ii) *op* is already the name of a type or type operator in the current theory;

(iii) there already exists a constant definition, constant specification, type definition or axiom named *op*_TY_DEF in the current theory;

(iv) *t* does not have a type of the form *σ*->bool.

11.8.2.4 Defining bijections

The result of a type definition using new_type_definition is a theorem which asserts only the *existence* of a bijection from the type it defines to the corresponding subset of an existing type. To introduce constants that in fact denote such a bijection and its inverse, the following ML function is provided:

```
define_new_type_bijections : string -> string -> string -> thm -> thm
```

This function takes three string arguments and a theorem argument. The theorem argument must be a definitional axiom of the form returned by new_type_definition. The first string argument is the name under which the constant definition (a constant specification, in fact) made by define_new_type_bijections will be stored in the current theory segment, and the second and third string arguments are user-specified names for the two constants that are to be defined. These constants are defined so as to denote mutually inverse bijections between the defined type, whose definition is given by the supplied theorem, and the representing type of this defined type.

Evaluating:

```
define_new_type_bijections ‘name‘ ‘abs‘ ‘rep‘
        |- ?rep:newty->ty. TYPE_DEFINITION P rep
```

automatically defines two new constants *abs*:ty->newty and *rep*:ty->newty such that:

[6] In previous versions of HOL, type definitions were stored as axioms rather than definitions.

```
|- (!a. abs(rep a) = a) /\ (!r. P r = (rep(abs r) = r))
```

This theorem, which is the defining property for the constants *abs* and *rep*, is stored under the name '*name*' in the current theory segment. It is also the value returned by `define_new_type_bijections`. The theorem states that *abs* is the left inverse of *rep* and—for values satisfying P—that *rep* is the left inverse of *abs*.

A call to `define_new_type_bijections` *name abs rep th* fails if:

(i) not in draft mode;

(ii) either *abs* or *rep* is already the name of a constant in the current theory;

(iii) there already exists a constant definition, constant specification, type definition or axiom named *name* in the current theory;

(iv) *th* is not a theorem of the form returned by `new_type_definition`.

11.8.2.5 Properties of type bijections

The following ML functions are provided for proving that the bijections introduced by `define_new_type_isomorphisms` are injective (one-to-one) and surjective (onto):

```
prove_rep_fn_one_one : thm -> thm
prove_rep_fn_onto    : thm -> thm
prove_abs_fn_one_one : thm -> thm
prove_abs_fn_onto    : thm -> thm
```

The theorem argument to each of these functions must be a theorem of the form returned by `define_new_type_bijections`:

```
|- (!a. abs(rep a) = a) /\ (!r. P r = (rep(abs r) = r))
```

If *th* is a theorem of this form, then evaluating `prove_rep_fn_one_one` *th* proves that the function *rep* is one-to-one, and returns the theorem:

```
|- !a a'. (rep a = rep a') = (a = a')
```

Likewise, `prove_rep_fn_onto` *th* proves that *rep* is onto the set of values that satisfy P:

```
|- !r. P r = (?a. r = rep a)
```

Evaluating `prove_abs_fn_one_one` *th* proves that *abs* is one-to-one for values that satisfy P, and returns the theorem:

```
|- !r r'. P r ==> P r' ==> ((abs r = abs r') = (r = r'))
```

And evaluating `prove_abs_fn_onto` *th* proves that *abs* is onto, returning the theorem:

```
|- !a. ?r. (a = abs r) /\ P r
```

All four functions will fail if applied to any theorem that does not have the form of a theorem returned by `define_new_type_bijections`. None of these functions saves anything on the current theory file. In fact, it should usually be unnecessary to save the results proved by these functions, since they can be generated quickly whenever required from the theorem returned by `define_new_type_bijections`, which is itself saved. Of course, within any given session, one would bind results to ML identifiers.

11.8.3 Primitive ML function for accessing theories

The arguments of ML type `string` to `new_axiom`, `new_definition` etc. are the names of the corresponding axioms and definitions. These names are used when accessing theories with the functions `axiom`, `definition`, etc., described below.

A theory with no descendants can be extended by adding new parents, types, constants, axioms and definitions. Theories that are already the parents of other theories cannot be extended in this way.[7] When a new theory is being created or an existing one extended it is necessary to be in draft mode. In proof mode the functions with prefix '`new_`' listed above are not available.

The functions for entering an already existing theory in either draft mode or proof mode are, respectively:

```
extend_theory : string -> void
load_theory   : string -> void
```

The function `close_theory` (see Section 11.8.1) is used to finish a session and write all new declarations to the theory file.

There are various functions for loading the contents of theory files:

```
parents     : string -> string list
types       : string -> (int # string) list
constants   : string -> term list
infixes     : string -> term list
binders     : string -> term list
axioms      : string -> (string # thm) list
definitions : string -> (string # thm) list
theorems    : string -> (string # thm) list
```

[7] It would be difficult to implement the necessary checks to ensure that added types, constants etc. did not invalidate declarations in the descendant theories.

The first argument is the name of a theory (which must be in the ancestry of the current theory segment); the result is a list of the components of the theory. The name of the current theory can be abbreviated by `‘-‘`. For example, `parents ‘-‘` returns the parents of the current theory.

In the case of `types` a list of arity-name pairs is returned; in the case of `axioms`, `definitions` or `theorems` a list of string-theorem pairs is returned, where the string is the name of the theorem representing the axiom, definition or theorem that was supplied by the user. Note that constant specifications and type definitions are both retrieved using the function `definitions`.

Individual axioms, definitions and theorems can be read (from the current or ancestor theories) using the following ML functions:

```
axiom      : string -> string -> thm
definition : string -> string -> thm
theorem    : string -> string -> thm
```

The first argument is the theory (`‘-‘` can be used for the current theory); the second argument is the user supplied name of the axiom, definition or theorem in the theory.

Theories can be printed using the function `print_theory`, which takes a theory name and then prints out the named theory in a readable format.

11.8.4 The theory `HOL`

At start-up, the initial theory for users of the HOL system is called `HOL`, which is constructed when the HOL system is made. The ancestry of this theory is quite complicated and reflects, to some extent, the history of HOL as it evolved from LCF. This ancestry, which is described in detail in Section 11.9, contains two important theories: `bool` and `ind`. These define the primitive logical basis of the HOL logic.

11.8.4.1 The theory `bool`

The theory `bool` introduces the type `bool` and contains four of the five axioms for higher order logic (the fifth axiom is in the theory `ind`). These axioms, together with the rules of inference described later in Section 11.8.5, constitute the core of the HOL logic. Because of the way the HOL system evolved from LCF,[8] the particular axiomatization of higher order logic it uses differs from the classical axiomatization due to Church [3]. The biggest difference is that in Church's formulation type variables are in the meta-language, whereas in the HOL logic they are part of the object language.

There are three primitive constants in the theory `bool`: `=` (equality, an infix), `==>` (implication, an infix) and `@` (choice, a binder). Equality and implication are standard

[8]To simplify the porting of the LCF theorem-proving tools to the HOL system, the HOL logic was made as like PPλ (the logic built-in to LCF) as possible.

predicate calculus notions, but choice is more exotic: if t is a term having type σ`->bool`, then `@x.`t `x` (or, equivalently, `$@`t) denotes *some* member of the set whose characteristic function is t. If the set is empty, then `@x.`t `x` denotes an arbitrary member of the set denoted by σ. The constant `@` is a higher order version of Hilbert's ε-operator; it is related to the constant ι in Church's formulation of higher order logic. For more details, see Church's original paper [3], Leisenring's book on Hilbert's ε-symbol [12], or Andrews' textbook on type theory [1].

The logical constants `T` (truth), `F` (falsity), `~` (negation), `/\` (conjunction), `\/` (disjunction), `!` (universal quantification), `?` (existential quantification) and `?!` (unique existence quantifier) can all be defined in terms of equality, implication and choice. The definitions listed below are fairly standard; each one is preceded by its ML name. (Later definitions sometimes use earlier ones.)

```
T_DEF              |- T   = ((\x:bool. x)=(\x. x))

FORALL_DEF         |- $!  = \P:*->bool. P=(\x. T)

EXISTS_DEF         |- $?  = \P:*->bool. P($@ P)

AND_DEF            |- $/\ = \t1 t2. !t. (t1 ==> t2 ==> t) ==> t

OR_DEF             |- $\/ = \t1 t2. !t. (t1 ==> t) ==> (t2 ==> t) ==> t

F_DEF              |- F   = !t. t

NOT_DEF            |- $~  = \t. t ==> F

EXISTS_UNIQUE_DEF  |- $?! = (\P. $? P /\ (!x y. P x /\ P y ==> (x = y)))
```

There are four axioms in the theory `bool`:

```
BOOL_CASES_AX   |- !t. (t = T) \/ (t = F)

IMP_ANTISYM_AX  |- !t1 t2. (t1 ==> t2) ==> (t2 ==> t1) ==> (t1 = t2)

ETA_AX          |- !t. (\x. t x) = t

SELECT_AX       |- !P:*->bool x. P x ==> P($@ P)
```

The fifth and last axiom of the HOL logic is the Axiom of Infinity; this is in the theory `ind` described in Section 11.8.4.2.

The theory `bool` also supplies the definitions of a number of useful constants. The constant `LET` is used in representing terms containing local variable bindings (i.e. `let`-terms, as discussed in Section 11.4), and the constant `COND` is used in representing conditionals. Both constants are defined in the theory `bool`, and have the following definitions:

```
LET_DEF      |- LET       = \f x. f x

COND_DEF     |- COND      = \t t1 t2.@x.((t=T)==>(x=t1))/\((t=F)==>(x=t2))
```

The theory `bool` also contains the definitions of the constants `RES_FORALL`, `RES_EXISTS`, `RES_SELECT`, `ARB` and `RES_ABSTRACT`, which are used to support restricted quantification in the HOL logic (see Section 11.5). The definitions are:

```
RES_FORALL   |- !P B. RES_FORALL P B = (!x. P x ==> B x)

RES_EXISTS   |- !P B. RES_EXISTS P B = (?x. P x /\ B x)

RES_SELECT   |- !P B. RES_SELECT P B = (@x. P x /\ B x)

ARB          |- ARB = (@x. T)

RES_ABSTRACT |- !P B. RES_ABSTRACT P B = (\x. (P x => B x | ARB))
```

The theory `bool` also contains the definitions of the constants `ONE_ONE` and `ONTO`, which are used in stating the Axiom of Infinity (see Section 11.8.4.2). The definitions are:

```
ONE_ONE_DEF |- ONE_ONE f = (!x1 x2. (f x1 = f x2) ==> (x1 = x2))

ONTO_DEF    |- ONTO f    = (!y. ?x. y = f x)
```

For further discussion of the theory `bool` see Section 11.9.2.

11.8.4.2 The theory `ind`

The theory `ind` introduces the type `ind` of *individuals* and the *Axiom of Infinity*. This axiom states that the set denoted by `ind` is infinite. The four axioms of the theory `bool`, the rules of inference in Section 11.8.5 and the Axiom of Infinity are, together, sufficient for developing all of standard mathematics. Thus, in principle, the user of the HOL system should never need to make a non-definitional theory. In practice, it is often very tempting to take the risk of introducing new axioms because deriving them from definitions can be tedious—proving that 'axioms' follow from definitions amounts to proving their consistency.

The Axiom of Infinity is

```
INFINITY_AX  |- ?f:ind->ind. ONE_ONE f /\ ~(ONTO f)
```

This asserts that there exists a one-to-one map from `ind` to itself that is not onto. This implies that the type `ind` denotes an infinite set.

11.8.5 Primitive rules of inference of the HOL Logic

The primitive rules of inference of the logic were described abstractly in Section 10.3.1. The descriptions relied on meta-variables t, t_1, t_2, and so on. In the HOL logic, infinite families of primitive inferences are grouped together and thought of as single primitive inference schemes. Each family contains all the concrete instances of one particular inference 'pattern'. These can be produced, in abstract form, by instantiating the meta-variables in Section 10.3.1 to concrete terms.

In HOL, primitive inference schemes are represented by ML functions that return theorems as values. That is, for particular HOL terms, the ML functions return the instance of the theorem at those terms. The ML functions are part of the ML abstract type `thm`: although `thm` has no primitive constructors, it has (eight) operations which return theorems as values: `ASSUME`, `REFL`, `BETA_CONV`, `SUBST`, `ABS`, `INST_TYPE`, `DISCH` and `MP`.

The ML functions that implement the primitive inference schemes in the HOL system are described below. The same notation is used here as in Section 10.3.1: hypotheses above a horizontal line and conclusion beneath. The machine-readable ASCII notation is used for the logical constants.

11.8.5.1 Assumption introduction

```
ASSUME : term -> thm
```

$$\frac{}{t \texttt{ |- } t}$$

`ASSUME` t evaluates to t`|-` t. Failure if t is not of type `bool`.

11.8.5.2 Reflexivity

```
REFL : term -> thm
```

$$\frac{}{\texttt{|- } t \texttt{ = } t}$$

`REFL` t evaluates to `|-` t `=` t. A call to `REFL` never fails.

11.8.5.3 Beta-conversion

```
BETA_CONV : term -> thm
```

$$\frac{}{\texttt{|- } (\backslash x\,.\,t_1)t_2 \texttt{ = } t_1[t_2/x]}$$

- where $t_1[t_2/x]$ denotes the result of substituting t_2 for x in t_1, with suitable renaming of variables to prevent free variables in t_2 becoming bound after substitution. The substitution $t_1[t_2/x]$ is always defined.

`BETA_CONV` $(\backslash x.t_1)t_2$ evaluates to the theorem `|-` $(\backslash x.t_1)t_2$ `=` $t_1[t_2/x]$. Failure if the argument to `BETA_CONV` is not a β-redex (i.e. is not of the form $(\backslash x.t_1)t_2$).

11.8.5.4 Substitution

```
SUBST : (thm # term)list -> term -> thm -> thm
```

$$\frac{\Gamma_1 \texttt{ |- } t_1\texttt{=}t'_1 \quad \cdots \quad \Gamma_n \texttt{ |- } t_n\texttt{=}t'_n \qquad \Gamma \texttt{ |- } t[t_1,\ldots,t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \texttt{ |- } t[t'_1,\ldots,t'_n]}$$

- where $t[t_1,\ldots,t_n]$ denotes a term t with some free occurrences of the terms t_1, ..., t_n singled out and $t[t'_1,\ldots,t'_n]$ denotes the result of simultaneously replacing each such occurrences of t_i by t'_i (for $1{\leq}i{\leq}n$), with suitable renaming of variables to prevent free variables in t'_i becoming bound after substitution.

The first argument to `SUBST` is a list `[(|-`t_1`=`t'_1`,` x_1`);` ... `;(|-`t_n`=` t'_n`,` x_n`)]`. The second argument is a template term $t[x_1,\ldots,x_n]$ in which occurrences of the variable x_i (where $1 \leq i \leq n$) are used to mark the places where substitutions with `|-` t_i`=`t'_i are to be done. Thus

`SUBST [(|-`t_1`=`t'_1`,` x_1`);...;(|-`t_n`=` t'_n`,` x_n`)]` $t[x_1,\ldots,x_n]$ `|-` $t[t_1,\ldots,t_n]$

returns Γ `|-` $t[t'_1,\ldots,t'_n]$. Failure if:

(i) any of the arguments are of the wrong form;

(ii) the type of x_i is not equal to the type of t_i for some $1 \leq i \leq n$.

11.8.5.5 Abstraction

```
ABS : term -> thm -> thm
```

$$\frac{\Gamma \texttt{ |- } t_1 \texttt{ = } t_2}{\Gamma \texttt{ |- } (\backslash x.t_1) \texttt{ = } (\backslash x.t_2)}$$

- where x is not free in Γ.

`ABS` x Γ `|-` t_1`=`t_2 returns the theorem Γ `|-` $(\backslash x.t_1)$ `=` $(\backslash x.t_2)$. Failure if x is not a variable, or x occurs free in any assumption in Γ.

11.8.5.6 Type instantiation

```
INST_TYPE : (type#type) list -> thm -> thm
```

$$\frac{\Gamma \texttt{ |- } t}{\Gamma \texttt{ |- } t[\sigma_1, \ \ldots \ , \sigma_n/\alpha_1, \ \ldots \ , \alpha_n]}$$

- $t[\sigma_1, \ \ldots \ , \sigma_n/\alpha_1, \ \ldots \ , \alpha_n]$ denotes the result of substituting (in parallel) the types $\sigma_1, \ldots, \sigma_n$ for the type variables $\alpha_1, \ldots, \alpha_n$ in t, with the restriction that none of $\alpha_1, \ldots, \alpha_n$ occur in Γ.

`INST_TYPE[(`σ_1`,`α_1`);...;(`σ_n`,`α_n`)]` th returns the result of instantiating each occurrence of α_i in the theorem th to σ_i (for $1 \leq i \leq n$). Failure if:

(i) arguments of the wrong form (e.g. an α_i is not a type variable);

(ii) α_i (for $1 \leq i \leq n$) occurs in any assumption in Γ.

11.8.5.7 Discharging an assumption

```
DISCH : term -> thm -> thm
```

$$\frac{\Gamma \texttt{ |- } t_2}{\Gamma - \{t_1\} \texttt{ |- } t_1 \texttt{ ==> } t_2}$$

- $\Gamma - \{t_1\}$ denotes the set obtained by removing t_1 from Γ (note that t_1 need not occur in Γ; in this case $\Gamma - \{t_1\} = \Gamma$).

`DISCH` t_1 Γ `|-` t_2 evaluates to the theorem $\Gamma - \{t_1\}$ `|-` t_1 `==>` t_2. `DISCH` fails if the term given as its first argument is not of type `bool`.

11.8.5.8 Modus Ponens

```
MP : thm -> thm -> thm
```

$$\frac{\Gamma_1 \texttt{ |- } t_1 \texttt{ ==> } t_2 \qquad \Gamma_2 \texttt{ |- } t_1}{\Gamma_1 \cup \Gamma_2 \texttt{ |- } t_2}$$

`MP` takes two theorems (in the order shown above) and returns the result of applying Modus Ponens; it fails if the arguments are not of the right form.

11.8.6 Type abbreviations

It is possible to introduce an abbreviation for a monomorphic type using the function:

```
new_type_abbrev : (string # type) -> void
```

Evaluating `new_type_abbrev(`‘*name*‘`,":`σ`")` enables *name* to be used in quotations instead of σ. The evaluation fails if σ is polymorphic. Type abbreviations are recorded in theory files, so that when a theory is loaded, any type abbreviations made are activated. The list of currently active abbreviations in a theory is given by the function:

```
type_abbrevs : string -> (string # type) list
```

Note that abbreviation can also be made using antiquotation, without the restriction to monomorphic types. Such ML abbreviations are not, of course, stored in theory files and so do not persist beyond a single session. The following session illustrates various ways of abbreviating types:

1

```
#new_theory ‘numpair‘;;
() : void

#new_type_abbrev(‘numpair‘, ":num#num");;
() : void

#let t1 = "x:numpair";;
t1 = "x" : term

#type_of t1;;
":num # num" : type

#":numpair" = ":num#num";;
true : bool
```

The alternative to introducing a type abbreviation is to give an ML name to the type, and then to use this name via antiquotation. Continuing the session:

2

```
#let ty = ":num#num";;
ty = ":num # num" : type

#let t2 = "x:^ty";;
t2 = "x" : term

#t1 = t2;;
true : bool
```

The type abbreviation is stored in the theory file and so persists across sessions. This can be seen by the result of printing the theory `numpair`:

```
#print_theory‘numpair‘;;
The Theory numpair
Parents --  HOL
Type Abbreviations --  numpair ":num # num"
******************** numpair ********************

() : void
```

3

If the session is then ended:

```
#close_theory();;
() : void

#quit();;
```

4

and a new session is started in which the theory `numpair` is loaded:

```
#load_theory‘numpair‘;;
Theory numpair loaded
() : void

#"x:numpair";;
"x" : term

#type_abbrevs ‘-‘;;
[(‘numpair‘, ":num # num")] : (string # type) list
```

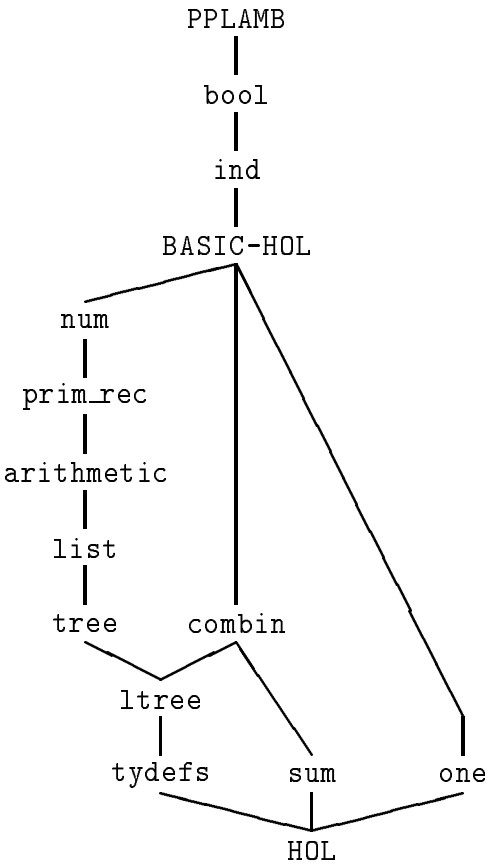
1

then the type abbreviation persists.

Type abbreviations tend to be little used in practice; the antiquotation method is usually sufficient.

11.9 The ancestry of the theory HOL

The ancestry of the theory `HOL` is:



In the rest of this section, each of the theories in the ancestry of `HOL` is briefly described. A complete list of all the definitions and theorems in each theory is not given here; the sections that follow provide only an overview of the contents of each theory. For a complete list of all the built-in axioms, definitions and theorems in HOL, see *REFERENCE*.

11.9.1 The theory `PPLAMB`

The most primitive theory is called `PPLAMB` and is a stripped down version of the theory underlying LCF. The HOL theory `PPLAMB` does not represent any logical principles but is an interface between the HOL and LCF systems (it contains the declaration of the primitive type operator `->`). For a while the two systems developed in parallel, upgrades in LCF from Gerard Huet of INRIA or Larry Paulson of Cambridge resulting in upgrades in HOL.

11.9.2 The theory `bool`

The theory `bool` has already been partly described in Section 11.8.4.1. It contains the definitions of several sorts of constants:

(i) logical constants: `T`, `F`, `~`, `/\`, `\/`, `==>`, `=`, `!`, `?`, `?!` and `@`.

(ii) system constants, which are used in coding HOL theories as LCF theories. These constants are `HOL_DEFINITION` and `BINDERS`. They have no logical significance and are part of the internal workings of the HOL system.

(iii) miscellaneous constants, which support various special syntactic forms: `COND` (for conditionals, see Section 11.5.1); `LET` (for `let`-terms, see Section 11.5.3); `CURRY` and `UNCURRY` (for paired abstractions, see Section 11.5.2); and `RES_FORALL`, `RES_EXISTS`, `RES_SELECT`, `RES_ABSTRACT` and `ARB` (for restricted quantification, see Section 11.5).

(iv) the constant `TYPE_DEFINITION`, which is used by the type definition mechanism (see Section 11.8.2.3).

(v) constants associated with pairs: `,` (i.e. the comma symbol), `MK_PAIR`, `IS_PAIR`, `REP_prod`, `FST` and `SND` (see Section 11.9.2.1 below).

11.9.2.1 Pairs and the type `prod`

The Cartesian product type operator `prod` is defined in the theory `bool`, although logically it should really be in a separate theory. The reason for its definition early on is that certain syntactic forms (e.g. paired abstractions) presuppose pairs.

Values of type `(`σ_1`,`σ_2`)prod` are ordered pairs whose first component has type σ_1 and whose second component has type σ_2. The HOL parser converts type expressions of the form "`:`σ_1`#`σ_2" into `(`σ_1`,`σ_2`)prod`, and the printer inverts this transformation. Pairs are constructed with an infixed comma symbol

```
$, : * -> ** -> *#**
```

so, for example, if t_1 and t_2 have types σ_1 and σ_2 respectively, then "t_1`,`t_2" is a term with type σ_1`#`σ_2. It is usual, but not necessary, to write pairs within brackets: "`(`t_1`,`t_2`)`". The comma symbol associates to the right, so that "`(`t_1`,`t_2`,...,`t_n`)`" means "`(`t_1`,(`t_2`,...,`t_n`))`". The constants

```
FST : * # ** -> *
SND : * # ** -> **
```

select the first and second components of pairs.

Cartesian products are defined by representing a pair `(`t_1`,`t_2`)` by the function

```
\a b. (a=t1) /\ (b=t2)
```

The representing type of σ_1`#`σ_2 is thus σ_1`->`σ_2`->bool`. To define pairs this way, the constants `MK_PAIR` and `IS_PAIR` are first defined.

```
MK_PAIR_DEF   |- !x y. MK_PAIR x y = (\a b. (a = x) /\ (b = y))

IS_PAIR_DEF   |- !p. IS_PAIR p = (?x y. p = MK_PAIR x y)
```

From these two definitions it is easy to prove that:

```
|- ?p:*->**->bool. IS_PAIR p
```

since `|- IS_PAIR(MK_PAIR x y)` follows easily from the definition of `IS_PAIR`. The existence theorem shown above is called `PAIR_EXISTS`. Given this theorem, the type operator `prod` is defined by evaluating:

```
new_type_definition(‘prod‘, "IS_PAIR:(*->**->bool)->bool", PAIR_EXISTS)
```

which results in the definitional axiom `prod_TY_DEF` shown below being asserted in the theory `bool`.

```
prod_TY_DEF  |- ?rep. TYPE_DEFINITION IS_PAIR rep
```

Next, a new constant `REP_prod` is defined, which maps a pair to its representation as a function:

```
REP_prod    |- REP_prod =
               (@rep : * # ** -> (* -> (** -> bool)).
                 (!p' p''. (rep p' = rep p'') ==> (p' = p'')) /\
                 (!p. IS_PAIR p = (?p'. p = rep p')))
```

The infix constructor ‘,’ and the selectors `FST:*#**->*` and `SND:*#**->**` are then defined by the equations shown below.

```
COMMA_DEF  |- !x y. x,y = (@p. REP_prod p = MK_PAIR x y)

FST_DEF    |- !p. FST p = (@x. ?y. MK_PAIR x y = REP_prod p)

SND_DEF    |- !p. SND p = (@y. ?x. MK_PAIR x y = REP_prod p)
```

The following standard theorems about pairs follow easily from these definitions and the axiom `prod_TY_DEF`. Although these theorems could be derived by formal proof, they are (for implementation reasons) asserted as axioms in the theory `bool` .

```
PAIR     |- !x. (FST x,SND x) = x

FST      |- !x y. FST(x,y) = x

SND      |- !x y. SND(x,y) = y

PAIR_EQ  |- !x y a b. (x,y = a,b)  =  (x = a) /\ (y = b)
```

11.9.3 The theory BASIC-HOL

The theory BASIC-HOL roughly corresponds to the theory INIT described in Section 10.4.3. It consists of the data in the theories bool and ind, together with one pre-proved theorem called ABS_REP_THM, which establishes various standard properties of isomorphisms and is used for deriving consequences of type definitions (see Section 11.8.2.3).

The descendents of BASIC-HOL are obtained by definitional extension. There are, however, a few impurities in these extensions, with the result that not all of them could be built purely by using the definitional mechanisms of HOL. The theory num, for example, contains an infinite family of constants 0, 1, 2 etc., but there is currently no mechanism that allows users to define such infinite families.

11.9.4 The theory num

The theory num defines the type num of natural numbers to be isomorphic to a countable subset of the primitive type ind. In this theory, the constants 0 and SUC (the successor function) are defined and Peano's axioms pre-proved in the form:

```
NOT_SUC    |- !n. ~(SUC n = 0)
INV_SUC    |- !m n. (SUC m = SUC n) ==> (m = n)
INDUCTION  |- !P. P 0 /\ (!n. P n ==> P(SUC n)) ==> (!n. P n)
```

In higher order logic, Peano's axioms are sufficient for developing number theory because addition and multiplication can be defined. In first order logic these must be taken as primitive. Note also that INDUCTION could not be stated as a single axiom in first order logic because predicates (e.g. P) cannot be quantified.

Uses of the theorem INDUCTION are supported by the built-in derived inference rule INDUCT and the built-in tactic INDUCT_TAC (see the documentation on these functions in *REFERENCE* for details).

11.9.4.1 Numerals

Associated with the theory num are the numerals of type :num, an infinite family of constants: 1, 2, 3, etc. These can be regarded logically as an infinite collection of defined constants, introduced by the infinite series of definitional axioms:

```
|- 1 = SUC 0 ,   |- 2 = SUC 1 ,   |- 3 = SUC 2 ,   etc.
```

This infinite list of theorems cannot, of course, actually be stored in the theory num. HOL therefore provides the ML function:

```
num_CONV : term -> thm
```

which can be used to generate the defining equation for any constant number of type `num`. For example:

```
#let thm1 = num_CONV "1";;                                                    1
thm1 = |- 1 = SUC 0

#let thm2 = num_CONV "2";;
thm2 = |- 2 = SUC 1

#let thm3 = num_CONV "3141592653";;
thm3 = |- 3141592653 = SUC 3141592652
```

The defining equation for any numeral of type `num` can likewise be obtained using `num_CONV`.

11.9.5 The theory `prim_rec`

In classical logic, unlike domain theory logics such as PPλ, arbitrary recursive definitions are not allowed. For example, there is no function f (of type `num->num`) such that

`!`x`. `$f\ x$` = (`$f\ x$`) + 1`

Certain restricted forms of recursive definition do, however, uniquely define functions. An important example are the *primitive recursive* functions.[9] For any x and f the *primitive recursion theorem* tells us that there is a unique function `fn` such that:

`fn 0 =` x`) /\ (!`n`.fn(SUC `n`) =` f `(fn `n`) `n`)`

The primitive recursion theorem follows from Peano's axioms. When the HOL system is built, the following theorem is proved and stored in the theory `prim_rec`:

```
num_Axiom   |- !x f. ?!fn. (fn 0 = x) /\ (!n. fn(SUC n) = f (fn n) n)
```

The theorem states the validity of primitive recursive definitions on the natural numbers: for any `x` and `f` there exists a corresponding total function `fn` which satisfies the primitive recursive definition whose form is determined by `x` and `f`.

11.9.5.1 Primitive recursive definitions

The primitive recursion theorem can be used to justify any definition of a function on the natural numbers by primitive recursion. For example, a primitive recursive definition in higher order logic of the form

[9] In higher order logic, primitive recursion is much more powerful than in first order logic; for example, Ackermann's function can be defined by primitive recursion in higher order logic.

```
fun 0       x1 ... xi = f1[x1,...,  xi]
fun (SUC n) x1 ... xi = f2[fun n t1 ... ti, n, x1,...,xi]
```

where all the free variables in the terms t_1, ..., t_i are contained in {n, x_1, ..., x_i}, is logically equivalent to:

```
fun 0       = \x1 ... xi.f1[x1,...,xi]
fun (SUC n) = \x1 ... xi.f2[fun n t1 ... ti, n,x1,...,xi]
            = (\f n x1 ... xi.f2[f t1 ... ti, n, x1,...,xi]) (fun n) n
```

The existence of a recursive function `fun` which satisfies these two equations follows directly from the primitive recursion theorem `num_Axiom` shown above. Specializing the quantified variables `x` and `f` in a suitably type-instantiated version of `num_Axiom` so that

```
x=\x1 ... xi.f1[x1,...,xi]   and   f=\f n x1 ... xi.f2[f t1 ... ti, n, x1,...,xi])
```

yields (ignoring the uniqueness of `fn`) the existence theorem shown below:

```
|- ?fn. fn 0       = \x1 ... xi.f1[x1,...,xi] /\
        fn (SUC n) = (\f n x1 ... xi.f2[f t1 ... ti, n, x1,...,xi]) (fn n) n
```

This theorem allows a constant `fun` to be introduced (via the definitional mechanism of constant specifications—see Section 11.8.2.2) to denote the recursive function that satisfies the two equations in the body of the theorem. Introducing a constant `fun` to name the function asserted to exist by the theorem shown above, and simplifying using β-reduction, yields the following theorem:

```
|- fun 0       = \x1 ... xi.f1[x1,...,xi] /\
   fun (SUC n) = \x1 ... xi.f2[fun n t1 ... ti, n, x1,...,xi]
```

It follows immediately from this theorem that the constant `fun` satisfies the primitive recursive defining equations given by the theorem shown below:

```
|- fun 0 x1 ... xi = f1[x1,...,xi]
   fun (SUC n) x1 ... xi = f2[fun n t1 ... ti, n, x1,...,xi]
```

To automate the use of the primitive recursion theorem in deriving recursive definitions of this kind, the HOL system provides two functions which, together, first do automatic proofs of the existence of primitive recursive functions and then make constant specifications to introduce constants that denote such functions:

```
new_prim_rec_definition       : (string # term) -> thm
new_infix_prim_rec_definition : (string # term) -> thm
```

Evaluating

```
new_prim_rec_definition
 (‘fun_DEF‘,
  "(fun 0 x1 ... xi = f1[x1,...,xi] /\
    (fun (SUC n) x1 ... xi = f2[fun n t1 ... ti, n, x1,...,xi])");;
```

automatically proves the theorem:

```
|-  ?fun. !x1 ... xi. fun 0 x1 ... xi = f1[x1,...,xi] /\
          !x1 ... xi. fun (SUC n) x1 ... xi = f2[fun n t1 ... ti, n, x1,...,xi]
```

and then declares a new constant `fun` with this property as its specification. This constant specification is returned as a theorem by `new_prim_rec_definition` and is saved with name `fun_DEF` in the current theory segment. Failure occurs if:

(i) HOL cannot prove there is a function satisfying the specification (i.e. if the term supplied to `new_prim_rec_definition` is not a well-formed primitive recursive definition);

(ii) any other condition for making a constant specification is violated (see the failure conditions for `new_specification` in Section 11.8.2.2).

The ML function `new_prim_rec_definition` is, in fact, slightly more general than is indicated by the example application shown above. In particular, a curried primitive recursive function defined using this ML function can be defined by primitive recursion on any one of its arguments. For example, a curried addition function `plus:num->num->num` can be defined by primitive recursion on its first argument:

```
#let PLUS = new_prim_rec_definition                                        1
#           (‘PLUS‘,
#            "(plus 0 n = n) /\
#             (plus (SUC m) n = SUC(plus m n))");;
PLUS = |- (!n. plus 0 n = n) /\ (!m n. plus(SUC m)n = SUC(plus m n))
```

or by primitive recursion on its second argument:

```
#let PLUS = new_prim_rec_definition                                        1
#            (‘PLUS‘,
#             "(plus m 0 = m) /\
#              (plus m (SUC n) = SUC(plus m n))");;
PLUS = |- (!m. plus m 0 = m) /\ (!m n. plus m(SUC n) = SUC(plus m n))
```

The ML function `new_prim_rec_definition` also allows the user to partially specify the value of a function defined (possibly recursively) on the natural numbers by giving its value for only one of `0` or `SUC n`. For example, a decrement function `DEC`, whose value is specified for only positive natural numbers, can be defined using `new_prim_rec_definition` as follows

```
#let DEC = new_prim_rec_definition                                    1
#            (‘DEC‘, "DEC (SUC n) = n");;
DEC = |- !n. DEC(SUC n) = n
```

This definition specifies the value of the function `DEC` only for positive natural numbers. In particular, the value of `DEC 0` is left unspecified, and the only non-trivial property that can be proved to hold of the constant `DEC` is the property stated by the theorem returned by the call to `new_prim_rec_definition` shown in the session above.

The function `new_infix_prim_rec_definition` can be used to define an infixed function by primitive recursion on the natural numbers. It takes its arguments in the same form as `new_prim_rec_definition` and has similar failure conditions; the only difference is that the resulting function constant has infix status. Here, for example, is the recursive definition of the constant `+` used by the system:

```
new_infix_prim_rec_definition
 (‘ADD‘,
  "($+ 0 n = n) /\
   ($+ (SUC m) n = SUC($+ m n))")
```

The `$`'s are there (as documentation) to indicate that the constant `+` is being declared to be an infix. Evaluating this ML expression will create the following constant specification in the current theory:

```
ADD   |- (!n. 0 + n = n) /\ (!m n. (SUC m) + n = SUC(m + n))
```

For further details about recursive definitions, see Section 11.10.2, or the *REFERENCE* documentation on `new_prim_rec_definition` and `new_infix_prim_rec_definition`.

11.9.5.2 The less-than relation

The less-than relation ‘`<`’ is most naturally defined by primitive recursion. However, it is needed for the proof of the primitive recursion theorem, so it must be defined before definition by primitive recursion is available. The theory `prim_rec` therefore contains the following non-recursive definition of `<`:

```
LESS  |- !m n. m < n = (?P. (!n'. P(SUC n') ==> P n') /\ P m /\ ~P n)
```

This definition says that "`m < n`" if there exists a set (with characteristic function `P`) that is downward closed[10] and contains `m` but not `n`.

[10] A set of numbers is *downward closed* if whenever it contains the successor of a number, it also contains the number.

11.9.6 The theory `arithmetic`

The built-in theory `arithmetic` contains primitive recursive definitions of following standard arithmetic operators.

```
ADD           |- (!n. 0 + n = n) /\
                 (!m n. (SUC m) + n = SUC(m + n))

SUB           |- (!m. 0 - m = 0) /\
                 (!m n. (SUC m) - n = (m < n => 0 | SUC(m - n)))

MULT          |- (!n. 0 * n = 0) /\
                 (!m n. (SUC m) * n = (m * n) + n)

EXP           |- (!m. m EXP 0 = 1) /\
                 (!m n. m EXP (SUC n) = m * (m EXP n))
```

It also contains the following non-recursive definitions.

```
GREATER       |- !m n. m > n = n < m

LESS_OR_EQ    |- !m n. m <= n = m < n \/ (m = n)

GREATER_OR_EQ |- !m n. m >= n = m > n \/ (m = n)

DIVISION      |- !n. 0 < n ==> (!k. (k = ((k DIV n) * n) + (k MOD n)) /\
                                    (k MOD n) < n)
```

An *ad hoc* but useful collection of over a hundred elementary theorems of arithmetic are pre-proved when HOL is built and stored in the theory `arithmetic`. Each theorem will be autoloaded when its name is first mentioned during any HOL session. For a complete list of available theorems, see *REFERENCE*.

11.9.7 The theory `list`

The theory `list` introduces the unary type operator `list` by a type definition.[11] The standard list processing functions are then defined on this type:

```
NIL  : (*)list
CONS : * -> (*)list -> (*)list
HD   : (*)list -> *
TL   : (*)list -> (*)list
NULL : (*)list -> bool
```

[11] For details of the definition, see [7, 14].

The HOL parser has been specially modified to parse the expression `[]` into `NIL` and to parse the expression $[t_1;t_2;\ldots;t_n]$ into `CONS` t_1 `(CONS` $t_2 \cdots$ `(CONS` t_n `NIL)` $\cdots$ `)`. The HOL printer reverses these transformations.

The functions `NIL` and `CONS` are defined in terms of the representing type of lists. From their definitions, the following fundamental theorems about lists are proved and stored in the theory `list`.

```
list_Axiom    |- !x f. ?!fn.(fn[] = x) /\ (!h t. fn(CONS h t) = f(fn t)h t)

list_INDUCT   |- !P. P[] /\ (!t. P t ==> (!h. P(CONS h t))) ==> (!l. P l)

list_CASES    |- !l. (l = []) \/ (?t h. l = CONS h t)

CONS_11       |- !h t h' t'. (CONS h t = CONS h' t') = (h = h') /\ (t = t')

NOT_NIL_CONS  |- !h t. ~([] = CONS h t)

NOT_CONS_NIL  |- !h t. ~(CONS h t = [])
```

A derived rule of structural induction called `LIST_INDUCT` is provided, together with an associated structural induction tactic `LIST_INDUCT_TAC`. These automate the use of the theorem `list_INDUCT`. See the *REFERENCE* documentation on these two functions for details.

The theorem `list_Axiom` shown above is analogous to the primitive recursion theorem on the natural numbers discussed above in Section 11.9.5.1. It states the validity of primitive recursive definitions on lists, and can be used to justify any such definition. The ML functions

```
new_list_rec_definition       : (string # term) -> thm
new_infix_list_rec_definition : (string # term) -> thm
```

use this theorem to do automatic proofs of the existence of primitive recursive functions on lists and then make constant specifications to introduce constants that denote such functions. They are analogous to the corresponding functions `new_prim_rec_definition` and `new_infix_prim_rec_definition` discussed in Section 11.9.5.1. For example, the HOL system defines a length function, `LENGTH`, on lists by the primitive recursive definition on lists shown below:

```
new_list_rec_definition
  (‘LENGTH‘,
  "(LENGTH NIL = 0) /\
   (!h:*. !t. LENGTH (CONS h t) = SUC (LENGTH t))")
```

When this ML expression is evaluated, HOL uses `list_Axiom` to prove existence of a function that satisfies the given primitive recursive definition, introduces a constant to name this function using a constant specification, and stores the resulting theorem:

```
LENGTH   |- (LENGTH[] = 0) /\ (!h t. LENGTH(CONS h t) = SUC(LENGTH t))
```

in the current theory (in this case, the theory `list`).

Using `new_list_rec_definition`, the predicate `NULL` and the selectors `HD` and `TL` are defined in the theory `list` by the specifications:

```
NULL |- NULL[] /\ (!h t. ~NULL(CONS h t))

HD   |- !(h:*) t. HD(CONS h t) = h

TL   |- !(h:*) t. TL(CONS h t) = t
```

The following primitive recursive definitions of functions on lists are also made in the theory `list`:

```
SUM        |- (SUM[] = 0) /\ (!h t. SUM(CONS h t) = h + (SUM t))

APPEND     |- (!l. APPEND[]l = l) /\
              (!l1 l2 h. APPEND(CONS h l1)l2 = CONS h(APPEND l1 l2))

FLAT       |- (FLAT[] = []) /\ (!h t. FLAT(CONS h t) = APPEND h(FLAT t))

LENGTH     |- (LENGTH[] = 0) /\ (!h t. LENGTH(CONS h t) = SUC(LENGTH t))

MAP        |- (!f. MAP f[] = []) /\
              (!f h t. MAP f(CONS h t) = CONS(f h)(MAP f t))

EL         |- (!l. EL 0 l = HD l) /\ (!l n. EL(SUC n)l = EL n(TL l))

EVERY_DEF  |- (!P. EVERY P[] = T) /\
              (!P h t. EVERY P(CONS h t) = P h /\ EVERY P t)
```

There is also a selection of pre-proved theorems about lists stored in the theory `list`. These are autoloaded when their names are first mentioned during any HOL session. For a complete list of available theorems, see *REFERENCE*.

11.9.8 The theory `combin`

The theory `combin` contains the definitions of function composition (infixed `o`) and the combinators `S`, `K` and `I`.

```
o_DEF |- !f g. f o g = (\x. f(g x))

K_DEF |- K = (\x y. x)

S_DEF |- S = (\f g x. f x(g x))

I_DEF |- I = S K K
```

The following elementary properties are pre-proved in the theory `combin`:

```
o_THM |- !f g x. (f o g)x = f(g x)

o_ASSOC |- !f g h. f o (g o h) = (f o g) o h

K_THM |- !x y. K x y = x

S_THM |- !f g x. S f g x = f x(g x)

I_THM |- !x. I x = x

I_o_ID |- !f. (I o f = f) /\ (f o I = f)
```

Having the symbols `o`, `S`, `K` and `I` as built-in constants is sometimes inconvenient because they are often wanted as mnemonic names for variables (e.g. `S` to range over sets and `o` to range over outputs). These names may therefore be changed in future releases of the system, or the theory `combin` may be made into a library. But variables (though not constants) with these names can be used in the current system if `o`, `S`, `K` and `I` are first hidden (see Section 11.11.6).

11.9.9 The theories `tree` and `ltree`

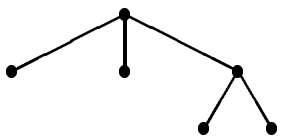
The theories `tree` and `ltree` contain the definitions of two structurally-isomorphic types of finitely-branching ordered trees. The types defined in these theories are used by Tom Melham's type definition package (see Section 11.10) to construct representations for arbitrary concrete recursive types. They are not intended for general use, and the theorems stored in these two built-in theories are therefore not loaded into the system at start-up. The following is a summary of the main theorems which are available in the theories `tree` and `ltree`, and which may be of use in certain specialized applications. For full details of the logical basis for these two theories, see [14].

11.9.9.1 The theory `tree`

In the theory `tree`, a type `tree` is defined to denote the set of all ordered trees whose nodes can branch any (finite) number of times. A constructor function

```
node : (tree)list -> tree
```

is then defined in the theory `tree`. This function can be used to construct any tree-structured value of type `tree`. The expression "`node []`" denotes the tree consisting of a single leaf node with no subtrees. If *tl*:`(tree)list` is a non-empty list of trees, then the term "`node` *tl*" denotes the tree whose immediate subtrees are the trees in the list *tl*. Using `node`, it is possible to construct a tree of any shape. For example, the tree



is denoted by the term "node[node[]; node[]; node[node[]; node[]]".

The next two theorems follow from the formal definition of node and are stored in the theory tree:

```
node_11      |- !tl1 tl2. (node tl1 = node tl2) = (tl1 = tl2)
tree_Induct  |- !P. (!tl. EVERY P tl ==> P(node tl)) ==> (!t. P t)
```

These two theorems are analogous to the Peano postulates for the natural numbers, and are used to prove the following abstract characterization of the defined type tree.

```
tree_Axiom  |- !f. ?! fn. !tl. fn(node tl) = f(MAP fn tl)tl
```

This theorem states the validity of general 'primitive recursive' definitions of functions over finitely-branching ordered trees.

11.9.9.2 The theory ltree

In the theory ltree, a type of *labelled* trees (called (*)ltree) is defined. Labelled trees have the same sort of structure as values of the defined type tree discussed above. The only difference is that a tree of type (*)ltree has a value or 'label' of type * associated with each of its nodes. A constructor

```
Node : * -> (*)ltree -> (*)ltree
```

is defined in the theory ltree. The function Node constructs labelled trees by mapping a label of type * and a list of labelled subtrees to a labelled tree of type (*)ltree. The following theorems about labelled trees are pre-proved and stored in the theory ltree.

```
Node_11       |- !v1 v2 trl1 trl2.
                 (Node v1 trl1 = Node v2 trl2) = (v1 = v2) /\ (trl1 = trl2)
ltree_Induct  |- !P. (!t. EVERY P t ==> (!h. P(Node h t))) ==> (!l. P l)
ltree_Axiom   |- !f. ?! fn. !v tl. fn(Node v tl) = f(MAP fn tl)v tl
```

These theorems are analogous to their counterparts in the theory tree discussed above. The theorems Node_11 and ltree_Induct amount to a Peano-type characterization of labelled trees, and the theorem ltree_Axiom is a primitive recursion theorem for labelled trees.

11.9.10 The theory tydefs

The theory `tydefs` is a technical theory used to support the type definition package described in Section 11.10. The main result proved in `tydefs` is a theorem called `TY_DEF_THM`, which is used by the type definition package to derive automatically abstract characterizations for arbitrary concrete recursive types. The interested reader can find the details of the derivation of `TY_DEF_THM` in [14].

11.9.11 The theory sum

The theory `sum` defines the binary disjoint union type operator `sum`. A type (σ_1,σ_2)`sum` denotes the disjoint union of types σ_1 and σ_2. The type operator `sum` can be defined just as `prod` was, but the details are omitted here.[12] The HOL parser converts "$:\sigma_1+\sigma_2$" into (σ_1,σ_2)`sum`, and the printer inverts this.

The standard operations on sums are:

```
INL  : *      -> * + **
INR  : **     -> * + **
ISL  : * + ** -> bool
ISR  : * + ** -> bool
OUTL : * + ** -> *
OUTR : * + ** -> **
```

These are all defined as constants in the theory `sum`. The constants `INL` and `INR` inject into the left and right summands, respectively. The constants `ISL` and `ISR` test for membership of the left and right summands, respectively. The constants `OUTL` and `OUTR` project from a sum to the left and right summands, respectively.

The following two theorems, which are minor variants of each other, are pre-proved in the built-in theory `sum`. Each one, on its own, provides a complete and abstract characterization of the disjoint sum type.

```
sum_axiom   |- !f g. ?! h. (h o INL = f) /\ (h o INR = g)

sum_Axiom = |- !f g. ?! h. (!x. h(INL x) = f x) /\ (!x. h(INR x) = g x)
```

Also provided as built-in, are the following theorems having to do with the discriminator functions `ISL` and `ISR`:

```
ISL         |- (!x. ISL(INL x)) /\ (!y. ~ISL(INR y))

ISR         |- (!x. ISR(INR x)) /\ (!y. ~ISR(INL y))

ISL_OR_ISR  |- !x. ISL x \/ ISR x
```

[12] The definition of disjoint unions in the HOL system is due to Tom Melham. The technical details of this definition can be found in [14].

The sum theory also provides the following built-in theorems:

```
OUTL        |- !x. OUTL(INL x) = x

OUTR        |- !x. OUTR(INR x) = x

INL         |- !x. ISL x ==> (INL(OUTL x) = x)

INR         |- !x. ISR x ==> (INR(OUTR x) = x)
```

which describe the projection functions OUTL and OUTR.

11.9.12 The theory one

The theory one defines the type ":one" which contains one element. The constant one is specified to denote this element. The pre-proved theorems in the theory one are:

```
one_axiom   |- !(f:* -> one) (g:* -> one). f = g
one         |- !(v:one). v = one
one_Axiom   |- !(e:*). ?!(fn:one->*). fn one = e
```

These three theorems are equivalent characterizations of the type with only one value.

11.10 The type definition package

In the HOL system, new types and type operators can be introduced using the consistency-preserving definitional mechanism of type definitions (see Sections 10.5.4 and 11.8.2.3). The ML rule for introducing a new type is:

```
new_type_definition : (string # term # thm) -> thm
```

This rule allows axioms of a restricted form to be added to the primitive basis of the logic. These axioms are analogous to definitional axioms for new constants: they define new types in terms of other type expressions already present in the logic. Like the rule new_definition for making constant definitions, the rule new_type_definition for type definitions ensures that adding a new syntactic entity (in this case, a type or type operator) is a conservative extension of the logic.

The basic idea behind new_type_definition is that a type definition is made by adding an axiom to the logic which asserts that the set of values denoted by a new type is isomorphic to an appropriate subset of the values denoted by a type expression already present in the logic. A definitional axiom of this form merely states that a new type is isomorphic to a particular subset of an existing type. From such type definition axioms, it is usual to prove theorems that characterize newly-defined types more abstractly. The

idea is to prove a collection of theorems that state the essential properties of a new type without reference to how it is defined. These theorems then constitute a derived 'abstract axiomatization' of the new type, and once they have been proved they become the basis for all further reasoning about it.

With this approach, introducing a new type (or type operator) in HOL involves two distinct steps:

1. Finding an appropriate representation for the new type, and making a type definition using `new_type_definition` based on this representation.
2. Using the axiomatic definition of the new type and the properties of its representation to prove a set of theorems that abstractly characterizes it.

Defining a new type using this approach can be hard work. But a set of tools is provided in the system which—for a certain class of commonly-used *concrete recursive types*—automatically carries out all the formal proofs necessary to define these types and derive abstract characterizations from their definitions. This section provides a user-level overview of these tools. Details of the formal proofs carried out by these tools are discussed in [14].

11.10.1 Defining types

The main ML function in the HOL type definition package is

```
define_type : string -> string -> thm
```

This function can be used to define any concrete recursive type in the HOL system. These are types whose values are generated by a set of *constructors* (i.e. functions) which yield concrete representations for these values. Examples include types which denote finite sets of atomic values (enumerated types), types which denote sets of structured values (record types) or finite disjoint unions of structured values (variant records), and types which denote sets of recursive data structures (recursive types).

The two inputs to `define_type` are both strings. The first string is a name under which the results of making the type definition will be stored in the current theory segment. The second is a user-supplied informal[13] specification of the concrete recursive type to be defined. This type specification is written in a notation (explained below) which resembles a data type declaration in functional programming languages like Standard ML [11]. It simply states the names of the new type's constructors and the logical types of their arguments. The output is a theorem which abstractly characterizes the properties of the desired recursive type—i.e. a derived 'abstract axiomatization' of the type.

[13] In this context, *informal* means not in the language of higher order logic.

11.10.1.1 Input syntax

The type specification given as input to `define_type` must be an ML string (of ML type `string`) of the form:

$$\text{`}op\ \texttt{=}\ C_1\ ty_1^1\ \ldots\ ty_1^{k_1}\quad \texttt{|}\ \cdots\ \texttt{|}\ C_m\ ty_m^1\ \ldots\ ty_m^{k_m}\text{`}$$

where each ty_i^j is either a type expression already defined as a type in the current theory (this type expression must not contain *op*) or is the name *op* itself. A string of this form describes an n-ary type operator *op*, where n is the number of distinct type variables in the types ty_i^j on the right hand side of the equation. If $n = 0$ then *op* is a type constant; otherwise *op* is an n-ary type operator. The concrete type described has m distinct constructors $C_1, \ldots, C_m$ where $m \geq 1$. Each constructor C_i takes k_i arguments, where $k_i \geq 0$; and the types of these arguments are given by the type expressions ty_i^j for $1 \leq j \leq k_i$. If one or more of the type expressions ty_i^j is the type *op* itself, then the equation specifies a *recursive* type. In any specification of a recursive type, at least one constructor must be non-recursive—i.e. all its arguments must have types which already exist in the current theory.

The input parser for `define_type` treats type expressions exactly as the HOL quotation parser does, with precedences among the various built-in type operators in force.

11.10.1.2 The type specified

The logical type described by an input string of the form shown above is intended to denote the set of all values which can be finitely generated using the constructors $C_1, \ldots, C_m$, where each constructor is one-to-one and any two different constructors yield different values. Every value of this type will be denoted by some term of the form:

$$C_i\ x_i^1\ \ldots\ x_i^{k_i}$$

where x_i^j is a term of type ty_i^j for $1 \leq j \leq k_i$. In addition, any two terms:

$$C_i\ x_i^1\ \ldots\ x_i^{k_i} \qquad \text{and} \qquad C_j\ x_j^1\ \ldots\ x_j^{k_j}$$

denote equal values exactly when their constructors are the same (i.e. $i = j$) and these constructors are applied to equal arguments (i.e. $x_i^n = x_j^n$ for $1 \leq n \leq k_i$).

11.10.1.3 The output

For any type specification in the form of an equation of the kind discussed above, executing:

$$\texttt{define_type}\ \text{`}name\text{`}\ \text{`}op\ \texttt{=}\ C_1\ ty_1^1\ \ldots\ ty_1^{k_1}\quad \texttt{|}\ \cdots\ \texttt{|}\ C_m\ ty_m^1\ \ldots\ ty_m^{k_m}\text{`}$$

will make a formal definition for a type (or type operator) *op* in the current theory segment, make appropriate definitions for constants $C_1, C_2, \ldots, C_m$, and automatically prove a theorem which provides an abstract characterization of the newly-defined type *op*. This theorem, which is stored in the current theory segment under the name *name* and also returned by `define_type`, has the form shown below:

$$\begin{array}{l}\texttt{|- !f}_1 \cdots \texttt{f}_m\texttt{. ?!fn:}op\texttt{->*.} \\ \quad \texttt{!x}_1^1 \cdots \texttt{x}_1^{k_1}\texttt{. fn(}C_1\ \texttt{x}_1^1 \ldots \texttt{x}_1^{k_1}\texttt{) = f}_1\ \texttt{(fn x}_1^1\texttt{)} \ldots \texttt{(fn x}_1^{k_1}\texttt{)}\ \texttt{x}_1^1 \ldots \texttt{x}_1^{k_1} \\ \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \vdots \\ \quad \texttt{!x}_m^1 \cdots \texttt{x}_1^{k_m}\texttt{. fn(}C_m\ \texttt{x}_m^1 \ldots \texttt{x}_m^{k_m}\texttt{) = f}_m\ \texttt{(fn x}_m^1\texttt{)} \ldots \texttt{(fn x}_m^{k_m}\texttt{)}\ \texttt{x}_m^1 \ldots \texttt{x}_m^{k_m}\end{array}$$

where the right hand sides of the equations include recursive applications '`fn` $\texttt{x}_i^j$' only for variables $\texttt{x}_i^j$ of type *op*. (See the examples given below.) A theorem of this form asserts the unique existence of primitive recursive functions defined by cases on the constructors $C_1, C_2, \ldots, C_m$. This is a slight extension of the *initiality* property by which structures of this kind are characterized in the 'initial algebra' approach to specifying abstract data types [6]. This property provides an abstract characterization of the type *op* which is both succinct and complete, in the sense that it completely determines the structure of the values of *op* up to isomorphism.

The call to `define_type` shown above fails if:

(i) not in draft mode;

(ii) *op* is already the name of a type constant or type operator in the current theory;

(iii) any one of $C_1, \ldots, C_m$ is already the name of a constant in the current theory.

(iv) either *op* or any one $C_1, \ldots, C_m$ is not a legal identifier. Identifiers must start with a letter (as defined by `is_letter`) and contain only alphanumeric characters (as defined by `is_alphanum`)

(v) `ABS_`*op* or `REP_`*op* are already constants in the current theory;

(vi) there is already an axiom, definition, constant specification or type definition stored under either the name *op*`_TY_DEF` or the name *op*`_ISO_DEF` in the current theory segment.

(vii) there is already a theorem stored under the name '*name*' in the current theory segment.

(viii) the input type specification does not conform to the syntax described above.

11.10.1.4 Examples

The session that follows illustrates the use of `define_type` in defining a variety of simple concrete types. It is assumed that the session begins with the user in draft mode.

The first definition is simple, the definition of a type `three` with exactly three distinct values: `ONE`, `TWO`, and `THREE`.

```
#let three_Axiom = define_type `three_Axiom` `three = ONE | TWO | THREE`;;  1
three_Axiom =
|- !e0 e1 e2. ?! fn. (fn ONE = e0) /\ (fn TWO = e1) /\ (fn THREE = e2)
```

The theorem returned by `define_type` provides a complete and abstract characterization of a defined logical type `three` which denotes a set of exactly three elements. This characterization takes the form of a degenerate 'primitive recursion' theorem for the concrete type `three`. Since `three` is an enumerated type with no recursive constructors, the theorem returned by `define_type` simply states that any function defined by cases on the three constants `ONE`, `TWO`, and `THREE` exists and is uniquely defined.

It follows immediately from this theorem that the type constant `three` denotes a set containing exactly three values: the fact that the function `fn` always exists implies that the constants `ONE`, `TWO`, and `THREE` denote distinct values of type `three`, and the fact that `fn` is uniquely determined by its values for `ONE`, `TWO`, and `THREE` implies that these constants denote the only values of type `three`.

The next call to `define_type` defines a 'record type' `rec`, values of which are records with three boolean fields (essentially 3-tuples):

```
#let rec_Axiom = define_type `rec_Axiom` `rec = REC bool bool bool`;;  2
rec_Axiom = |- !f. ?! fn. !b0 b1 b2. fn(REC b0 b1 b2) = f b0 b1 b2
```

Here, the resulting theorem states that a function `fn` on record values of type `rec` can be uniquely defined in terms of a function `f` of the three components of the record.

A more interesting *recursive* example is the type of natural numbers , which can be defined using `define_type` as follows:

```
#let nat_Axiom = define_type `nat_Axiom` `nat = Z | Suc nat`;;  3
nat_Axiom = |- !e f. ?! fn. (fn Z = e) /\ (!n. fn(Suc n) = f(fn n)n)
```

Here, the input string describes a type `nat` with two constructors: `Z`, which stands for zero; and `Suc`, which is the successor function on natural numbers. (The names `Z`, and `Suc` are used here because `0` and `SUC` are already constants in the built-in HOL theory `num`.) The output theorem is just the primitive recursion theorem[14] for the natural numbers; it states that any primitive recursive definition on the natural numbers (i.e. on values of type `nat`) uniquely defines a total function.

[14]See Section 11.9.5 for a discussion of the primitive recursion theorem.

A recursive type of labelled binary trees, where labels of type * appear only on leaf nodes, can likewise be defined using `define_type`. The input states that a binary tree is either a leaf node (`LEAF`) labelled by a value of type * or an internal node `NODE` with two binary trees as subtrees:

```
#let btree_Axiom =                                                    4
#    define_type `btree_Axiom` `btree = LEAF * | NODE btree btree`;;
btree_Axiom =
|- !f0 f1.
    ?! fn.
     (!x. fn(LEAF x) = f0 x) /\
     (!b1 b2. fn(NODE b1 b2) = f1(fn b1)(fn b2)b1 b2)
```

The result returned by the call to `define_type` is, in this case, an abstract characterization for a defined type `(*)btree`, in the form of a 'primitive recursion theorem' for the required type of labelled binary trees.

Any simple concrete recursive type can be defined automatically from a user-supplied equation using `define_type` in exactly the same way.

11.10.2 Defining recursive functions

An important property of the characterizing theorems for concrete types shown in the examples given above is that they provide a formal means for defining recursive functions on those types. When a concrete recursive type *op* is characterized by a theorem of the kind returned by `define_type` (see Section 11.10.1.3) this theorem can be used to prove the existence of any *primitive recursive* function on *op* and to define constants which denote such functions.

This is illustrated for a particular example by the method of defining primitive recursive functions on the natural numbers discussed in Section 11.9.5.1. In that section, an ML function `new_prim_rec_definition` was described which automates the logical inferences necessary to derive particular primitive recursive definitions on the built-in defined type `num` of natural numbers. The basis of this function is the primitive recursion theorem

```
num_Axiom   |- !x f. ?!fn. (fn 0 = x) /\ (!n. fn(SUC n) = f (fn n) n)
```

which is pre-proved and stored in the built-in theory `prim_rec` (see Section 11.9.5). The ML function `new_prim_rec_definition` uses `num_Axiom` to automate the justification of any user-supplied primitive recursive definition on the natural numbers.

The type definition package provides a similar function for defining primitive recursive functions on arbitrary concrete recursive types.[15] The ML function

[15] In fact, `new_prim_rec_definition` is defined in ML using the more general tools provided by the type definition package.

```
new_recursive_definition : bool -> thm -> string -> term -> thm
```

automates the inferences necessary to justify any given primitive recursive definition on a concrete recursive type of the kind definable by `define_type`. It takes four arguments. The first is a boolean flag which indicates if the function to be defined will be an infix or not. The second is the primitive recursion theorem for the concrete type in question (i.e. a theorem obtained from `define_type`). The third argument is a name under which the resulting definition will be saved in the current theory segment. The fourth argument is a term giving the desired primitive recursive definition. The value returned by `new_recursive_definition` is a theorem which states the primitive recursive definition requested by the user. This theorem is derived by formal proof from an instance of the general primitive recursion theorem given as the second argument.

If the ML variable *op*`_Axiom` is bound to a theorem of the form returned by `define_type`, then evaluating:

```
new_recursive_definition
  `flag` op_Axiom `name` " primitive recursive definition on op"
```

automatically proves the existence of the primitive recursive function supplied as the fourth argument, and then declares a new constant in the current theory with this definition as its specification. This constant specification is returned as a theorem and is saved in the current theory segment under the name *name*. If *flag* is `true`, the constant is given infix status. Failure occurs if:

(i) HOL cannot prove there is a function satisfying the defining equations supplied by the user (i.e. the term supplied to `new_recursive_definition` is not a well-formed primitive recursive definition on values of type *op*);

(ii) any other condition for making a constant specification is violated (see the failure conditions for `new_specification` in Section 11.8.2.2).

Curried functions defined using `new_recursive_definition` can be recursive on any one of their arguments. Furthermore, defining equations need not be given for all the constructors of the concrete type in question. See the examples given in the next section, or the examples of functions defined on `num` given in Section 11.9.5.1 for more details.

The ML function

```
prove_rec_fn_exists : thm -> term -> thm
```

is a version of `new_recursive_definition` which proves only that the required function exists; it does not make a constant specification. The first argument is a theorem of the form returned by `define_type`, and the second is a user-supplied primitive recursive function definition. The theorem which is returned asserts the existence of the recursively-defined function in question (if it is primitive recursive over the type characterized by the theorem given as the first argument).

11.10.2.1 More examples

Continuing the example session started above in Section 11.10.1.4, the following interactions with the system show how the ML function `new_recursive_definition` can be used to define functions on concrete types, which have themselves been defined using `define_type`.

Given the characterizing theorem `btree_Axiom` for the type of labelled binary trees defined in Section 11.10.1.4, a recursive function `Leaves`, which computes the number of leaf nodes in a binary tree, can be defined recursively in HOL as shown below:

5
```
#let Leaves =
#    new_recursive_definition false btree_Axiom `Leaves`
#      "(Leaves (LEAF (x:*)) = 1) /\
#       (Leaves (NODE t1 t2) = (Leaves t1) + (Leaves t2))";;
Leaves =
|- (!x. Leaves(LEAF x) = 1) /\
   (!t1 t2. Leaves(NODE t1 t2) = (Leaves t1) + (Leaves t2))
```

The result of the call to `new_recursive_definition` is a theorem which states that the constant `Leaves` satisfies the primitive-recursive defining equations supplied by the user. This theorem is derived automatically from an instance of the general primitive recursion theorem for binary trees (`btree_Axiom`) and an appropriate constant specification for the constant `Leaves`.

The function defined using `new_recursive_definition` need not, in fact, be recursive. Here is the definition of a predicate `IsLeaf`, which is true of binary trees which are leaves, but is false of the internal nodes in a binary tree:

6
```
#let IsLeaf =
#    new_recursive_definition false btree_Axiom `IsLeaf`
#      "(IsLeaf (NODE t1 t2) = F) /\ (IsLeaf (LEAF (x:*)) = T)";;
IsLeaf = |- (!t1 t2. IsLeaf(NODE t1 t2) = F) /\ (!x. IsLeaf(LEAF x) = T)
```

Note that two equations defining a (recursive or non-recursive) function on binary trees by cases can be given in either order. Here, the `NODE` case is given first, and the `LEAF` case second. The reverse order was used in the above definition of `Leaves`.

The ML function `new_recursive_definition` also allows the user to partially specify the value of a function defined on a concrete type, by allowing defining equations for some of the constructors to be omitted. Here, for example, is the definition of a function `Label` which extracts the label from a leaf node. The value of `Label` applied to an internal node is left unspecified:

7
```
#let Label =
#    new_recursive_definition false btree_Axiom `Label`
#      "Label (LEAF (x:*)) = x";;
Label = |- !x. Label(LEAF x) = x
```

Curried functions can also be defined, and the recursion can be on any argument. The next definition defines an infix (curried) function `<<` which expresses the idea that one tree is a proper subtree of another.

8
```
#let Subtree =
#    new_recursive_definition true btree_Axiom `Subtree`
#      "(<< (t:(*)btree) (LEAF (x:*)) = F) /\
#       (<< t (NODE t1 t2) = ((t=t1) \/ (t=t2) \/ (<< t t1) \/ (<< t t2)))";;
Subtree =
|- (!t x. t << (LEAF x) = F) /\
   (!t t1 t2.
     t << (NODE t1 t2) = (t = t1) \/ (t = t2) \/ t << t1 \/ t << t2)
```

Note that the first argument to the ML function is `true` (to indicate that the function being defined is to have infix status) and that the constant `<<` is an infix after the definition has been made. Furthermore, the function `<<` is recursive on its second argument.

Finally, the function `new_recursive_definition` can also be used to define functions by cases on enumerated types. For example, a predicate `One`, which is true of only the value `ONE` of the three-valued type `three` defined above in Section 11.10.1.4, can be defined as follows:

9
```
#let One = new_recursive_definition false three_Axiom `One`
#            "(One ONE = T) /\ (One TWO = F) /\ (One THREE = F)";;
One = |- (One ONE = T) /\ (One TWO = F) /\ (One THREE = F)
```

The existence only of any function definable using `new_recursive_definition` can be proved using `prove_rec_fn_exists`. For example:

10
```
#close_theory();;
() : void

#let exists = prove_rec_fn_exists three_Axiom
#             "(f ONE = T) /\ (f TWO = F) /\ (f THREE = F)";;
exists = |- ?f. (f ONE = T) /\ (f TWO = F) /\ (f THREE = F)
```

The resulting theorem simply states the existence of the required function. Here, a constant is not defined, and the user need not be in draft mode.

11.10.3 Structural induction

For any concrete recursive type definable using the HOL type definition package there is a structural induction theorem which states the validity of proof by induction on the structure of the type's values. The ML function

```
prove_induction_thm : thm -> thm
```

can be used to derive a structural induction theorem for any concrete recursive type defined using `define_type`. If the ML variable *op*`_Axiom` is bound to a theorem of the form returned by `define_type`, then executing

```
prove_induction_thm op_Axiom
```

will prove and return a structural induction theorem for the concrete type *op*. The 'induction' theorem is degenerate in the case of non-recursive types (see the examples given below). Failure occurs, or an unpredictable output theorem is returned, if the input theorem does not have the form of a theorem returned by `define_type`.

11.10.3.1 Examples

A structural induction theorem on the type of binary trees defined in the session beginning on Section 11.10.1.4 can be proved by:

11
```
#let btree_Induct = prove_induction_thm btree_Axiom;;
btree_Induct =
|- !P.
    (!x. P(LEAF x)) /\ (!b1 b2. P b1 /\ P b2 ==> P(NODE b1 b2)) ==>
    (!b. P b)
```

The output theorem states that a predicate `P` is true of all binary trees if it is true of all labelled leaf nodes, and whenever it is true of two binary trees `b1` and `b2` it is also true of the binary tree `NODE b1 b2`, in which `b1` and `b2` occur as immediate left and right subtrees.

For non-recursive types, the induction theorem returned by `prove_induction_thm` is degenerate: there are no 'step' cases in the induction. For the two types `three` and `rec` defined in the preceding interactions of this session, the induction theorems are:

12
```
#let three_Induct = prove_induction_thm three_Axiom;;
three_Induct = |- !P. P ONE /\ P TWO /\ P THREE ==> (!t. P t)

#let rec_Induct = prove_induction_thm rec_Axiom;;
rec_Induct = |- !P. (!b0 b1 b2. P(REC b0 b1 b2)) ==> (!r. P r)
```

Here, induction simply reduces to the consideration of cases, one for each of the constructors for the concrete type involved.

11.10.4 Structural induction tactics

This section has been included here for reference because it relates chiefly to the type definition package, but it involves concepts not defined until later, in Chapter 14. Tactics, goals and subgoals are defined in Section 14.1; and theorem continuations, in Section 14.5.

`MAP_EVERY` is defined in Section 14.5.1. `ASSUME_TAC` is defined in Section 14.3. `MP_TAC` and `INDUCT_TAC` can be found in *REFERENCE*.

The ML function

```
INDUCT_THEN : thm -> (thm -> tactic) -> tactic
```

can be used to generate a structural induction tactic for any concrete types definable using `define_type`. The first argument is an induction theorem of the form returned by the function `prove_induction_thm` discussed in the previous section. The second argument is a theorem continuation (see Chapter 14) that determines what is to be done with the induction hypotheses when the resulting tactic is applied to a goal.

If th is an induction theorem for a concrete type *op* with m constructors C_1, ..., C_m (i.e. a theorem of the kind returned by `prove_induction_thm`) and F is a theorem continuation, then the tactic `INDUCT_THEN` th F will reduce a goal (Γ,"!x:*op*.$t[x]$") to the collection of m induction subgoals generated by:

`MAP_EVERY` F `[`th_1^1`;` ... `;` $th_1^{k_1}$`]` (Γ, "$t[C_1\ x_1^1\ \ldots\ x_1^{k_1}]$"),
⋮
`MAP_EVERY` F `[`th_m^1`;` ... `;` $th_m^{k_m}$`]` (Γ, "$t[C_m\ x_m^1\ \ldots\ x_m^{k_m}]$")

where th_i^j is a theorem of the form `|-` $t[x_i^j]$ asserting the truth of $t[x_i^j]$ for the jth recursive argument (for non-recursive arguments, there will be no th_i^j in the list) of the ith constructor C_i (for $1 \leq i \leq m$).

The most common use of `INDUCT_THEN` is in conjunction with the theorem continuation `ASSUME_TAC`. For example, the built-in induction tactic `INDUCT_TAC` for mathematical induction on the natural numbers is defined in ML by:

```
let INDUCT_TAC = INDUCT_THEN INDUCTION ASSUME_TAC
```

This built-in tactic reduces a goal (Γ,"!n.$t[n]$") to a basis subgoal (Γ,"t[`0`]") and a step subgoal (Γ ∪ {"$t[n]$"},"t[`SUC` n]"). The extra assumption "$t[n]$" (i.e. the induction hypothesis) is added to the assumptions Γ by `ASSUME_TAC`.

By contrast, the induction tactic `INDUCT_MP_TAC` (which is not built-in) defined by:

```
let INDUCT_MP_TAC = INDUCT_THEN INDUCTION MP_TAC
```

reduces a goal (Γ,"!n.$t[n]$") to a basis subgoal (Γ,"t[`0`]") and an induction step subgoal (Γ, "$t[n]$ `==>` t[`SUC` n]"). Here, the theorem continuation `MP_TAC` makes the induction hypothesis an antecedent of the step subgoal, rather than an assumption.

As this example illustrates, the theorem continuation F in an induction tactic

`INDUCT_THEN` th F

generated using an induction theorem th can be thought of as a function which determines what is to be done with the induction hypotheses corresponding to the recursive arguments of constructors in the step cases of a proof by structural induction. When F is `ASSUME_TAC`, the induction hypotheses become assumptions in the subgoals generated; and when F is `MP_TAC`, the induction hypotheses become the antecedents of implicative subgoals. Other theorem continuations (for which, see Chapter 14 and *REFERENCE*) can also be used.

11.10.5 Other tools

The function

```
prove_constructors_one_one : thm -> thm
```

proves that the constructors of a concrete type which take arguments are one-to-one. The argument to `prove_constructors_one_one` is a theorem of the form returned by `define_type`.

The function

```
prove_constructors_distinct : thm -> thm
```

proves that the constructors of a concrete type yield distinct values. The argument to `prove_constructors_distinct` is again a theorem of the form returned by `define_type`.

The function

```
prove_cases_thm : thm -> thm
```

proves a cases theorem for any concrete type. Such a theorem states that every value can be constructed using one of the type's constructors. This property follows more easily (and therefore is faster to prove) from induction than from primitive recursion, so the function `prove_cases_thm` takes as an argument an induction theorem of the kind returned by `prove_induction_thm`.

These auxiliary tools work for any concrete type definable using `define_type`.

11.10.5.1 Examples

The following interactions with the system show the proof that the constructor `LEAF` for the type `(*)btree` is one-one, and also that the constructor `REC` for the type `rec` is one-to-one.

```
#let LEAF_one_one = prove_constructors_one_one btree_Axiom;;                    13
LEAF_one_one =
|- (!x x'. (LEAF x = LEAF x') = (x = x')) /\
   (!b1 b2 b1' b2'.
     (NODE b1 b2 = NODE b1' b2') = (b1 = b1') /\ (b2 = b2'))

#let REC_one_one = prove_constructors_one_one rec_Axiom;;
REC_one_one =
|- !b0 b1 b2 b0' b1' b2'.
    (REC b0 b1 b2 = REC b0' b1' b2') =
    (b0 = b0') /\ (b1 = b1') /\ (b2 = b2')
```

The function `prove_constructors_one_one` fails when the concrete type involved has no constructors that take arguments. For example:

```
#let th = prove_constructors_one_one three_Axiom;;                              14
evaluation failed     prove_constructors_one_one: invalid input theorem
```

The function `prove_constructors_distinct` returns the theorem stating that the constructors of a concrete type yield pair-wise distinct values. For example:

```
#let NOT_LEAF_NODE = prove_constructors_distinct btree_Axiom;;                  15
NOT_LEAF_NODE = |- !x b1 b2. ~(LEAF x = NODE b1 b2)

#let three_distinct = prove_constructors_distinct three_Axiom;;
three_distinct = |- ~(ONE = TWO) /\ ~(ONE = THREE) /\ ~(TWO = THREE)
```

Cases theorems are proved from structural induction theorems. For the binary tree example considered in the present session, here is the cases theorem:

```
#let btree_cases = prove_cases_thm btree_Induct;;                               16
btree_cases = |- !b. (?x. b = LEAF x) \/ (?b1 b2. b = NODE b1 b2)
```

Note that the structural induction theorem for binary trees, `btree_Induct`, is used.

11.11 Miscellaneous system features

This section describes some of the features that exist for managing the interface to the HOL system. (Other system functions can be found in Chapter 8.)

- Flags for controlling the parsing and printing of terms.
- A mechanism called *sticky types* allows default types of variables to be established.
- A facility called *interface maps* permits a limited amount of overloading to be achieved. Interface maps also support the use of conventional logical characters on bit-mapped screens.

- Functions for changing the ASCII versions to λ and $\vdash$ inside quotations (the defaults are \ and |- respectively).

- A method for hiding constants from the quotation parser, so that variables with the same names as constants can be input by quotation.

- A method for arranging for definitions and theorems (and so on) to be loaded automatically when their names are encountered by the ML parser.

- Functions to control whether entire theories are kept in memory ('cached') or not.

- A function for adjusting the maximum depth to which terms and theorems are printed by the pretty printer (the default is 500).

- Functions for counting the number of primitive inferences done in an evaluation, and timing it.

11.11.1 Flags for the HOL logic

See also Chapter 8 for a description of the flag mechanism which is used to control the state of the HOL system. The subset of flags that control aspects of HOL relating to the logic is summarized in the table below.

Settable system flags

Flag	*Function*	*Default value*
`timing`	Print number of theorems proved	`false`
`show_types`	Prints types in quotations	`false`
`sticky`	Activates sticky types	`false`
`theory_pp`	Pretty printing of theory files	`false`
`type_error`	Verbose type checking errors in quotations	`true`
`interface_print`	Causes inverse of interface map to be used when printing	`true`
`print_cond`	Pretty print HOL conditionals	`true`
`print_quant`	Pretty print HOL quantifications	`true`
`print_let`	Pretty print HOL `let`-expressions	`true`
`print_list`	Pretty print HOL lists	`true`
`print_uncurry`	Pretty print HOL paired abstractions	`true`
`print_infix`	Pretty print HOL infixes	`true`

11.11.2 Sticky types and default types

The quotation type checker normally requires that there be enough information in a quotation to fully determine the types of all subterms of the term being type checked. This information comes from two sources:

(i) the generic types of constants, and

(ii) explicit type information given after a colon.

If sticky types are activated (by setting the flag `sticky` to `true`) then the last type a variable had in a session is remembered, and this is used as a default type during type checking. The sticky type is only used, however, if the variable in question is completely unconstrained by its context in the quotation. If the context partially determines the type of a variable, then sticky types are not used. For example, in the term `"f 1"` the variable `f` is constrained to have a function type, but the range type is not determined. In the quotation `"(f,1)"` the variable `f` is completely unconstrained. In the latter case an (appropriate) sticky type would resolve the typing, but in the former it would not. Sticky types can never cause the type checking of a quotation to fail when it would succeed without them.

1

```
#set_flag(‘sticky‘,true);;
false : bool

#"f:num->num";;
"f" : term

#"f 1";;
Indeterminate types:  "f:num -> ?"

evaluation failed     types indeterminate in quotation

#"f";;
"f" : term
```

A name can be given a sticky type explicity with the function:

```
set_sticky_type : string # type -> void
```

This sets sticky types whether or not sticky types are active.

Note that the flag `sticky` controls whether types are automatically stuck onto names; it does not control whether such sticky types are actually used during type checking: sticky types are always used if they are defined.

Sticky types can be removed with the function:

```
remove_sticky_type : string -> type
```

The sticky type is returned (failure if the variable does not have a sticky type). A function to return the sticky type of a variable can be defined by:

```
let sticky_type v =
 let ty = remove_sticky_type v
 in
 (set_sticky_type(v,ty);ty)
```

The list of sticky types active in a session is computed by the function:

```
sticky_list : void -> (string # type) list
```

11.11.3 Interface maps

An interface map is a list of pairs of strings:

[‘a_1‘,‘c_1‘; ... ;‘a_n‘,‘c_n‘]

If such a map is active, then whenever a_i (for $1 \leq i \leq n$) occurs as a constant or variable in a quoted term, it is translated (by the quotation parser) to c_i, which must be an existing constant (‘a’ is for ‘abbreviation’ and ‘c’ for ‘constant’).

The function:

```
set_interface_map : string list -> string list
```

installs an interface map (the previous map is undone and returned). The effect of executing:

set_interface_map [‘a_1‘,‘c_1‘; ... ;‘a_n‘,‘c_n‘];;

is to install [‘a_1‘,‘c_1‘; ... ;‘a_n‘,‘c_n‘] as the current interface map. Any binder or infix status (including precedences) of c_i is propagated to a_i in the order:

$$status(c_1) \rightsquigarrow a_1 \text{ then } status(c_2) \rightsquigarrow a_2 \text{ then } \ldots \text{ then } status(c_n) \rightsquigarrow a_n$$

This order can be important. For example, if ix is the name of an infixed constant and c the name of a non-infixed constant, then executing:

set_interface_map[‘c‘,‘ix‘; ‘ix‘,‘c‘];;

will result in both c and ix being infixes; but executing:

set_interface_map[‘ix‘,‘c‘; ‘c‘,‘ix‘];;

results in neither c nor ix being an infix:

The current interface map is given by:

```
interface_map : void -> string list
```

To avoid parsing and printing ambiguities, maps must be single valued (i.e. one abbreviation cannot abbreviate two constants) and one-to-one (i.e. different abbreviations must map to different constants). Furthermore, if a_i is already a constant then it must be given an abbreviation by the map. For example:

```
set_interface_map[`+`,`APPEND`];;
```

is not allowed because it would make it unclear how to print pre-existing terms involving `+`. To avoid this problem, an abbreviation for the old `+` must also be provided.

The flag `interface_print` determines whether the inverse of the current interface map is applied when printing. This should be useful for debugging as it shows what is 'really there'.

The following functions are also provided:

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

these tell whether a string is the name of a constant, infix or binder respectively (note: being an infix or binder entails being a constant).

The function

```
allowed_constant : string -> bool
```

tests whether a string has the correct lexical structure to be a constant.

The function:

```
draft_mode : void -> bool
```

returns `true` in draft mode[16] and `false` otherwise.

The following sessions illustrate the use of interface maps (the comments are provided by the author, not by the system). They also show the great potential of interface maps for generating confusion! The first session illustrates checking that interface maps are well formed.

[16] See Section 11.8 for an explanation of 'draft mode'.

```
#set_interface_map[`+`,`APPEND`];;                                    1
evaluation failed     + would become hidden

#set_interface_map[`+`,`APPEND`;`+`,`++`];;
evaluation failed     interface map not single valued

#set_interface_map[`+`,`APPEND`;`++`,`+`];;
[] : (string # string) list
```

The next session shows the propagation of infix status by the interface map declared above:

```
#"1 ++ 2";;                              % ++ is not an infix. %       2
Badly typed application of:  "1"
   which has type:           ":num"
to the argument term:        "++"
   which has type:           ":num -> (num -> num)"

evaluation failed     mk_comb in quotation

#"++ 1 2";;
"++ 1 2" : term

#set_interface_map[`++`,`+`; `+`,`APPEND`];;
[(`+`,`APPEND`); (`++`,`+`)] : (string # string) list

#"1 ++ 2";;
"1 ++ 2" : term
```

Setting the flag `interface_print` to `false` causes the true form of terms to be shown.

```
#set_flag(`interface_print`, false);;                                 3
true : bool

#"1 ++ 2";;
"1 + 2" : term

#dest_const "++";;           % This illustrates potential for confusion. %
`+`,":num -> (num -> num)" : (string # type)
```

Continuing:

```
#"[1]+[2]";;                  % APPEND is not an infix. %                    4
Badly typed application of:  "[1]"
   which has type:           ":(num)list"
to the argument term:        "APPEND"
   which has type:           ":(?1)list -> ((?1)list -> (?1)list)"

evaluation failed     mk_comb in quotation

#"+[1][2]";;
"APPEND[1][2]" : term

#"APPEND [1] [2]";;
"APPEND[1][2]" : term
```

Here is an example illustrating the kind of thing to avoid.

```
#ADD1;;                       % Built in theorem. %                          5
Theorem ADD1 autoloaded from theory `arithmetic`.
ADD1 = |- !m. SUC m = m + 1

|- !m. SUC m = m + 1

#set_flag(`interface_print`, true);;
false : bool

#ADD1;;
|- !m. SUC m = m ++ 1

#set_interface_map[`+`,`-`; `-`,`+`];;
[`++`,`+`; `+`,`APPEND`] : (string # string) list

#ADD1;;
|- !m. SUC m = m - 1
```

There are some subtleties and rough edges to the current implementation of interface maps. For example, any special parsing or printing properties of special constants (`!`, `?`, `/\`, `\/`, etc.) are transferred to the names that abbreviate them. For example, notice the disappearing spaces in the next session.

```
#set_interface_map[`Forall`,`!`; `Exists`,`?`];;                            1
[] : (string # string) list

#"Forall x. Exists y. x < y";;
"Forallx. Existsy. x < y" : term
```

This happens because the printer does not print spaces after `!` and `?` and this behaviour gets transferred to `Forall` and `Exists` by the interface map. It would be straightforward, but rather messy, to have the printer insert spaces after `Forall` and `Exists`. This will

be implemented, and other changes or additions, if experience indicates it is sensible. Interface maps are still rather experimental, and it is expected that various details will need modification as experience is gained. A feature not currently implemented, but possibly useful, would allow the relative precedences of user-defined infixes to be adjusted.

11.11.4 Changing the lambda symbol

The ML function:

```
set_lambda : string -> string
```

allows an alternative symbol (or string) to be used to represent lambda (λ) inside quotations. The previous λ-representing string is returned. Note that after calling `set_lambda`:

- The symbol \ is still available for input.
- All abstractions will be printed with the new string.
- It is necessary to use: `set_lambda '\\'` to restore the alternative lambda to \ (since \ is the escape character inside ML strings).

For example:

1
```
#set_lambda `Lambda`;;
`\` : string

#"Lambda x.x+1";;
"Lambda x. x + 1" : term

#"\x.x+1";;
"Lambda x. x + 1" : term
```

11.11.5 Changing the turnstile symbol

The turnstile symbol |-, which is used as an ASCII approximation to $\vdash$ when printing theorems, can be changed with the ML function:

```
set_turnstile : string -> string
```

The previous representation of turnstile is returned. For example:

```
#REFL "x:num";;                                                    1
|- x = x

#set_turnstile `Proved! `;;
`|- ` : string

#REFL "x:num";;
Proved! x = x

#set_turnstile `       |-  `;;
`Proved! ` : string

#REFL "x:num";;
       |-  x = x
```

11.11.6 Hiding constants

The following function can be used to hide the constant status of a name from the quotation parser.

```
   hide_constant : string -> void
```

Evaluating `hide_constant` `$x$` makes the quotation parser treat x as a variable (lexical rules permitting), even if x is the name of a constant in the current theory (constants and variables can have the same name). This is useful if one wants to use variables with the same names as previously declared (or built-in) constants (e.g. `o`, `I`, `S` etc.). The name x is still a constant for the constructors, theories, etc; `hide_constant` affects only parsing.

Hiding a constant and then attempting to declare it as a new constant will fail (as it must, if the system is to remain sound).

The function

```
   unhide_constant : string -> void
```

undoes the hiding; it fails if its argument is not a previously hidden constant.

The function:

```
   is_hidden : string -> bool
```

tests whether a string is the name of a hidden constant.

11.11.7 Autoloading of axioms, definitions and theorems

It is possible to mark the name of an axiom, definition or theorem so that if it occurs in an ML expression then the named item will be automatically loaded from the appropriate theory before the ML expression is evaluated. This provides a kind of crude 'paging' of theories. Many standard definitions and theorems are set up to autoload if they are used.

The function to set up an autoloading action is:

```
autoload_theory : (string # string # string) -> void
```

Evaluating `autoload_theory(`*`kind`*`,`*`thy`*`,`*`name`*`)`, where *kind* is `axiom`, `definition` or `theorem`, has the side effect of setting the name *name* so that when it is encountered by the ML parser the appropriate sort of object will be autoloaded from the theory *thy*. Note that constant specifications and type definitions are regarded as special cases of definitions.

An autoloading action set up with `autoload_theory` will only be done once, namely the first time the name is encountered in the session. Autoload actions can be removed from a name using the function:

```
undo_autoload : string -> bool
```

The value `true` is returned if an autoload action had been set up, `false` is returned otherwise.

11.11.8 Cached theories

When a theory is first accessed using the functions `load_theory`, `axiom`, `definition`, `theorem` etc., the whole theory is read from disk into a data-structure in memory called a *theory cache*. Subsequent accesses then refer to this in-memory cache and are therefore faster. Theories, however, can be very large, and on machines with limited memory, this can lead to a lot of garbage collection, or even running out of space. To deal with this problem, the ML function:

```
delete_cache : string -> void
```

can be used to delete the theory cache (but not the theory file on disk), thereby freeing up some space (at the cost of making the next theory access slower). For example, a function that fetches a single theorem without bringing in the whole theory is:

```
let fetch_thm_without_caching thy thm =
    let th = theorem thy thm
    in  (delete_cache thy; th);;
```

Whether a theory has been cached or not can be discovered with the function:

```
cached_theories : void -> (string # bool) list
```

This returns a list of pairs (*thy*,*b*) where *thy* is the name of a theory that has been loaded and *b* is a truth value that is `true` if *thy*'s cache has been deleted with `delete_cache` and `false` otherwise.

11.11.9 Adjusting the pretty-print depth

The following ML function can be used to adjust the maximum depth of printing.

```
max_print_depth : int -> int
```

The default print depth is 500. Evaluating `max_print_depth` n sets the maximum to n and returns the previous value of the maximum. Subterms nested more deeply than the maximum print depth are printed as `&`. For example:

```
#ADD_CLAUSES;;                                                          1
Theorem ADD_CLAUSES autoloaded from theory `arithmetic`.
ADD_CLAUSES =
|- (0 + m = m) /\
   (m + 0 = m) /\
   ((SUC m) + n = SUC(m + n)) /\
   (m + (SUC n) = SUC(m + n))

|- (0 + m = m) /\
   (m + 0 = m) /\
   ((SUC m) + n = SUC(m + n)) /\
   (m + (SUC n) = SUC(m + n))

#max_print_depth 7;;
500 : int

#ADD_CLAUSES;;
|- (& + & = m) /\ (& + & = m) /\ ((& + & = &(&) /\ (& + (& = &(&)

#max_print_depth 5;;
7 : int

#ADD_CLAUSES;;
|- (& /\ (& /\ (& /\ (&

#max_print_depth 3;;
5 : int

#ADD_CLAUSES;;
|- &
```

For more details about pretty printing in ML, see Section 8.2.1.

11.11.10 Timing and counting theorems

Whenever HOL performs a primitive inference (or accepts an axiom or definition) a counter is incremented. The value of this counter is returned by the function:

```
thm_count : void -> int
```

This counter can be reset with the function:

```
set_thm_count : int -> int
```

The previous value of the counter is returned.

The following function is used to switch ML into a mode in which the number of primitive inferences done during each top-level interaction is shown. Run-time and garbage collection time are also shown.

```
timer : bool -> bool
```

Executing `timer true` causes the number of primitive inferences and timings to be printed; `timer false` switches the printing off. The previous setting is returned. Executing `timer` *b* is equivalent to setting the flag `timing` to the value *b* (see Section 8.13).

Part IV

Theorem Proving with HOL

Chapter 12

Derived Inference Rules

The notion of *proof* was defined in the abstract in Chapter 9: a proof of a sequent (Γ, t) from a set of sequents Δ (with respect to a deductive system $\mathcal{D}$) was defined to be a chain of sequents culminating in (Γ, t), such that every element of the chain either belongs to Δ or else follows from Δ and earlier elements of the chain by deduction. The notion of a *theorem* was also defined in Chapter 9: a theorem of a deductive system is a sequent that follows from the empty set of sequents by deduction; i.e., it is the last element of a proof from the empty set of sequents, in the deductive system. In this section, proofs and theorems are made concrete in HOL.

The deductive system of HOL was sketched in Section 11.8.5, where the eight families of primitive inferences making up the deductive system were specified by diagrams. It was explained that these families of inferences are represented in HOL via ML functions, and that theorems are represented by an ML abstract type called `thm`. The eight ML functions corresponding to the inferences are operations of the type `thm`, and each of the eight returns a value of type `thm`. It was explained that the type `thm` has primitive destructors, but no primitive constructor; and that in that way, the logic is protected against the computation of theorems except by functions representing primitive inferences, or compositions of these.

Finally, the primitive HOL logic was supplemented by three primitive constants and five axioms, to form the basic logic[1]. The primitive inferences, together with the primitive constants, the five axioms, and a collection of definitions, give a starting point for constructing proofs, and hence computing theorems. However, proving even the simplest theorems from this minimal basis costs considerable effort. The basis does not immediately provide the transitivity of equality, for example, or a means of universal quantification; both of these themselves have to be derived.

12.1 Simple derivations

As an illustration of a proof in HOL the following chain of theorems forms a proof (from the empty set, in the HOL deductive system), for the particular terms "t_1" and "t_2", both of HOL type "`:bool`":

1. t_1 `==>` t_2 `|-` t_1 `==>` t_2

[1]This corresponds to the HOL theory `BASIC-HOL`; see Section 11.9

2. t_1 `|-` t_1

3. t_1 `==>` t_2`,` t_1 `|-` t_2

That is, the third theorem follows from the first and second.

In the session below, the proof is performed in the HOL system, using the ML functions `ASSUME` and `MP`. (It is helpful to adjust the theorem printing format first; see Section 8.14.2.)

1
```
#top_print print_all_thm;;
- : (thm -> void)

#let th1 = ASSUME "t1 ==> t2";;
th1 = t1 ==> t2 |- t1 ==> t2

#let th2 = ASSUME "t1:bool";;
th2 = t1 |- t1

#MP th1 th2;;
t1 ==> t2, t1 |- t2
```

More briefly, one could evaluate the following, and 'count' the invocations of functions representing primitive inferences (see Section 8.13, for the counter flag 'timing').

2
```
#set_flag(`timing`, true);;
false : bool
Run time: 0.0s

#MP(ASSUME "t1 ==> t2")(ASSUME "t1:bool");;
t1 ==> t2, t1 |- t2
Run time: 0.0s
Intermediate theorems generated: 3
```

Each of the three inference steps of the abstract proof corresponds to the application of an ML function in the performance of the proof in HOL; and each of the ML functions corresponds to a primitive inference of the deductive system.

It is worth emphasising that, in either case, every primitive inference in the proof chain is made, in the sense that for each inference, the corresponding ML function is evaluated. That is, HOL permits no short-cut around the necessity of performing complete proofs. The short-cut provided by derived inference rules (as implemented in ML) is around the necessity of *specifying* every step; something that would be impossible for a proof of any length. It can be seen from this that the derived rule, and its representation as an ML function, is essential to the HOL methodology; theorem proving would be otherwise impossible.

There are, of course, an infinite number of proofs, of the 'form' shown in the example, that can be conducted in HOL: one for every pair of `":bool"`-typed terms. Moreover, every time a theorem of the form

$$t_1 \Rightarrow t_2,\ t_1 \vdash t_2$$

is required, its proof must be constructed anew. To capture the general pattern of inference, an ML function can be written to implement an inference rule as a derivation from the primitive inferences. Abstractly, a *derived inference rule* is a rule that can be justified on the basis of the primitive inference rules (and/or the axioms). In the present case, the rule required 'undischarges' assumptions. It is specified for HOL by

$$\frac{\Gamma \texttt{ |- } t_1 \texttt{ ==> } t_2}{\Gamma \cup \{t_1\} \texttt{ |- } t_2}$$

This general rule is valid because from a HOL theorem of the form Γ `|-` t_1`==>`t_2, the theorem $\Gamma \cup \{t_1\}$ `|-` t_2 can be derived as for the specific instance above. The rule can be implemented in ML as a function (`UNDISCH`, say) that calls the appropriate sequence of primitive inferences. The ML definition of `UNDISCH` is simply

```
#let UNDISCH th = MP th (ASSUME(fst(dest_imp(concl th))));;                    3
UNDISCH = - : (thm -> thm)
```

This provides a function that maps a theorem to a theorem; that is, performs proofs in HOL. The following session illustrates the use of the derived rule, on a consequence of the axiom `IMP_ANTISYM_AX`. (The inferences are counted; see Section 8.13.) Assume that the printing of theorems has been adjusted as above and `th` is bound as shown below:

```
#th;;                                                                          1
|- (t1 ==> t2) ==> (t2 ==> t1) ==> (t1 = t2)
Run time: 0.0s

#set_flag(`timing`,true);;
true : bool
Run time: 0.0s

#UNDISCH th;;
t1 ==> t2 |- (t2 ==> t1) ==> (t1 = t2)
Run time: 0.1s
Intermediate theorems generated: 2

#UNDISCH it;;
t1 ==> t2, t2 ==> t1 |- t1 = t2
Run time: 0.0s
Intermediate theorems generated: 2
```

Each successful application of UNDISCH to a theorem invokes an application of ASSUME, followed by an application of MP; UNDISCH constructs the 2-step proof for any given theorem (of appropriate form). As can be seen, it relies on the class of ML functions that access HOL syntax: in particular, concl to produce the conclusion of the theorem, dest_imp to separate the implication, and the selector fst to choose the antecedent.

This particular example is very simple, but a derived inference rule can perform proofs of arbitrary length. It can also make use of previously defined rules. In this way, the normal inference patterns can be developed much more quickly and easily; transitivity, generalization, and so on, support the familiar patterns of inference.

A number of derived inference rules are pre-defined when the HOL system is entered (of which UNDISCH is one of the first). In Section 12.3, the abstract derivations are given for the pre-defined rules that reflect the more usual inference patterns of the predicate (and lambda) calculi. Like those shown, some of the pre-defined derived rules in HOL generate relatively short proofs. Others invoke thousands of primitive inferences, and clearly save a great deal of effort. Furthermore, rules can be defined by the user to make still larger steps, or to implement more specialized patterns.

All of the pre-defined derived rules in HOL are described in *REFERENCE*.

12.2 Rewriting

Included in the set of derived inferences that are pre-defined in HOL is a group of rules with complex definitions that do a limited amount of 'automatic' theorem-proving in the form of rewriting. The ideas and implementation were originally developed by Milner and Wadsworth for Edinburgh LCF, and were later implemented more flexibly and efficiently by Paulson and Huet for Cambridge LCF. They appear in HOL in the Cambridge form. The basic rewriting rule is REWRITE_RULE. All of the rewriting rules are described in detail in *REFERENCE*.

REWRITE_RULE uses a list of equational theorems (theorems whose conclusions can be regarded as having the form t_1 = t_2) to replace any subterms of an object theorem that 'match' t_1 by the corresponding instance of t_2. The rule matches recursively and to any depth, until no more replacements can be made, using internally defined search, matching and instantiation algorithms. The validity of REWRITE_RULE rests ultimately on the primitive rules SUBST (for making the substitutions); INST_TYPE (for instantiating types); and the derived rules for generalization and specialization (see Sections 12.3.13 and 12.3.11) for instantiating terms. The definition of REWRITE_RULE in ML also relies on a large number of general and HOL-oriented ML functions. The implementation is partly described in Chapter 13.

In practice, the derived rule REWRITE_RULE plays a central role in proofs, because it takes over a very large number of inferences which may happen in a complex and unpredictable order. It is unlike any other primitive or pre-defined rule, first because of the number of

inferences it generates[2]; and second because its outcome is often unexpected. Its power is increased by the fact that any existing equational theorem can be supplied as a 'rewrite rule', including a standard HOL set of pre-proved tautologies; and these rewrite rules can interact with each other in the rewriting process to transform the original theorem.

The application of REWRITE_RULE, in the session below, illustrates that replacements are made at all levels of the structure of a term. The example is numerical; the infixes "$>" and "$<" are the usual 'greater than' and 'less than' relations, respectively, and "SUC", the usual successor function. Use is made of the pre-existing definition of "$>": GREATER (see *REFERENCE*). The timing facility is used again, for interest, and the printing of theorems is adjusted as above.

1

```
#top_print print_all_thm;;
- : (thm -> void)

#set_flag(`timing`,true);;
false : bool
Run time: 0.0s

#REWRITE_RULE
 [GREATER]
 (ASSUME "SUC 4 > 0 = (SUC 3 > 0 = (SUC 2 > 0 = (SUC 1 > 0 = SUC 0 > 0)))");;
##Definition GREATER autoloaded from theory `arithmetic`.
GREATER = |- !m n. m > n = n < m
Run time: 1.5s
Intermediate theorems generated: 1

(SUC 4) > 0 =
((SUC 3) > 0 = ((SUC 2) > 0 = ((SUC 1) > 0 = (SUC 0) > 0)))
|- 0 < (SUC 4) =
   (0 < (SUC 3) = (0 < (SUC 2) = (0 < (SUC 1) = 0 < (SUC 0))))
Run time: 0.3s
Intermediate theorems generated: 23
```

Notice that rewriting equations can be extracted from universally quantified theorems. To construct the proof step-wise, with all of the instantiations, substitutions, and uses of transitivity, etc., would be a lengthy process. The rewriting rules make it easy, and do so whilst still generating the entire chain of inferences.

12.3 Derivation of the standard rules

The HOL system provides all the standard introduction and elimination rules of the predicate calculus pre-defined as derived inferences. It is these derived rules, rather than

[2]The number of inferences performed by this rule is generally 'inflated'; i.e. is generally greater than the length of the proof itself, if the proof could be 'seen'. This is because, in the current implementation, some inference is done during the search phase that is not necessarily in support of successful replacements.

the primitive rules, that one normally uses in practice. In this section, the derivations of some of the standard rules are given, in sequence. These derivations only use the axioms and definitions in the theory `bool` (see Section 11.8.4.1), the eight primitive inferences of the HOL logic, and inferences defined earlier in the sequence.

Theorems, in accordance with the definition given at the beginning of this chapter, are treated as rules without hypotheses; thus the derivation of a theorem resembles the derivation of a rule except in not having hypotheses. (The derivation of `TRUTH`, Section 12.3.9, is the only example given of this, but there are several others in HOL.) There are also some rules that are intrinsically more general than theorems. For example, for any two terms t_1 and t_2, the theorem $\vdash (\lambda x.\ t_1)t_2 = t_1[t_2/x]$ follows by the primitive rule `BETA_CONV`. The rule `BETA_CONV` returns a theorem for each pair of terms t_1 and t_2, and is therefore equivalent to an infinite family of theorems. No single theorem can be expressed in the HOL logic that is equivalent to `BETA_CONV`. (See Chapter 13 for further discussion of this point.) (`UNDISCH` is not a rule of this sort, as it can, in fact, be expressed as a theorem.)

For each derivation given below, there is an ML function definition in the HOL system that implements the derived rule as a procedure in ML. The actual implementation in the HOL system differs in some cases from the derivations given here, since the system code has been optimised for improved performance.

In addition, for reasons that are mostly historical, not all the inferences that are derived in terms of the abstract logic are actually derived in the current version of the HOL system. That is, there are currently about forty rules that are installed in the system on an 'axiomatic' basis, all of which should be derived by explicit inference. Although the current status of these rules is not satisfactory, and it is planned, as a high priority, to derive them properly in a future version, their current status does not actually compromise the consistency of the logic. In effect, the existing HOL system has a deductive system more comprehensive than the one presented abstractly, but the model outlined in Chapter 10 would easily extend to cover it.

For reference, in HOL Version 2.0 the following rules that should be derived are not derived, but (for efficiency) are implemented as primitives. The list includes some conversions and conversion-valued functions (conversions are discussed in Chapter 13).

```
ADD_ASSUM              CONTR                IMP_ANTISYM_RULE
ALPHA                  DEF_EXISTS_RULE      IMP_TRANS
AP_TERM                DISJ_CASES           INST
AP_THM                 DISJ1                MK_ABS
SUBS                   DISJ2                MK_COMB
SUBS_OCCS              EQ_IMP_RULE          MK_EXISTS
CCONTR                 EQ_MP                NOT_ELIM
CHOOSE                 EQT_INTRO            NOT_INTRO
CONJ                   ETA_CONV             num_CONV
EXISTS                 SPEC                 TRANS
EXT                    SUBST_CONV           CONJUNCT1
GEN                    SYM                  CONJUNCT2
```

The derivations that follow consist of sequences of numbered steps each of which

1. is an axiom, or
2. is a hypothesis of the rule being derived, or
3. follows from preceding steps by a rule of inference (either primitive or previously derived).

Note that the abbreviation `conv` is used for the ML type `term -> thm`.[3]

12.3.1 Adding an assumption

```
ADD_ASSUM : term -> thm -> thm
```

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

1. $t' \vdash t'$ [ASSUME]
2. $\Gamma \vdash t$ [Hypothesis]
3. $\Gamma \vdash t' \Rightarrow t$ [DISCH 2]
4. $\Gamma,\ t' \vdash t$ [MP 3,1]

[3]This stands for 'conversion', as explained in Chapter 13.

12.3.2 Undischarging

```
UNDISCH : thm -> thm
```

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

1. $t_1 \vdash t_1$ [ASSUME]
2. $\Gamma \vdash t_1 \Rightarrow t_2$ [Hypothesis]
3. $\Gamma, t_1 \vdash t_2$ [MP 2,1]

12.3.3 Symmetry of equality

```
SYM : thm -> thm
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_2 = t_1}$$

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]
2. $\vdash t_1 = t_1$ [REFL]
3. $\Gamma \vdash t_2 = t_1$ [SUBST 1,2]

12.3.4 Transitivity of equality

```
TRANS : thm -> thm -> thm
```

$$\frac{\Gamma_1 \vdash t_1 = t_2 \qquad \Gamma_2 \vdash t_2 = t_3}{\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3}$$

1. $\Gamma_2 \vdash t_2 = t_3$ [Hypothesis]
2. $\Gamma_1 \vdash t_1 = t_2$ [Hypothesis]
3. $\Gamma_1 \cup \Gamma_2 \vdash t_1 = t_3$ [SUBST 1,2]

12.3.5 Application of a term to a theorem

```
AP_TERM : term -> thm -> thm
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t\ t_1 = t\ t_2}$$

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]
2. $\vdash t\ t_1 = t\ t_1$ [REFL]
3. $\Gamma \vdash t\ t_1 = t\ t_2$ [SUBST 1,2]

12.3.6 Application of a theorem to a term

```
AP_THM : thm -> conv
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_1\ t = t_2\ t}$$

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]
2. $\vdash t_1\ t = t_1\ t$ [REFL]
3. $\Gamma \vdash t_1\ t = t_2\ t$ [SUBST 1,2]

12.3.7 Modus Ponens for equality

```
EQ_MP : thm -> thm -> thm
```

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

1. $\Gamma_1 \vdash t_1 = t_2$ [Hypothesis]
2. $\Gamma_2 \vdash t_1$ [Hypothesis]
3. $\Gamma_1 \cup \Gamma_2 \vdash t_2$ [SUBST 1,2]

12.3.8 Implication from equality

```
EQ_IMP_RULE : thm -> (thm # thm)
```

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash t_1 \Rightarrow t_2 \qquad \Gamma \vdash t_2 \Rightarrow t_1}$$

1. $\Gamma \vdash t_1 = t_2$ [Hypothesis]
2. $t_1 \vdash t_1$ [ASSUME]
3. $\Gamma,\ t_1 \vdash t_2$ [EQ_MP 1,2]
4. $\Gamma \vdash t_1 \Rightarrow t_2$ [DISCH 3]
5. $\Gamma \vdash t_2 = t_1$ [SYM 1]
6. $t_2 \vdash t_2$ [ASSUME]
7. $\Gamma,\ t_2 \vdash t_1$ [EQ_MP 5,6]
8. $\Gamma \vdash t_2 \Rightarrow t_1$ [DISCH 7]
9. $\Gamma \vdash t_1 \Rightarrow t_2$ and $\Gamma \vdash t_2 \Rightarrow t_1$ [4,8]

12.3.9 ⊤-Introduction

```
TRUTH
```

$$\vdash \top$$

1. $\vdash \top = ((\lambda x.\ x) = (\lambda x.\ x))$ [Definition of ⊤]
2. $\vdash ((\lambda x.\ x) = (\lambda x.\ x)) = \top$ [SYM 1]
3. $\vdash (\lambda x.\ x) = (\lambda x.\ x)$ [REFL]
4. $\vdash \top$ [EQ_MP 2,3]

12.3.10 Equality-with-⊤ elimination

```
EQT_ELIM : thm -> thm
```

$$\frac{\Gamma \vdash t = \top}{\Gamma \vdash t}$$

1. $\Gamma \vdash t = \mathsf{T}$ [Hypothesis]
2. $\Gamma \vdash \mathsf{T} = t$ [SYM 1]
3. $\vdash \mathsf{T}$ [TRUTH]
4. $\Gamma \vdash t$ [EQ_MP 2,3]

12.3.11 Specialization (∀-elimination)

```
SPEC : term -> thm -> thm
```

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

- $t[t'/x]$ 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.

1. $\vdash \forall = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))$ [INST_TYPE applied to the definition of ∀]
2. $\Gamma \vdash \forall(\lambda x.\ t)$ [Hypothesis]
3. $\Gamma \vdash (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t)$ [SUBST 1,2]
4. $\vdash (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \mathsf{T}))$ [BETA_CONV]
5. $\Gamma \vdash (\lambda x.\ t) = (\lambda x.\ \mathsf{T})$ [EQ_MP 4,3]
6. $\Gamma \vdash (\lambda x.\ t)\ t' = (\lambda x.\ \mathsf{T})\ t'$ [AP_THM 5]
7. $\vdash (\lambda x.\ t)\ t' = t[t'/x]$ [BETA_CONV]
8. $\Gamma \vdash t[t'/x] = (\lambda x.\ t)\ t'$ [SYM 7]
9. $\Gamma \vdash t[t'/x] = (\lambda x.\ \mathsf{T})\ t'$ [TRANS 8,6]
10. $\vdash (\lambda x.\ \mathsf{T})\ t' = \mathsf{T}$ [BETA_CONV]
11. $\Gamma \vdash t[t'/x] = \mathsf{T}$ [TRANS 9,10]
12. $\Gamma \vdash t[t'/x]$ [EQT_ELIM 11]

12.3.12 Equality-with-T introduction

```
EQT_INTRO : thm -> thm
```

$$\frac{\Gamma \vdash t}{\Gamma \vdash t = \mathsf{T}}$$

1. $\vdash \forall b_1\ b_2.\ (b_1 \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow b_1) \Rightarrow (b_1 = b_2)$ [Axiom]

2. $\vdash\ \forall b_2.\ (t \Rightarrow b_2) \Rightarrow (b_2 \Rightarrow t) \Rightarrow (t = b_2)$ [SPEC 1]
3. $\vdash\ (t \Rightarrow \mathsf{T}) \Rightarrow (\mathsf{T} \Rightarrow t) \Rightarrow (t = \mathsf{T})$ [SPEC 2]
4. $\vdash\ \mathsf{T}$ [TRUTH]
5. $\vdash\ t \Rightarrow \mathsf{T}$ [DISCH 4]
6. $\vdash\ (\mathsf{T} \Rightarrow t) \Rightarrow (t = \mathsf{T})$ [MP 3,5]
7. $\Gamma\ \vdash\ t$ [Hypothesis]
8. $\Gamma\ \vdash\ \mathsf{T} \Rightarrow t$ [DISCH 7]
9. $\Gamma\ \vdash\ t = \mathsf{T}$ [MP 6,8]

12.3.13 Generalization ($\forall$-introduction)

```
GEN : term -> thm -> thm
```

$$\frac{\Gamma\ \vdash\ t}{\Gamma\ \vdash\ \forall x.\ t}$$

- Where x is not free in Γ.

1. $\Gamma\ \vdash\ t$ [Hypothesis]
2. $\Gamma\ \vdash\ t = \mathsf{T}$ [EQT_INTRO 1]
3. $\Gamma\ \vdash\ (\lambda x.\ t) = (\lambda x.\ \mathsf{T})$ [ABS 2]
4. $\vdash\ \forall(\lambda x.\ t) = \forall(\lambda x.\ t)$ [REFL]
5. $\vdash\ \forall = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))$ [INST_TYPE applied to the definition of $\forall$]
6. $\vdash\ \forall(\lambda x.\ t) = (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t)$ [SUBST 5,4]
7. $\vdash\ (\lambda P.\ P = (\lambda x.\ \mathsf{T}))(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \mathsf{T}))$ [BETA_CONV]
8. $\vdash\ \forall(\lambda x.\ t) = ((\lambda x.\ t) = (\lambda x.\ \mathsf{T}))$ [TRANS 6,7]
9. $\vdash\ ((\lambda x.\ t) = (\lambda x.\ \mathsf{T})) = \forall(\lambda x.\ \mathsf{T})$ [SYM 8]
10. $\Gamma\ \vdash\ \forall(\lambda x.\ t)$ [EQ_MP 9,3]

12.3.14 Simple α-conversion

```
SIMPLE_ALPHA
```

$$\vdash\ (\lambda x_1.\ t\ x_1) = (\lambda x_2.\ t\ x_2)$$

- Where neither x_1 nor x_2 occurs free in t.[4]

1. $\vdash\ (\lambda x_1.\ t\ x_1)\ x = t\ x$ [BETA_CONV]
2. $\vdash\ (\lambda x_2.\ t\ x_2)\ x = t\ x$ [BETA_CONV]
3. $\vdash\ t\ x = (\lambda x_2.\ t\ x_2)\ x$ [SYM 2]
4. $\vdash\ (\lambda x_1.\ t\ x_1)\ x = (\lambda x_2.\ t\ x_2)\ x$ [TRANS 1,3]
5. $\vdash\ (\lambda x.\ (\lambda x_1.\ t\ x_1)\ x) = (\lambda x.\ (\lambda x_2.\ t\ x_2)\ x)$ [ABS 4]
6. $\vdash\ \forall f.\ (\lambda x.\ f\ x) = f$ [Appropriately type-instantiated axiom]
7. $\vdash\ (\lambda x.\ (\lambda x_1.\ t\ x_1)x) = \lambda x_1.\ t\ x_1$ [SPEC 6]
8. $\vdash\ (\lambda x.\ (\lambda x_2.\ t\ x_2)x) = \lambda x_2.\ t\ x_2$ [SPEC 6]
9. $\vdash\ (\lambda x_1.\ t\ x_1) = (\lambda x.\ (\lambda x_1.\ t\ x_1)x)$ [SYM 7]
10. $\vdash\ (\lambda x_1.\ t\ x_1) = (\lambda x.\ (\lambda x_2.\ t\ x_2)x)$ [TRANS 9,5]
11. $\vdash\ (\lambda x_1.\ t\ x_1) = (\lambda x_2.\ t\ x_2)$ [TRANS 10,8]

12.3.15 η-conversion

```
ETA_CONV : conv
```

$$\vdash\ (\lambda x'.\ t\ x') = t$$

- Where x' does not occur free in t (we use x' rather than just x to motivate the use of SIMPLE_ALPHA in the derivation below).

1. $\vdash\ \forall f.\ (\lambda x.\ f\ x) = f$ [Appropriately type-instantiated axiom]
2. $\vdash\ (\lambda x.\ t\ x) = t$ [SPEC 1]
3. $\vdash\ (\lambda x'.\ t\ x') = (\lambda x.\ t\ x)$ [SIMPLE_ALPHA]
4. $\vdash\ (\lambda x'.\ t\ x') = t$ [TRANS 3,2]

[4] SIMPLE_ALPHA is included here because it is used in a subsequent derivation, but it is not actually in the HOL system, as it is subsumed by other functions.

12.3.16 Extensionality

```
EXT : thm -> thm
```

$$\frac{\Gamma \vdash \forall x.\ t_1\ x = t_2\ x}{\Gamma \vdash t_1 = t_2}$$

- Where x is not free in Γ, t_1 or t_2.

1. $\Gamma \vdash \forall x.\ t_1\ x = t_2\ x$ [Hypothesis]
2. $\Gamma \vdash t_1\ x = t_2\ x$ [SPEC 1]
3. $\Gamma \vdash (\lambda x.\ t_1\ x) = (\lambda x.\ t_2\ x)$ [ABS 2]
4. $\vdash (\lambda x.\ t_1\ x) = t_1$ [ETA_CONV]
5. $\vdash t_1 = (\lambda x.\ t_1\ x)$ [SYM 4]
6. $\Gamma \vdash t_1 = (\lambda x.\ t_2\ x)$ [TRANS 5,3]
7. $\vdash (\lambda x.\ t_2\ x) = t_2$ [ETA_CONV]
8. $\Gamma \vdash t_1 = t_2$ [TRANS 6,7]

12.3.17 ε-introduction

```
SELECT_INTRO : thm -> thm
```

$$\frac{\Gamma \vdash t_1\ t_2}{\Gamma \vdash t_1(\varepsilon\ t_1)}$$

1. $\vdash \forall P\ x.\ P\ x \Rightarrow P(\varepsilon\ P)$ [Suitably type-instantiated axiom]
2. $\vdash t_1\ t_2 \Rightarrow t_1(\varepsilon\ t_1)$ [SPEC 1 (twice)]
3. $\Gamma \vdash t_1\ t_2$ [Hypothesis]
4. $\Gamma \vdash t_1(\varepsilon\ t_1)$ [MP 2,3]

12.3.18 ε-elimination

```
SELECT_ELIM : thm -> (term # thm) -> thm
```

$$\frac{\Gamma_1 \vdash t_1(\varepsilon\, t_1) \qquad \Gamma_2,\ t_1\ v \vdash t}{\Gamma_1 \cup \Gamma_2 \vdash t}$$

- Where v occurs nowhere except in the assumption $t_1\ v$ of the second hypothesis.

1. $\Gamma_2,\ t_1\ v \vdash t$ [Hypothesis]
2. $\Gamma_2 \vdash t_1\ v \Rightarrow t$ [DISCH 1]
3. $\Gamma_2 \vdash \forall v.\ t_1\ v \Rightarrow t$ [GEN 2]
4. $\Gamma_2 \vdash t_1(\varepsilon\, t_1) \Rightarrow t$ [SPEC 3]
5. $\Gamma_1 \vdash t_1(\varepsilon\, t_1)$ [Hypothesis]
6. $\Gamma_1 \cup \Gamma_2 \vdash t$ [MP 4,5]

12.3.19 $\exists$-introduction

```
EXISTS : (term # term) -> thm -> thm
```

$$\frac{\Gamma \vdash t_1[t_2]}{\Gamma \vdash \exists x.\ t_1[x]}$$

- Where $t_1[t_2]$ denotes a term t_1 with some free occurrences of t_2 singled out, and $t_1[x]$ denotes the result of replacing these occurrences of t_1 by x, subject to the restriction that x doesn't become bound after substitution.

1. $\vdash (\lambda x.\ t_1[x])t_2 = t_1[t_2]$ [BETA_CONV]
2. $\vdash t_1[t_2] = (\lambda x.\ t_1[x])t_2$ [SYM 1]
3. $\Gamma \vdash t_1[t_2]$ [Hypothesis]
4. $\Gamma \vdash (\lambda x.\ t_1[x])t_2$ [EQ_MP 2,3]
5. $\Gamma \vdash (\lambda x.\ t_1[x])(\varepsilon(\lambda x.\ t_1[x]))$ [SELECT_INTRO 4]
6. $\vdash \exists = \lambda P.\ P(\varepsilon\ P)$ [INST_TYPE applied to the definition of $\exists$]
7. $\vdash \exists(\lambda x.\ t_1[x]) = (\lambda P.\ P(\varepsilon\ P))(\lambda x.\ t_1[x])$ [AP_THM 6]
8. $\vdash (\lambda P.\ P(\varepsilon\ P))(\lambda x.\ t_1[x]) = (\lambda x.\ t_1[x])(\varepsilon(\lambda x.\ t_1[x]))$ [BETA_CONV]
9. $\vdash \exists(\lambda x.\ t_1[x]) = (\lambda x.\ t_1[x])(\varepsilon(\lambda x.\ t_1[x]))$ [TRANS 7,8]
10. $\vdash (\lambda x.\ t_1[x])(\varepsilon(\lambda x.\ t_1[x])) = \exists(\lambda x.\ t_1[x])$ [SYM 9]
11. $\Gamma \vdash \exists(\lambda x.\ t_1[x])$ [EQ_MP 10,5]

12.3.20 ∃-elimination

```
CHOOSE : (term # thm) -> thm -> thm
```

$$\frac{\Gamma_1 \vdash \exists x.\, t[x] \qquad\qquad \Gamma_2,\ t[v] \vdash t'}{\Gamma_1 \cup \Gamma_2 \vdash t'}$$

- Where $t[v]$ denotes a term t with some free occurrences of the variable v singled out, and $t[x]$ denotes the result of replacing these occurrences of v by x, subject to the restriction that x doesn't become bound after substitution.

1. $\vdash \exists = \lambda P.\, P(\varepsilon\ P)$ [INST_TYPE applied to the definition of ∃]
2. $\vdash \exists(\lambda x.\, t[x]) = (\lambda P.\, P(\varepsilon\ P))(\lambda x.\, t[x])$ [AP_THM 1]
3. $\Gamma_1 \vdash \exists(\lambda x.\, t[x])$ [Hypothesis]
4. $\Gamma_1 \vdash (\lambda P.\, P(\varepsilon\ P))(\lambda x.\, t[x])$ [EQ_MP 2,3]
5. $\vdash (\lambda P.\, P(\varepsilon\ P))(\lambda x.\, t[x]) = (\lambda x.\, t[x])(\varepsilon(\lambda x.\, t[x]))$ [BETA_CONV]
6. $\Gamma_1 \vdash (\lambda x.\, t[x])(\varepsilon(\lambda x.\, t[x]))$ [EQ_MP 5,4]
7. $\vdash (\lambda x.\, t[x])v = t[v]$ [BETA_CONV]
8. $\vdash t[v] = (\lambda x.\, t[x])v$ [SYM 7]
9. $\Gamma_2,\ t[v] \vdash t'$ [Hypothesis]
10. $\Gamma_2 \vdash t[v] \Rightarrow t'$ [DISCH 9]
11. $\Gamma_2 \vdash (\lambda x.\, t[x])v \Rightarrow t'$ [SUBST 8,10]
12. $\Gamma_2,\ (\lambda x.\, t[x])v \vdash t'$ [UNDISCH 11]
13. $\Gamma_1 \cup \Gamma_2 \vdash t'$ [SELECT_ELIM 6,12]

12.3.21 Use of a definition

```
RIGHT_BETA : thm -> thm
```

$$\frac{\Gamma \vdash t = \lambda x.\, t'[x]}{\Gamma \vdash t\ t = t'[t]}$$

- Where t does not contain x.

1. $\Gamma \vdash t = \lambda x.\, t'[x]$ [Suitably type-instantiated hypothesis]
2. $\Gamma \vdash t\ t = (\lambda x.\, t'[x])\ t$ [AP_THM 1]
3. $\vdash (\lambda x.\, t'[x])\ t = t'[t]$ [BETA_CONV]
4. $\Gamma \vdash t\ t = t'[t]$ [TRANS 2,3]

12.3.22 Use of a definition

```
RIGHT_LIST_BETA : thm -> thm
```

$$\frac{\Gamma \vdash t = \lambda x_1 \cdots x_n.\ t'[x_1, \ldots, x_n]}{\Gamma \vdash t\ t_1 \cdots t_n = t'[t_1, \ldots, t_n]}$$

- Where none of the t_i contain any of the x_i.

1. $\Gamma \vdash t = \lambda x_1 \cdots x_n.\ t'[x_1, \ldots, x_n]$ [Suitably type-instantiated hypothesis]
2. $\Gamma \vdash t\ t_1 \cdots t_n = (\lambda x_1 \cdots x_n.\ t'[x_1, \ldots, x_n])\ t_1 \cdots t_n$ [AP_THM 1 (n times)]
3. $\vdash (\lambda x_1 \cdots x_n.\ t'[x_1, \ldots, x_n])\ t_1 \cdots t_n = t'[t_1, \ldots, t_n]$ [BETA_CONV (n times)]
4. $\Gamma \vdash t\ t_1 \cdots t_n = t'[t_1, \ldots, t_n]$ [TRANS 2,3]

12.3.23 $\wedge$-introduction

```
CONJ : thm -> thm -> thm
```

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

1. $\vdash \wedge = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$ [Definition of $\wedge$]
2. $\vdash t_1 \wedge t_2 = \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [RIGHT_LIST_BETA 1]
3. $t_1 \Rightarrow (t_2 \Rightarrow b) \vdash t_1 \Rightarrow (t_2 \Rightarrow b)$ [ASSUME]
4. $\Gamma_1 \vdash t_1$ [Hypothesis]
5. $\Gamma_1,\ t_1 \Rightarrow (t_2 \Rightarrow b) \vdash t_2 \Rightarrow b$ [MP 3,4]
6. $\Gamma_2 \vdash t_2$ [Hypothesis]
7. $\Gamma_1 \cup \Gamma_2,\ t_1 \Rightarrow (t_2 \Rightarrow b) \vdash b$ [MP 5,6]
8. $\Gamma_1 \cup \Gamma_2 \vdash (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [DISCH 7]
9. $\Gamma_1 \cup \Gamma_2 \vdash \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$ [GEN 8]
10. $\Gamma_1 \cup \Gamma_2 \vdash t_1 \wedge t_2$ [EQ_MP (SYM 2),9]

12.3.24 $\wedge$-elimination

```
CONJUNCT1 : thm -> thm, CONJUNCT2 : thm -> thm
```

$$\frac{\Gamma \vdash t_1 \wedge t_2}{\Gamma \vdash t_1 \qquad\qquad \Gamma \vdash t_2}$$

1. $\vdash \ \wedge = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow (b_2 \Rightarrow b)) \Rightarrow b$	[Definition of $\wedge$]
2. $\vdash t_1 \wedge t_2 = \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$	[RIGHT_LIST_BETA 1]
3. $\Gamma \vdash t_1 \wedge t_2$	[Hypothesis]
4. $\Gamma \vdash \forall b.\ (t_1 \Rightarrow (t_2 \Rightarrow b)) \Rightarrow b$	[EQ_MP 2,3]
5. $\Gamma \vdash (t_1 \Rightarrow (t_2 \Rightarrow t_1)) \Rightarrow t_1$	[SPEC 4]
6. $t_1 \vdash t_1$	[ASSUME]
7. $t_1 \vdash t_2 \Rightarrow t_1$	[DISCH 6]
8. $\vdash t_1 \Rightarrow (t_2 \Rightarrow t_1)$	[DISCH 7]
9. $\Gamma \vdash t_1$	[MP 5,8]
10. $\Gamma \vdash (t_1 \Rightarrow (t_2 \Rightarrow t_2)) \Rightarrow t_2$	[SPEC 4]
11. $t_2 \vdash t_2$	[ASSUME]
12. $\vdash t_2 \Rightarrow t_2$	[DISCH 11]
13. $\vdash t_1 \Rightarrow (t_2 \Rightarrow t_2)$	[DISCH 12]
14. $\Gamma \vdash t_2$	[MP 10,13]
15. $\Gamma \vdash t_1$ and $\Gamma \vdash t_2$	[9,14]

12.3.25 Right $\vee$-introduction

```
DISJ1 : thm -> conv
```

$$\frac{\Gamma \vdash t_1}{\Gamma \vdash t_1 \vee t_2}$$

1. $\vdash \ \vee = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow (b_2 \Rightarrow b) \Rightarrow b$	[Definition of $\vee$]
2. $\vdash t_1 \vee t_2 = \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$	[RIGHT_LIST_BETA 1]
3. $\Gamma \vdash t_1$	[Hypothesis]
4. $t_1 \Rightarrow b \vdash t_1 \Rightarrow b$	[ASSUME]
5. $\Gamma,\ t_1 \Rightarrow b \vdash b$	[MP 4,3]

6. $\Gamma,\ t_1 \Rightarrow b\ \vdash\ (t_2 \Rightarrow b) \Rightarrow b$ [DISCH 5]
7. $\Gamma\ \vdash\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [DISCH 6]
8. $\Gamma\ \vdash\ \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [GEN 7]
9. $\Gamma\ \vdash\ t_1\ \vee\ t_2$ [EQ_MP (SYM 2),8]

12.3.26 Left $\vee$-introduction

```
DISJ2 : term -> thm -> thm
```

$$\frac{\Gamma\ \vdash\ t_2}{\Gamma\ \vdash\ t_1\ \vee\ t_2}$$

1. $\vdash\ \ \vee\ = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow (b_2 \Rightarrow b) \Rightarrow b$ [Definition of $\vee$]
2. $\vdash\ t_1\ \vee\ t_2 = \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [RIGHT_LIST_BETA 1]
3. $\Gamma\ \vdash\ t_2$ [Hypothesis]
4. $t_2 \Rightarrow b\ \vdash\ t_2 \Rightarrow b$ [ASSUME]
5. $\Gamma,\ t_2 \Rightarrow b\ \vdash\ b$ [MP 4,3]
6. $\Gamma\ \vdash\ (t_2 \Rightarrow b) \Rightarrow b$ [DISCH 5]
7. $\Gamma\ \vdash\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [DISCH 6]
8. $\Gamma\ \vdash\ \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [GEN 7]
9. $\Gamma\ \vdash\ t_1\ \vee\ t_2$ [EQ_MP (SYM 2),8]

12.3.27 $\vee$-elimination

```
DISJ_CASES : thm -> thm -> thm -> thm
```

$$\frac{\Gamma\ \vdash\ t_1\ \vee\ t_2 \qquad \Gamma_1,\ t_1\ \vdash\ t \qquad \Gamma_2,\ t_2\ \vdash\ t}{\Gamma \cup \Gamma_1 \cup \Gamma_2\ \vdash\ t}$$

1. $\vdash\ \ \vee\ = \lambda b_1\ b_2.\ \forall b.\ (b_1 \Rightarrow b) \Rightarrow (b_2 \Rightarrow b) \Rightarrow b$ [Definition of $\vee$]
2. $\vdash\ t_1\ \vee\ t_2 = \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [RIGHT_LIST_BETA 1]
3. $\Gamma\ \vdash\ t_1\ \vee\ t_2$ [Hypothesis]
4. $\Gamma\ \vdash\ \forall b.\ (t_1 \Rightarrow b) \Rightarrow (t_2 \Rightarrow b) \Rightarrow b$ [EQ_MP 2,3]
5. $\Gamma\ \vdash\ (t_1 \Rightarrow t) \Rightarrow (t_2 \Rightarrow t) \Rightarrow t$ [SPEC 4]
6. $\Gamma_1,\ t_1\ \vdash\ t$ [Hypothesis]

7. $\Gamma_1 \vdash t_1 \Rightarrow t$ [DISCH 6]
8. $\Gamma \cup \Gamma_1 \vdash (t_2 \Rightarrow t) \Rightarrow t$ [MP 5,7]
9. $\Gamma_2,\ t_2 \vdash t$ [Hypothesis]
10. $\Gamma_2 \vdash t_2 \Rightarrow t$ [DISCH 9]
11. $\Gamma \cup \Gamma_1 \cup \Gamma_2 \vdash t$ [MP 8,10]

12.3.28 Classical contradiction rule

```
CCONTR : term -> thm -> thm
```

$$\frac{\Gamma,\ \neg t \vdash \mathsf{F}}{\Gamma \vdash t}$$

1. $\vdash\ \neg = \lambda b.\ b \Rightarrow \mathsf{F}$ [Definition of $\neg$]
2. $\vdash\ \neg t = t \Rightarrow \mathsf{F}$ [RIGHT_LIST_BETA 1]
3. $\Gamma,\ \neg t \vdash \mathsf{F}$ [Hypothesis]
4. $\Gamma \vdash \neg t \Rightarrow \mathsf{F}$ [DISCH 3]
5. $\Gamma \vdash (t \Rightarrow \mathsf{F}) \Rightarrow \mathsf{F}$ [SUBST 2,4]
6. $t = \mathsf{F} \vdash t = \mathsf{F}$ [ASSUME]
7. $\Gamma,\ t = \mathsf{F} \vdash (\mathsf{F} \Rightarrow \mathsf{F}) \Rightarrow \mathsf{F}$ [SUBST 6,5]
8. $\mathsf{F} \vdash \mathsf{F}$ [ASSUME]
9. $\vdash\ \mathsf{F} \Rightarrow \mathsf{F}$ [DISCH 8]
10. $\Gamma,\ t = \mathsf{F} \vdash \mathsf{F}$ [MP 7,9]
11. $\vdash\ \mathsf{F} = \forall b.\ b$ [Definition of F]
12. $\Gamma,\ t = \mathsf{F} \vdash \forall b.\ b$ [SUBST 11,10]
13. $\Gamma,\ t = \mathsf{F} \vdash t$ [SPEC 12]
14. $\vdash\ \forall b.\ (b = \mathsf{T})\ \vee\ (b = \mathsf{F})$ [Axiom]
15. $\vdash\ (t = \mathsf{T})\ \vee\ (t = \mathsf{F})$ [SPEC 14]
16. $t = \mathsf{T} \vdash t = \mathsf{T}$ [ASSUME]
17. $t = \mathsf{T} \vdash t$ [EQT_ELIM 16]
18. $\Gamma \vdash t$ [DISJ_CASES 15,17,13]

Chapter 13

Conversions

A *conversion* in HOL is a rule that maps a term to a theorem expressing the equality of that term to some other term. An example is the rule for β-conversion:

$$(\backslash x.t_1)t_2 \quad \mapsto \quad \texttt{|- } (\backslash x.t_1)t_2 \texttt{ = } t_1[t_2/x]$$

Theorems of this sort are used in HOL in a variety of contexts, to justify the replacement of particular terms by semantically equivalent terms.

The ML type of conversions is `conv`:

```
conv = term -> thm
```

For example, `BETA_CONV` is an ML function of type `conv` (i.e. a conversion) that expresses β-conversion in HOL. It produces the appropriate equational theorem on β-redexes and fails elsewhere.

1

```
#BETA_CONV;;
- : conv

#BETA_CONV "(\x. (\y. (\z. x + y + z)3)2) 1";;
|- (\x. (\y. (\z. x + (y + z))3)2)1 = (\y. (\z. 1 + (y + z))3)2

#BETA_CONV "(\y. (\z. 1 + (y + z))3) 2";;
|- (\y. (\z. 1 + (y + z))3)2 = (\z. 1 + (2 + z))3

#BETA_CONV "(\z. 1 + (2 + z)) 3";;
|- (\z. 1 + (2 + z))3 = 1 + (2 + 3)

#BETA_CONV "1 + (2 + 3)";;
evaluation failed     BETA_CONV
```

The basic conversions, as well as a number of those commonly used, are provided in HOL. There are also groups of application-specific conversions to be found in some of the libraries. (Of those provided, some are derived and some, like `BETA_CONV` are taken as axiomatic[1].) In addition, HOL provides a collection of ML functions enabling users to define new conversions (as well as new rules and tactics) as functions of existing ones.

[1] A list of the axiomatic rules was supplied in Section 12.3.

Some of these are described in Sections 13.1 and 13.2. The notion of conversions is inherited from Cambridge LCF; the underlying principles are described in [15, 16].

Conversions such as `BETA_CONV` represent infinite families of equations[2]. They are particularly useful in cases in which it is impossible to state, within the logic, a single axiom or theorem instantiable to every equation in a family.[3] Instead, an ML procedure returns the instance of the desired theorem for any given term. This is also the reason that quite a few of the other rules in HOL are not stated instead as axioms or theorems. As rules, conversions are distinguished with an ML type abbreviation simply because there are relatively many of them with the same type, and because they return equational theorems that lend themselves directly to term rewriting.[4] In many HOL applications, the main use of conversions is to produce these equational theorems. A few examples of conversions are illustrated below.

```
#NOT_FORALL_CONV "~!x. (f:*->*) x = g x";;                                    1
|- ~(!x. f x = g x) = (?x. ~(f x = g x))

#CONTRAPOS_CONV "(!x. f x = g x) ==> ((f:*->*) = g)";;
|- (!x. f x = g x) ==> (f = g) = ~(f = g) ==> ~(!x. f x = g x)

#SELECT_CONV "(@f:*->*. f x = g x)x = g x";;
|- ((@f. f x = g x)x = g x) = (?f. f x = g x)

#EXISTS_UNIQUE_CONV "?!z. (f:*->*) z = g z";;
|- (?! z. f z = g z) =
   (?z. f z = g z) /\ (!z z'. (f z = g z) /\ (f z' = g z') ==> (z = z'))
```

An example of an application specific conversion is `num_CONV`:

```
#num_CONV "2";;                                                              1
|- 2 = SUC 1

#num_CONV "1";;
|- 1 = SUC 0

#num_CONV "0";;
evaluation failed     num_CONV: argument less than 1
```

Another application of conversions, related to the first, is in the implementation of the existing rewriting tools, `REWRITE_RULE` (Section 12.2) and `REWRITE_TAC` (Chapter 14),

[2] This was also mentioned in Section 12.3.

[3] In the case of β-conversion specifically, it is the substitution of one term for another in a context that is inexpressible; but in general, there is a variety of reasons that arise.

[4] In fact, some ML functions have names with the suffix '`_CONV`' but do not have the type `conv`; `SUBST_CONV`, for example, has type `(thm # term) list -> term -> conv`. Those that eventually produce conversion are thought of as 'conversion schemas'.

which are central to theorem proving in HOL. This use is explained in Section 13.4, both as an example and because users may have occasion to construct rewriting tools of their own design, by similar methods. The next section introduces the conversion-building tools in general.

13.1 Conversion combining operators

A term u is said to *reduce* to a term v by a conversion c if there exists a finite sequence of terms t_1, t_2, ..., t_n such that:

(i) $u = t_1$ and $v = t_n$;

(ii) $c\ t_i$ evaluates to the theorem `|-` t_i `=` t_{i+1} for $1 \leq i < n$;

(iii) The evaluation of $c\ t_n$ fails.

The first session of this chapter illustrates the reduction of the term

```
(\x. (\y. (\z. x + y + z)3)2)1
```

to `1 + (2 + 3)` by the conversion `BETA_CONV`, in a reduction sequence of length four:

```
(\x. (\y. (\z. x + (y + z))3)2)1
(\y. (\z. 1 + (y + z))3)2
(\z. 1 + (2 + z))3
1 + (2 + 3)
```

That is, `BETA_CONV` applies to each term of the sequence, except the fourth and last, to give a theorem equating that term to the next term. Therefore, each term of the sequence, from the second on, can be extracted from the theorem for the previous term; namely, it is the right hand side of the conclusion. The whole reduction can therefore be accomplished by repeated application of `BETA_CONV` to the terms of the sequence as they are generated.

To transform `BETA_CONV` to achieve this effect, two operators on conversions are introduced. The first one, infixed, is `THENC`, which sequences conversions.

```
$THENC : conv -> conv -> conv
```

If $c_1\ t_1$ evaluates to Γ_1 `|-` t_1`=`t_2 and $c_2\ t_2$ evaluates to Γ_2 `|-` t_2`=`t_3, then (`(`c_1 `THENC` c_2`)`) t_1 evaluates to $\Gamma_1 \cup \Gamma_2$ `|-` t_1`=`t_3. If the evaluation of $c_1\ t_1$ or the evaluation of $c_2\ t_2$ fails, then so does the evaluation of c_1 `THENC` c_2. `THENC` is justified by the transitivity of equality.

The second, also infixed, is `ORELSEC`; this applies a second conversion if the application of the first one fails.

```
$ORELSEC : conv -> conv -> conv
```

(c_1 `ORELSEC` c_2) t evaluates to c_1 t if that evaluation succeeds, and to c_2 t otherwise. (The failure to evaluate is detected via the ML failure construct.)

The functions `THENC` and `ORELSEC` are used to define the desired operator, `REPEATC`, which successively applies a conversion until it fails:

```
REPEATC : conv -> conv
```

`REPEATC` c is intuitively equivalent to:

(c `THENC` c `THENC ... THENC` c `THENC ...`) `ORELSEC ALL_CONV`

It is defined recursively:[5]

```
letrec REPEATC c t = ((c THENC (REPEATC c)) ORELSEC ALL_CONV) t
```

The current example term can thus be completely reduced by use of `BETA_CONV` transformed by the `REPEATC` operator:

```
#REPEATC BETA_CONV;;                                                    1
- : conv

#REPEATC BETA_CONV "(\x. (\y. (\z. x + y + z)3)2)1";;
|- (\x. (\y. (\z. x + (y + z))3)2)1 = 1 + (2 + 3)
```

`BETA_CONV` applies to terms of a certain top level form only, namely to β-redexes, and fails on terms of any other form. In addition, no number of repetitions of `BETA_CONV` will β-reduce *arbitrary* β-redexes embedded in terms. For example, the term shown below fails even at the top level because it is not a β-redex:

```
#BETA_CONV "(((\x.(\y.(\z. x + y + z))) 1) 2) 3";;                      1
evaluation failed     BETA_CONV

#is_abs "(((\x.(\y.(\z. x + y + z))) 1) 2)";;
false : bool
```

The β-redex `(\w.w)3` is not affected in the third input of the session shown below, because of its position in the structure of the whole term. This is so even though the whole term is reduced, and the subterm at top level could be reduced:

[5] Note that because ML is a call-by-value language, the extra argument t is needed in the definition of `REPEATC`; without it the definition would loop. There is a similar problem with the tactical `REPEAT`; see Chapter 14.

```
#BETA_CONV "(\z. x + y + z)3";;                                          1
|- (\z. x + (y + z))3 = x + (y + 3)

#BETA_CONV "(\w.w)3";;
|- (\w. w)3 = 3

#REPEATC BETA_CONV "(\z. x + y + z)((\w.w)3)";;
|- (\z. x + (y + z))((\w. w)3) = x + (y + ((\w. w)3))
```

To produce, from a conversion c, a conversion that applies c to every subterm of a term, the function `DEPTH_CONV` can be applied to c:

```
   DEPTH_CONV : conv -> conv
```

`DEPTH_CONV` c is a conversion

$$t \quad \mapsto \quad \texttt{|- } t \texttt{ = } t'$$

where t' is obtained from t by replacing every subterm u by v if u reduces to v by c. (Subterms for which c u fails are left unchanged.) The definition leaves open the search strategy; in fact, `DEPTH_CONV` c traverses a term[6] 'bottom up', once, and left-to-right, repeatedly applying c to each subterm until no longer applicable. This helps with the two problems thus far:

```
#DEPTH_CONV BETA_CONV "(((\x.(\y.(\z. x + y + z))) 1) 2) 3";;             1
|- (\x y z. x + (y + z))1 2 3 = 1 + (2 + 3)

#DEPTH_CONV BETA_CONV "(\z. x + y + z)((\w.w)3)";;
|- (\z. x + (y + z))((\w. w)3) = x + (y + 3)
```

It may happen, however, that the result of such a conversion still contains subterms that could themselves be reduced at top level. For example:

```
#let t = "(\f.\x.f x)(\n.n+1)";;                                         1
t = "(\f x. f x)(\n. n + 1)" : term

#DEPTH_CONV BETA_CONV t;;
|- (\f x. f x)(\n. n + 1) = (\x. (\n. n + 1)x)
```

The function `TOP_DEPTH_CONV` does more searching and reduction than `DEPTH_CONV`: it replaces every subterm u by v' if u reduces to v by c and v *recursively* reduces to v' by `TOP_DEPTH_CONV` c.[7]

[6] That is, it traverses the abstract parse tree of the term.

[7] Readers interested in characterizing the search strategy of `TOP_DEPTH_CONV` should study the ML definitions near the end of this section.

```
TOP_DEPTH_CONV : conv -> conv
```

Thus:

```
#TOP_DEPTH_CONV BETA_CONV t;;                                          2
|- (\f x. f x)(\n. n + 1) = (\x. x + 1)
```

Finally, the simpler function `ONCE_DEPTH_CONV` is provided:

```
ONCE_DEPTH_CONV : conv -> conv
```

`ONCE_DEPTH_CONV` *c t* applies *c* once to the first term (and only the first term) on which it succeeds in a top-down traversal:

```
#ONCE_DEPTH_CONV BETA_CONV t;;                                         3
|- (\f x. f x)(\n. n + 1) = (\x. (\n. n + 1)x)

#ONCE_DEPTH_CONV BETA_CONV "(\x. (\n. n + 1)x)";;
|- (\x. (\n. n + 1)x) = (\x. x + 1)
```

The equational theorems returned by conversions are not always useful in equational form. To make the results more useful for theorem proving, a conversion can be converted to a rule or a tactic, using the functions `CONV_RULE` or `CONV_TAC`, respectively.

```
CONV_RULE : conv -> thm -> thm
CONV_TAC  : conv -> tactic
```

`CONV_RULE` *c* (`|-` *t*) returns `|-` *t′*, where *c t* evaluates to the equation `|-` *t*=*t′*. `CONV_TAC` *c* is a tactic that converts the conclusion of a goal using *c*. `CONV_RULE` is defined by:

```
let CONV_RULE c th = EQ_MP (c(concl th)) th
```

(The validation of `CONV_TAC` also uses `EQ_MP`[8].) For example, the built-in rule `BETA_RULE` reduces some of the β-redex subterms of a term.

```
BETA_RULE : thm -> thm
```

It is defined by:

```
let BETA_RULE = CONV_RULE(DEPTH_CONV BETA_CONV)
```

The search invoked by `BETA_RULE` is adequate for some purposes but not others; for example, the first use shown below but not the second:

[8] For `EQ_MP`, see 12.3.7.

```
#BETA_RULE(ASSUME "(((\x.(\y.(\z. x + y + z))) 1) 2) 3 < 10");;
. |- (1 + (2 + 3)) < 10

#let th = ASSUME "NEXT = ^t";;
th = . |- NEXT = (\f x. f x)(\n. n + 1)

#BETA_RULE th;;
. |- NEXT = (\x. (\n. n + 1)x)

#BETA_RULE(BETA_RULE th);;
. |- NEXT = (\x. x + 1)
```

4

A more powerful β-reduction rule that used the second search strategy could be defined as shown below (this is not built into HOL).

```
#let TOP_BETA_RULE = CONV_RULE(TOP_DEPTH_CONV BETA_CONV);;
TOP_BETA_RULE = - : (thm -> thm)

#TOP_BETA_RULE th;;
. |- NEXT = (\x. x + 1)
```

5

`TOP_DEPTH_CONV` is the traversal strategy used by the HOL rewriting tools described in Section 13.4.

13.2 Writing compound conversions

There are several other conversion operators in HOL, which, together with `THENC`, `ORELSEC` and `REPEATC` are available for building more complex conversions, as well as rules, tactics, and so on. These are described below; several are good illustrations themselves of how functions are built using conversions. The section culminates with the explanation of how `DEPTH_CONV`, `TOP_DEPTH_CONV`, and `ONCE_DEPTH_CONV` are built.

The conversion `NO_CONV` is an identity for `ORELSEC`, useful in building functions.

```
NO_CONV : conv
```

`NO_CONV` t always fails.

The function `FIRST_CONV` returns $c\ t$ for the first conversion c, in a list of conversions, for which the evaluation of $c\ t$ succeeds.

```
FIRST_CONV : conv list -> conv
```

`FIRST_CONV` $[c_1; \ldots; c_n]$ is equivalent, intuitively, to:

c_1 `ORELSEC` c_2 `ORELSEC` ... `ORELSEC` c_n

It is defined by:

```
let FIRST_CONV cl t =
    itlist $ORELSEC cl NO_CONV t ? failwith `FIRST_CONV`;;
```

The conversion ALL_CONV is an identity for THENC, useful in building functions.

```
ALL_CONV : conv
```

ALL_CONV t evaluates to |- t=t. It is defined as being identical to REFL.

The function EVERY_CONV applies a list of conversions in sequence.

```
EVERY_CONV : conv list -> conv
```

EVERY_CONV [c_1;...;c_n] is equivalent, intuitively, to:

c_1 THENC c_2 THENC ... THENC c_n

It is defined by:

```
let EVERY_CONV cl t =
    itlist $THENC cl ALL_CONV t ? failwith `EVERY_CONV`
```

The operator CHANGED_CONV converts one conversion to another that fails on arguments that it cannot change.

```
CHANGED_CONV : conv -> conv
```

If c t evaluates to |- t=t', then CHANGED_CONV c t also evaluates to |- t=t', unless t and t' are the same (up to α-conversion), in which case it fails.

The operator TRY_CONV maps one conversion to another that always succeeds, by replacing failures with the identity conversion.

```
TRY_CONV : conv
```

If c t evaluates to |- t=t', then TRY_CONV c t also evaluates to |- t=t'. If c t fails, then TRY_CONV c t evaluates to |- t=t. TRY_CONV is implemented by:

```
let TRY_CONV c = c ORELSEC ALL_CONV
```

It is used in the implementation of TOP_DEPTH_CONV (given later).

There are a number of operators for applying conversions to the immediate subterms of a term. These use the ML functions:

```
MK_COMB : thm # thm -> thm
MK_ABS  : thm -> thm
```

MK_COMB and MK_ABS implement the following derived rules:

$$\frac{\Gamma_1 \ \texttt{|-}\ u_1\texttt{=}v_1 \qquad \Gamma_2 \ \texttt{|-}\ u_2\texttt{=}v_2}{\Gamma_1 \cup \Gamma_2 \ \texttt{|-}\ u_1\ u_2\texttt{=}v_1\ v_2}\ \texttt{MK_COMB}$$

$$\frac{\Gamma \ \texttt{|-}\ \texttt{!}x.u\texttt{=}v}{\Gamma \ \texttt{|-}\ (\backslash x.u) = (\backslash x.v)}\ \texttt{MK_ABS}$$

The function SUB_CONV applies a conversion to the immediate subterms of a term.

```
SUB_CONV : conv
```

In particular:

- SUB_CONV c "x" = |- x=x;
- SUB_CONV c "$u\ v$" = |- $u\ v$=$u'\ v'$, if $c\ u$ = |- u=u' and $c\ v$ = |- v=v';
- SUB_CONV c "\$x.u$" = |- (\$x.u$) = (\$x.u'$), if $c\ u$ = |- u=u'.

SUB_CONV is implemented in terms of MK_COMB and MK_ABS:

```
let SUB_CONV c t =
    if is_comb t then
       (let rator,rand = dest_comb t in
        MK_COMB (c rator, c rand))
    if is_abs t then
       (let bv,body = dest_abs t in
        let bodyth = c body in
        MK_ABS (GEN bv bodyth))
                else (ALL_CONV t)
```

SUB_CONV, too, is used in the definitions of DEPTH_CONV and TOP_DEPTH_CONV.

Three other useful conversion operators, also for applying conversions to the immediate subterms of a term, are as follows:

```
RATOR_CONV : conv -> conv
RAND_CONV  : conv -> conv
ABS_CONV   : conv -> conv
```

RATOR_CONV c converts the operator of an application using c; RAND_CONV c converts the operand of an application; and ABS_CONV c converts the body of an abstraction. Combinations of these are useful for applying conversions to particular subterms. These are implemented by:

```
let RATOR_CONV c t =
 (let rator,rand = dest_comb t in
  MK_COMB (c rator, REFL rand)) ? failwith `RATOR_CONV`

let ABS_CONV c t =
 (let bv,body = dest_abs t in
  let bodyth = c body in
  MK_ABS (GEN bv bodyth)) ? failwith `ABS_CONV`
```

The following is an example session illustrating these immediate subterm conversions (recalling that the expression t_1+t_2 actually parses as + t_1 t_2).

```
#let t = "(\x.x+1)m + (\x.x+2)n";;                                        1
t = "((\x. x + 1)m) + ((\x. x + 2)n)" : term

#RAND_CONV BETA_CONV t;;
|- ((\x. x + 1)m) + ((\x. x + 2)n) = ((\x. x + 1)m) + (n + 2)

#RATOR_CONV (RAND_CONV BETA_CONV) t;;
|- ((\x. x + 1)m) + ((\x. x + 2)n) = (m + 1) + ((\x. x + 2)n)
```

Finally, the definitions of `DEPTH_CONV` and `TOP_DEPTH_CONV` are given below.

```
letrec DEPTH_CONV c t =
 (SUB_CONV (DEPTH_CONV c) THENC (REPEATC c)) t

letrec TOP_DEPTH_CONV c t =
 (REPEATC c THENC
  (TRY_CONV
    (CHANGED_CONV (SUB_CONV (TOP_DEPTH_CONV c)) THENC
     TRY_CONV(c THENC TOP_DEPTH_CONV c)))) t

letrec ONCE_DEPTH_CONV c t =
 (c ORELSEC (SUB_CONV (ONCE_DEPTH_CONV c))) t
```

Note that the extra argument `t` is needed to stop these definitions looping (because ML is a call-by-value language). Note also that the actual definition of `ONCE_DEPTH_CONV` used in the system has been optimised to use failure to avoid rebuilding unchanged subterms.

13.3 Built in conversions

Many conversions are predefined in HOL; only those likely to be of general interest are listed here.

13.3.1 Generalized beta-reduction

The conversion:

```
PAIRED_BETA_CONV : conv
```

does generalized beta-conversion of tupled lambda abstractions applied to tuples.

Given the term:

```
"(\(x1, ... ,xn).t) (t1, ... ,tn)"
```

`PAIRED_BETA_CONV` proves that:

```
|- (\(x1, ... ,xn). t[x1,...,xn]) (t1, ... ,tn)  =  t[t1, ... ,tn]
```

The conversion works for arbitrarily nested tuples. For example:

```
#PAIRED_BETA_CONV "(\((a,b),(c,d)). [a;b;c;d]) ((1,2),(3,4))";;                1
|- (\((a,b),c,d). [a;b;c;d])((1,2),3,4) = [1;2;3;4]
```

13.3.2 Arithmetical conversions

The conversion:

```
ADD_CONV : conv
```

does addition by formal proof. If n and m are numerals then `ADD_CONV "`n `+` m`"` returns the theorem `|-` n `+` m `=` s, where s is the numeral denoting the sum of n and m. For example:

```
#ADD_CONV "1 + 2";;                                                            1
|- 1 + 2 = 3

#ADD_CONV "0 + 1000";;
|- 0 + 1000 = 1000

#ADD_CONV "101 + 102";;
|- 101 + 102 = 203
```

The next conversion decides the equality of natural number constants.

```
num_EQ_CONV : conv
```

If n and m are terms constructed from numeral constants and the successor function `SUC`, then: `num_EQ_CONV "`n`=`m`"` returns:

```
|- (n=m) = T   if n and m represent the same number
|- (n=m) = F   if n and m represent different numbers
```

In addition, `num_EQ_CONV "`t `=` t`"` returns: `|- (`t`=`t`) = T`

13.3.3 List processing conversions

There are two useful built-in conversions for lists:

```
LENGTH_CONV : conv
list_EQ_CONV: conv
```

`LENGTH_CONV`: computes the length of a list. A call to:

`LENGTH_CONV "LENGTH[`t_1`;`...`;`t_n`]"`

generates the theorem:

`|- LENGTH [`t_1`;`...`;`t_n`] = n`

The other conversion, `list_EQ_CONV`, proves or disproves the equality of two lists, given a conversion for deciding the equality of elements. A call to:

`list_EQ_CONV` *conv* `"[`u_1`;`...`;`u_n`] = [`v_1`;`...`;`v_m`]"`

returns: `|- ([`u_1`;`...`;`u_n`] = [`v_1`;`...`;`v_m`]) = F` if:

(i) `~(`n`=`m`)` or

(ii) *conv* proves `|- (`u_i `= `v_i`) = F` for any $1 \leq i \leq m$.

`|- ([`u_1`;`...`;`u_n`] = [`v_1`;`...`;`v_m`]) = T` is returned if:

(i) `(`n`=`m`)` and u_i is syntactically identical to v_i for $1 \leq i \leq m$, or

(ii) `(`n`=`m`)` and *conv* proves `|- (`u_i`=`v_i`)=T` for $1 \leq i \leq n$.

13.3.4 Simplifying `let`-terms

A conversion for reducing `let`-terms is now provided.

```
let_CONV : conv
```

Given a term:

`"let` v_1 `=` t_1 `and` $\cdots$ `and` v_n `=` t_n `in` $t[v_1, \ldots, v_n]$`"`

`let_CONV` proves that:

`|- let` v_1 `=` t_1 `and` $\cdots$ `and` v_n `=` t_n `in` $t[v_1, \ldots, v_n]$ `=` $t[t_1, \ldots, t_n]$

The v_i's can take any one of the following forms:

(i) Variables: `x` etc.

(ii) Tuples: `(x,y)`, `(a,b,c)`, `((a,b),(c,d))` etc.

(iii) Applications: `f (x,y) z`, `f x` etc.

Variables are just substituted for. With tuples, the substitution is done component-wise, and function applications are effectively rewritten in the body of the `let`-term.

1

```
#let_CONV "let x = 1 in x+y";;
|- (let x = 1 in x + y) = 1 + y

#let_CONV "let (x,y) = (1,2) in x+y";;
|- (let (x,y) = 1,2 in x + y) = 1 + 2

#let_CONV "let f x = 1 and f y = 2 in (f 10) + (f 20)";;
|- (let f x = 1 and f y = 2 in (f 10) + (f 20)) = 2 + 2

#let_CONV "let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))";;
|- (let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))) =
(((0 + 2) + 1) + 2) + 1

#CONV_RULE(DEPTH_CONV ADD_CONV)it;;
|- (let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))) = 6

#let_CONV "let f x y = x+y in f 1";;  % NB: partial application %
|- (let f x y = x + y in f 1) = (\y. 1 + y)
```

13.3.5 Skolemization

Two conversions are provided for a higher-order version of Skolemization (using existentially quantified function variables rather than first-order Skolem constants).

The conversion

```
X_SKOLEM_CONV : term -> conv
```

takes a variable parameter, f say, and proves:

`|- (!`x_1 ... x_n`. ?`y`.` $t[x_1,\ldots,x_n,y]$ `=  (?`f`. !`x_1 ... x_n`.` $t[x_1,\ldots,x_n,f\ x_1 \ldots\ x_n]$

for any input term `!`x_1 ... x_n`. ?`y`.` $t[x_1,\ldots,x_n,y]$. Note that when $n = 0$, this is equivalent to alpha-conversion:

`|- (?`y`.` $t[y]$`) = (?`f`.` $t[f]$`)`

and that the conversion fails if there is already a free variable f of the appropriate type in the input term. For example:

```
X_SKOLEM_CONV "f:num->*" "!n:num. ?x:*. x = (f n)"
```

will fail. The conversion SKOLEM_CONV is like X_SKOLEM_CONV, except that it uses a primed variant of the name of the existentially quantified variable as the name of the skolem function it introduces. For example:

```
SKOLEM_CONV "!x. ?y. P x y"
```

proves that:

```
|- ?y. !x. P x (y x)
```

13.3.6 Quantifier movement conversions

A complete and systematically-named set of conversions for moving quantifiers inwards and outwards through the logical connectives ~, /\, \/, and ==> is provided. The naming scheme is based on the following atoms:

```
<quant> := FORALL | EXISTS
<conn>  := NOT | AND | OR | IMP
[dir]   := LEFT | RIGHT          (optional)
```

The conversions for moving quantifiers inwards are called:

```
<quant>_<conn>_CONV
```

where the quantifier <*quant*> is to be moved inwards through <*conn*>.

The conversions for moving quantifiers outwards are called:

```
[dir]_<conn>_<quant>_CONV
```

where <*quant*> is to be moved outwards through <*conn*>, and the optional [*dir*] identifies which operand (left or right) contains the quantifier. The complete set is:

```
NOT_FORALL_CONV    |- ~(!x.P) = ?x.~P
NOT_EXISTS_CONV    |- ~(?x.P) = !x.~P
EXISTS_NOT_CONV    |- (?x.~P) = ~!x.P
FORALL_NOT_CONV    |- (!x.~P) = ~?x.P

FORALL_AND_CONV         |- (!x. P /\ Q) = (!x.P) /\ (!x.Q)
AND_FORALL_CONV         |- (!x.P) /\ (!x.Q) = (!x. P /\ Q)
LEFT_AND_FORALL_CONV    |- (!x.P) /\ Q = (!x'. P[x'/x] /\ Q)
RIGHT_AND_FORALL_CONV   |- P /\ (!x.Q) = (!x'. P /\ Q[x'/x])
```

```
EXISTS_OR_CONV          |- (?x. P \/ Q) = (?x.P) \/ (?x.Q)
OR_EXISTS_CONV          |- (?x.P) \/ (?x.Q) = (?x. P \/ Q)
LEFT_OR_EXISTS_CONV     |- (?x.P) \/ Q = (?x'. P[x'/x] \/ Q)
RIGHT_OR_EXISTS_CONV    |- P \/ (?x.Q) = (?x'. P \/ Q[x'/x])


FORALL_OR_CONV
  |- (!x.P \/ Q) = P \/ !x.Q          [x not free in P]
  |- (!x.P \/ Q) = (!x.P) \/ Q        [x not free in Q]
  |- (!x.P \/ Q) = (!x.P) \/ (!x.Q)   [x not free in P or Q]


OR_FORALL_CONV
  |- (!x.P) \/ (!x.Q) = (!x.P \/ Q)   [x not free in P or Q]


LEFT_OR_FORALL_CONV    |- (!x.P) \/ Q = !x'. P[x'/x] \/ Q
RIGHT_OR_FORALL_CONV   |- P \/ (!x.Q)  = !x'. P \/ Q[x'/x]


EXISTS_AND_CONV
  |- (?x.P /\ Q) = P /\ ?x.Q          [x not free in P]
  |- (?x.P /\ Q) = (?x.P) /\ Q        [x not free in Q]
  |- (?x.P /\ Q) = (?x.P) /\ (?x.Q)   [x not free in P or Q]


AND_EXISTS_CONV
  |- (?x.P) /\ (?x.Q) = (?x.P /\ Q)   [x not free in P or Q]


LEFT_AND_EXISTS_CONV    |- (?x.P) /\ Q = ?x'. P[x'/x] /\ Q
RIGHT_AND_EXISTS_CONV   |- P /\ (?x.Q)  = ?x'. P /\ Q[x'/x]


FORALL_IMP_CONV
  |- (!x.P ==> Q) = P ==> !x.Q          [x not free in P]
  |- (!x.P ==> Q) = (?x.P) ==> Q        [x not free in Q]
  |- (!x.P ==> Q) = (?x.P) ==> (!x.Q)   [x not free in P or Q]


LEFT_IMP_FORALL_CONV   |- (!x.P) ==> Q = !x'. P[x/'x] ==> Q
RIGHT_IMP_FORALL_CONV  |- P ==> (!x.Q) = !x'. P ==> Q[x'/x]


EXISTS_IMP_CONV
  |- (?x.P ==> Q) = P ==> ?x.Q          [x not free in P]
  |- (?x.P ==> Q) = (!x.P) ==> Q        [x not free in Q]
  |- (?x.P ==> Q) = (!x.P) ==> (?x.Q)   [x not free in P or Q]


LEFT_IMP_EXISTS_CONV   |- (?x.P) ==> Q = !x'. P[x/'x] ==> Q
RIGHT_IMP_EXISTS_CONV  |- P ==> (?x.Q) = ?x'. P ==> Q[x'/x]
```

13.4 Rewriting tools

The rewriting tool `REWRITE_RULE` was introduced in Chapter 12, but its implementation was not described. All of the various rewriting tools provided in HOL are implemented by use of conversions. Certain new tools could also be built in a similar way.

The rewriting primitive in HOL is `REWRITE_CONV`:

```
REWRITE_CONV : thm -> conv
```

`REWRITE_CONV` (Γ `|-` u=v) t evaluates to a theorem Γ `|-` t=t' if t is an instance (by type and/or variable instantiation) of u and t' is the corresponding instance of v. The first argument to `REWRITE_CONV` can be quantified. Below is an illustration.

1
```
#REWRITE_CONV ADD1 "SUC 0";;
Theorem ADD1 autoloaded from theory `arithmetic`.
ADD1 = |- !m. SUC m = m + 1

|- SUC 0 = 0 + 1
```

All subterms of t can be rewritten according to an equation th using

`DEPTH_CONV(REWRITE_CONV` th`)`

as shown below. The function `"PRE"` is the usual predecessor function.

2
```
#DEPTH_CONV (REWRITE_CONV ADD1) "SUC(SUC 0) = PRE(SUC 2)";;
|- (SUC(SUC 0) = PRE(SUC 2)) = ((0 + 1) + 1 = PRE(2 + 1))
```

In itself, this is not a very useful rewriting tool, but a collection of others have been developed for use in HOL. All of the rewriting rules are, in fact, logically derived, and are based on conversions similar to `DEPTH_CONV`. They have been optimized in various ways, so their implementation is in some cases rather complex and is not given here. The rules and tactics for rewriting all take a list of theorems to be used as rewrites. The theorems in the list need not be in simple equational form (e.g. a conjunction of equations is permissible); but are converted to equational form automatically (and internally). (For example, a conjunction of equations is split into its constituent conjuncts.) There are also a number of standard equations (representing common tautologies) held in the ML variable `basic_rewrites`, and these are used by some of the rewriting tools. All the built-in rewriting tools are listed below, for reference, beginning with the rules. (All are fully described in *REFERENCE*.)

The prefix '`PURE_`' indicates that the built-in equations in `basic_rewrites` are not used, (i.e. only those given explicitly are used). The prefix '`ONCE_`' indicates that the tool makes only one rewriting pass through the expression (this is useful to avoid divergence). It is based on `ONCE_DEPTH_CONV`, while the other tools traverse using `TOP_DEPTH_CONV`. The basic rewriting rules are:

```
REWRITE_RULE                : thm list -> thm -> thm
PURE_REWRITE_RULE           : thm list -> thm -> thm
ONCE_REWRITE_RULE           : thm list -> thm -> thm
PURE_ONCE_REWRITE_RULE      : thm list -> thm -> thm
```

Tactics are introduced in Chapter 14, but are listed here for reference. The tactics corresponding to the above rules are the following:

```
REWRITE_TAC                 : thm list -> tactic
PURE_REWRITE_TAC            : thm list -> tactic
ONCE_REWRITE_TAC            : thm list -> tactic
PURE_ONCE_REWRITE_TAC       : thm list -> tactic
```

The prefix '`ASM_`' indicates that the tactic rewrites using the assumed assumptions of the goal as rewrites.

```
ASM_REWRITE_TAC             : thm list -> tactic
PURE_ASM_REWRITE_TAC        : thm list -> tactic
ONCE_ASM_REWRITE_TAC        : thm list -> tactic
PURE_ONCE_ASM_REWRITE_TAC   : thm list -> tactic
```

The prefix '`FILTER_`' indicates that the tactic only rewrites with those assumptions of the goal satisfying the predicate supplied.

```
FILTER_ASM_REWRITE_TAC      : (term -> bool) -> thm list -> tactic
FILTER_ONCE_ASM_REWRITE_TAC : (term -> bool) -> thm list -> tactic
```

Chapter 14

Goal Directed Proof: Tactics and Tacticals

There are three primary devices that together make theorem proving practical in HOL. All three originate with Milner for Edinburgh LCF. The first is the theory as a record of (among other things) facts already proved and thence available as lemmas without having to be re-proved. The second, the subject of Chapter 12, is the derived rule of inference as a meta-language procedure that implements a broad pattern of inference, but that also, at each application, generates every primitive step of the proof. The third device is the tactic as a means of organizing the construction of proofs; and the use of tacticals for composing tactics.

Even with recourse to derived inference rules, it is still surprisingly awkward to work forward, to find a chain of theorems that culminates in a desired theorem. This is in part because chains have no structure, while 'proof efforts' do. For instance, if within one sequence, two chains of steps are to be combined in the end by conjunction, then one chain must follow or be interspersed with the other in the overall sequence. It can also be difficult to direct the proof toward its object when starting from only hypotheses (if any), lemmas (if any), axioms, and theorems following from no hypotheses (e.g. by `ASSUME` or `REFL`). Likewise, it can be equally difficult to reconstruct the plan of the proof effort after the fact, from the linear sequence of theorems; the sequence is unhelpful as documentation.

The idea of goal directed proof is a simple one, well known in artificial intelligence: to organize the search as a tree, and to reverse the process and *begin* with the objective. The goal is then decomposed, successively if necessary, into what one hopes are more tractable subgoals, each decomposition accompanied by a plan for translating the solution of subgoals into a solution of the goal. The choice of decomposition is an explicit way of expressing a proof 'strategy'.

Thus, for example, instead of the linear sequencing of two branches of the proof of the conjunction, each branch starting from scratch, the proof task is organized as a tree search, starting with a conjunctive goal and decomposing it into the two conjunct subgoals (undertaken in optional order), with the intention of conjoining the two solutions when and if found. The proof itself, as a sequence of steps, is the same however it is found; the difference is in the search, and in the preservation, if required, of the structured proof plan.

The representation of this idea in LCF was Milner's inspiration; the idea is similarly central to theorem proving in HOL. Although subgoaling theorem provers had already been built at the time, Milner's particular contribution was in formalizing the method for translating subgoals solutions to solutions of goals.

14.1 Tactics, goals and justifications

A *tactic* is an ML function that when applied to a *goal* reduces it to (i) a list[1] of (sub)goals, along with (ii) a *justification* function mapping a list of theorems to a theorem. The idea is that the function justifies the decomposition of the goal. A goal is an ML value whose type is isomorphic to, but distinct from, the ML abstract type `thm` of theorems. That is, a goal is a list of terms (*assumptions*) paired with a term. These two components correspond, respectively, to the list of hypotheses and the conclusion of a theorem. The list of assumptions is a working record of facts that may be used in decomposing the goal.

The relation of theorems to goals is achievement: a theorem achieves a goal if the conclusion of the theorem is equal to the term part of the goal (up to α-conversion), and if each hypothesis of the theorem is equal (up to α-conversion, again) to some assumption of the goal. This definition assures that the theorem purporting to satisfy a goal does not depend on assumptions beyond the working assumptions of the goal.

A justification is (rather confusingly) called a *proof* in HOL following the LCF usage; it is, as mentioned, an ML function from a theorem list to a theorem. The ML 'proof' function corresponds to a proof in the logical sense (of a sequence of theorems depending on inference rules) only in that it must evaluate the ML function corresponding to each inference rule on which the sequence depends in order to compute its `thm`-valued result. ('Justification', or 'validation', as is sometimes used, are less confusing terms for the ML function in question.) The proof function, or justification, returned by a tactic is intended to map the list of theorems respectively achieving the subgoals to the theorem achieving the original goal; it justifies the decomposition into subgoals.

A tactic is said to *solve* a goal if it reduces the goal to the empty set of subgoals. This depends, obviously, on there being at least one tactic that maps a goal to the empty subgoal list. The simplest tactic that does this is one that can recognize when a goal is achieved by an axiom or an existing theorem; in HOL, the function `ACCEPT_TAC` does this. `ACCEPT_TAC` takes a theorem *th* and produces a tactic that maps a value of type `thm` to the empty list of subgoals. It justifies this 'decomposition' by a proof function that maps the empty list of theorems to the theorem *th*. The use of this technical device, or other such tactics, ends the decomposition of subgoals, and allows the proof to be built up.

Unlike theorems, goals need not be defined as an abstract type; they are transparent and can be constructed freely. Thus, an ML type abbreviation (see Section 5.4) is introduced

[1] The ordering is necessary for selecting a tree search strategy.

for goals.[2]. The operations on goals are therefore just the ordinary pair selectors and constructor (see Section 6.1). Likewise, type abbreviations are introduced for justifications (proofs) and tactics. Conceptually, the following abbreviations are made in HOL:

```
goal       = term list # term
tactic     = goal -> goal list # proof
proof      = thm list -> thm
```

In fact, the type `goal list # proof` is abbreviated in ML to `subgoals`, and the abbreviation of `tactic` made indirectly through it. Thus, if T is a tactic and g is a goal, then applying T to g (i.e. evaluating the ML expression $T\ g$) results in an ML value of type `subgoals`, i.e. a pair whose first component is a list of goals and whose second component has ML type `proof`. (The word 'tactic' is occasionally used loosely to mean a tactic-valued function.)

It does not follow, of course, from the type `tactic` that a particular tactic is well-behaved. For example, suppose that $T\ g$ `=` `([`g_1`;`$\ldots$`;`g_n`],`p`)`, and that the subgoals g_1 , ..., g_n have been solved. That means that some theorems th_1 , ..., th_n have been proved such that each th_i $(1 \leq i \leq n)$ achieves the goal g_i. The justification p is intended to be a function that when applied to the list `[`th_1`;`$\ldots$`;`th_n`]`, succeeds in returning a theorem, th, achieving the original goal g; but, of course, it might sometimes not succeed. If p succeeds for every list of achieving theorems, then the tactic T is said to be *valid*. This does not guarantee, however, that the subgoals are solvable in the first place. If, in addition to being valid, a tactic always produces solvable subgoals from a solvable goal, it is called *strongly valid*.

Tactics can be perfectly useful without being strongly valid, or without even being valid; in fact, some of the most basic theorem proving strategies, expressed as tactics, are invalid or not strongly valid.[3] An invalid tactic cannot result in the proof of false theorems; theorems in HOL are always the result of performing a proof in the basic logic, whether the proof is found by goal directed search or forward search.[4] However, an invalid tactic may produce an unintended theorem—one that does not achieve the original goal. The typical case is when a theorem purporting to achieve a goal actually depends on hypotheses that extend beyond the assumptions of the goal. The inconvenience to the HOL user in this case is that the problem may be not immediately obvious; the default print format of theorems has hypotheses abbreviated as dots. Invalidity may also be the result of the failure of the proof function, in the ML sense of failure, when applied to a list of theorems (if, for example, the function were defined incorrectly); but again, no

[2]However, if goals were an abstract type, the print abbreviation could be avoided where not intended.

[3]The subgoal package, discussed later in the chapter, prevents the use of invalid tactics when they are liable to result in unexpected theorem results, but the HOL system used directly allows it.

[4]'Invalid' is perhaps a misleading term, since there is nothing logically amiss in the use of invalid tactics or the theorems produced thereby; but the term has stuck over time.

false theorems can result. Likewise, a tactic that is not strongly valid cannot result in a false theorem; the worst outcome of applying such a tactic is the production of unsolvable subgoals.

Tactics are specified using the following notation:

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

For example, the tactic for decomposing conjunctions into two conjunct subgoals is called `CONJ_TAC`. It is described by:

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

This indicates that `CONJ_TAC` reduces a goal of the form $(\Gamma, t_1 \texttt{/\backslash} t_2)$ to subgoals (Γ, t_1) and (Γ, t_2). The fact that the assumptions of the original goal are propagated unchanged to the two subgoals is indicated by the absence of assumptions in the notation. The notation gives no indication of the proof function.

Another example is `INDUCT_TAC`, the tactic for performing mathematical induction on the natural numbers:

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

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

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

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

is the identity tactic; it reduces a goal (Γ, t) to the single subgoal (Γ, t)—i.e. it has no effect. `ALL_TAC` is useful for writing compound tactics, as discussed later (see Section 14.4).

In just the way that the derived rule `REWRITE_RULE` is central to forward proof (Section 12.2), the corresponding function `REWRITE_TAC` is central to goal directed proof. Given a goal and a list of equational theorems, `REWRITE_TAC` transforms the term component of the goal by applying the equations as left-to-right rewrites, recursively and to all

depths, until no more changes can be made. Unless not required, the function includes as rewrites the same standard set of pre-proved tautologies that REWRITE_RULE uses. By use of the tautologies, some subgoals can be solved internally by rewriting, and in that case, an empty list of subgoals is returned. The transformation of the goal is justified in each case by the appropriate chain of inferences. Rewriting often does a large share of the work in goal directed proof searches.

A simple example from list theory (Section 11.9.7) illustrates the use of tactics. A conjunctive goal is declared, and CONJ_TAC applied to it:

1

```
#let g = ([]:term list),"(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])";;
g = ([], "(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])") : goal

#let gl1,p1 = CONJ_TAC g;;
gl1 = [([], "HD[1;2;3] = 1"); ([], "TL[1;2;3] = [2;3]")] : goal list
p1 = - : proof
```

The subgoals are each rewritten, using the definitions of "HD" and "TL":

2

```
#HD;;
Definition HD autoloaded from theory `list`.
HD = |- !h t. HD(CONS h t) = h

|- !h t. HD(CONS h t) = h

#TL;;
Definition TL autoloaded from theory `list`.
TL = |- !h t. TL(CONS h t) = t

|- !h t. TL(CONS h t) = t

#let gl1_1,p1_1 = REWRITE_TAC[HD;TL](hd gl1);;
gl1_1 = [] : goal list
p1_1 = - : proof

#let gl1_2,p1_2 = REWRITE_TAC[HD;TL](hd(tl gl1));;
gl1_2 = [] : goal list
p1_2 = - : proof
```

Both of the two subgoals are now solved, so the decomposition is complete and the proof can be built up in stages. First the theorems achieving the subgoals are proved, then from those, the theorem achieving the original goal:

```
#let th1 = p1_1[];;                                                    3
th1 = |- HD[1;2;3] = 1

#let th2 = p1_2[];;
th2 = |- TL[1;2;3] = [2;3]

#p1[th1;th2];;
|- (HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])
```

Although only the theorems achieving the subgoals are 'seen' here, the proof functions of the three tactic applications together perform the entire chain of inferences leading to the theorem achieving the goal. The same proof could be constructed by forward search, starting from the definitions of "HD" and "TL", but not nearly as easily.

The HOL system provides a collection of pre-defined tactics (and tactic-valued functions) that includes CONJ_TAC, INDUCT_TAC, ALL_TAC and REWRITE_TAC. The pre-defined tactics are adequate for many applications. In addition, there are two means of defining new tactics. Since a tactic is an ML function, the user can define a new tactic directly in ML. Definitions of this sort use ML functions to construct the term part of the subgoals from the term part of the original goal (if any transformation is required); and they specify the justification, which expects a list of theorems achieving the subgoals and returns the theorem achieving (one hopes) the goal. The proof of the theorem is encoded in the definition of the justification function; that is, the means for deriving the desired theorem from the theorems given. This typically involves references to axioms and primitive and defined inference rules, and is usually the more difficult part of the project.

A simple example of a tactic written in ML is afforded by CONJ_TAC, whose definition in HOL is as follows:

```
let CONJ_TAC : tactic (asl,w) =
(let l,r = dest_conj w in
           [(asl,l);(asl,r)],(\[th1;th2].CONJ th1 th2)
) ? failwith `CONJ_TAC`;;
```

This shows how the subgoals are constructed, and how the proof function is specified in terms of the derived rule CONJ (Section 12.3.23).

The second method is to compose existing tactics by the use of ML functions called *tacticals*. The tacticals provided in HOL are listed in Section 14.4. For example, two existing tactics can be sequenced by use of the tactical THEN: if T_1 and T_2 are tactics, then the ML expression T_1 THEN T_2 evaluates to a tactic that first applies T_1 to a goal and then applies T_2 to each subgoal produced by T_1. The tactical THEN is an infixed ML function. Complex and powerful tactics can be constructed in this way; and new tacticals can also be defined, although this is unusual.

The example from earlier is continued, to illustrate the use of the tactical THEN:

```
#let gl2,p2 = (CONJ_TAC THEN REWRITE_TAC[HD;TL])g;;                    4
gl2 = [] : goal list
p2 = - : proof

#p2[];;
|- (HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])
```

The single tactic `CONJ_TAC THEN REWRITE_TAC[HD;TL]` solves the goal in one single application. The chain of inference computed, however, is exactly the same as in the interactive proof; only the search is different.

In general, the second method is both easier and more reliable. It is easier because it does not involve writing ML procedures (usually rather complicated procedures); and more reliable because the composed tactics are valid when the constituent tactics are valid, as a consequence of the way the tacticals are defined. Tactics written directly in ML may fail in a variety of ways, and although, as usual, they cannot cause false theorems to appear, the failures can be difficult to understand and trace.[5] On the other hand, there are some proof strategies that cannot be implemented as compositions of existing tactics, and these have to be implemented directly in ML. Certain sorts of inductions are an example of this; as well as tactics to support some personal styles of proof.

14.1.1 Details of proving theorems

When a theorem is proved that the user wishes to preserve for future use, it can be stored in the current theory by using the function `save_thm` (see Section 11.8.1).

To simplify the use of tactics there are three standard functions

```
TAC_PROOF : (goal # tactic) -> thm
prove_thm : (string # term # tactic) -> thm
PROVE     : (term # tactic) -> thm
```

`TAC_PROOF` takes a goal and a tactic, and applies the tactic to the goal; the goal can have assumptions. Executing `prove_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. Executing `PROVE(`t`,`T`)` proves the goal `([],`t`)` using T and returns the result without saving it. In all cases the evaluation fails if T does not solve the goal `([],`t`)`.

In short, HOL provides a very general framework in which proof strategies can be designed, implemented, applied and tested. Tactics range from the very simple to the very advanced; in theory, a conventional automatic theorem prover could be expressed as a tactic or group of tactics. In contrast, some users never have need to go beyond the built in tactics of the system. The vital support that HOL provides in all cases is the

[5] A possible extension to HOL would be a 'debugging environment' for this class of tactic.

assurance that only theorems of the deductive system can be represented as theorems of the HOL system—security is always preserved.

14.2 The subgoal package

It was mentioned earlier that goal directed proof is a way of organizing the construction of a proof as a tree search. For any tactic and goal, the tactic implicitly determines a tree of subgoals for that goal: each node is a subgoal, and each edge is a tactic. Associated with each node in a successful proof effort is also an achieving theorem—the theorem that achieves that subgoal. That is, the tree is traversed in two phases: from the root (the original goal) to the final layer of subgoals; and from the theorems achieving the final subgoals back to the theorem achieving the goal at the root. The first phase is the decomposition phase, in which goals are reduced to subgoals (and eventually to trivial subgoals). The second phase is the computation of the proof, through each primitive step, culminating in the desired theorem. The tree, however, is not explicitly represented in the HOL system, so each proof effort requires some amount of book-keeping: application of tactics to goals, naming of subgoals and proof functions, application of the appropriate proof functions to theorem lists, naming of theorems, and so on.

When conducting a proof that involves many subgoals and tactics, it is difficult to keep track of this book-keeping. While it is actually feasible for the user to take responsibility, even in large proofs, it is tedious and error-prone. Therefore HOL provides a package traversing the tree of subgoals once through, stacking the subgoals and proof functions, and applying the proof functions automatically, when appropriate to do so. This package was originally implemented for Cambridge LCF by Paulson.

The subgoal package implements a simple framework for interactive proof, and this is adequate for most users in most applications. The tree is traversed depth first. The current goal can be expanded into subgoals and a proof function by supplying it with a tactic; the subgoals are pushed onto a goal stack and the justifications onto a proof stack. Subgoals at the same depth in the tree can be considered in any order by rotating through them, but one otherwise has to work through the tree depth first. When a tactic solves a subgoal (i.e. returns an empty subgoal list), then the package computes a part of the proof, and presents the user with the next subgoal.

For many users, the subgoal package is the primary interface to HOL for proving theorems. As mentioned, it is very convenient to be relieved of all the book-keeping labour. However, there is some cost in that the subgoal-proof tree cannot be inspected; it only exists ephemerally for the user during an interaction, and can only be viewed at the current top subgoal of the stack. Achieving-theorems are only displayed at the moment they are proved; there is no naming or preserving of subgoals or justifications. If there is any reason to view other subgoals, this can only be accomplished by undoing segments of the proof effort (backing up). Likewise, intermediate achieving-theorems cannot be inspected

after they have been displayed at proof time. One situation in which it is necessary to have the tree available is in the debugging of tactics written by the user directly in ML. It is planned in future versions of HOL to implement a more sophisticated subgoal management package.

Finally, the application of certain tactics to certain goals generates a failure in ML where the tactic is invalid; this does not happen when using HOL directly.

The example from earlier is continued below. In the session below, the conjunction proof is generated again, but using the subgoal package, in which a goal is 'set' using the function `set_goal` and 'expanded' using the function `expand`. The side effects of these functions on the subgoal package's stacks can be inferred.

5

```
#set_goal([],"(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])");;
"(HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])"

() : void

#expand CONJ_TAC;;
OK..
2 subgoals
"TL[1;2;3] = [2;3]"

"HD[1;2;3] = 1"

() : void

#expand(REWRITE_TAC[HD;TL]);;
OK..
goal proved
|- HD[1;2;3] = 1

Previous subproof:
"TL[1;2;3] = [2;3]"

() : void

#expand(REWRITE_TAC[HD;TL]);;
OK..
goal proved
|- TL[1;2;3] = [2;3]
|- (HD[1;2;3] = 1) /\ (TL[1;2;3] = [2;3])
```

The following functions are available for interacting with the subgoal package. The function

```
   set_goal: goal -> void
```

initializes the subgoal package with a new goal.

The function

```
expand : tactic -> void
```

applies a tactic to the top goal on the stack, then pushes the resulting subgoals onto the goal stack and prints them. If there are no resulting subgoals (i.e. if the current goal was just solved), then the appropriate proof function is applied to the empty list of theorems and the resulting theorems are printed.

The function

```
backup : void -> void
```

allows backing up from last state-change. The assignable variable `backup_limit`, initially set to `12`, determines the maximum number of proof states saved on the backup list. The function `backup` can be repeated until the list is exhausted; backing up discards the current state irretrievably.

The function

```
rotate : int -> void
```

rotates the order of subgoals on the stack. Calling `rotate` on n rotates by n steps the set of subgoals on top of the stack. This enables subgoals at a given depth in the subgoal tree to be considered in any order. However, subgoals deeper in the stack cannot be worked on, nor can subgoals higher in the tree.

The function

```
top_goal : void -> goal
```

returns the top goal on the stack.

The function

```
top_thm : * -> thm
```

returns the theorem on top of the theorem stack. It is used to access the result of an interactive proof session with the subgoal package.

The function

```
save_top_thm : string -> thm
```

saves the top theorem on the goal stack in the current theory, and also returns it as a value. (It is generally used only to save the final theorem, rather than the intermediate theorems in the proof search.)

The function

```
get_state : void -> goalstack
```

returns the current proof state, which can then be assigned to a variable for additional backup.

The function

```
set_state : goalstack -> void
```

restores the proof state to that saved earlier using `get_state`.

The function

```
print_state : int -> void
```

applied to n, prints n levels of the goal stack.

The following abbreviations are pre-declared for use in the subgoal package:

```
let g t = set_goal([],t)
and e   = expand
and p   = print_state
and b   = backup
and r   = rotate;;
```

The flag `print_all_subgoals` affects all operations where the subgoal stack is printed. If the flag is `true`, the entire subgoal stack is printed. If the flag is `false`, only the top subgoal on the stack is printed. If only the current subgoal is to be printed, the subgoal package will report the number of subgoals remaining before displaying the subgoal on the top of the goal stack. The default value of this flag is `true`.

14.3 Some tactics built into HOL

This section contains a selection of the more commonly used tactics in the HOL system. (see *REFERENCE* for the complete list, with fuller explanations.)

It should be recalled that the ML type `thm_tactic` abbreviates `thm->tactic`, and the type `conv` abbreviates `term->thm`.

14.3.1 Acceptance of a theorem

```
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` (see Section 14.4.5).

14.3.2 Adding an assumption

```
ASSUME_TAC : thm_tactic
```

- **Summary:** `ASSUME_TAC |-`u adds u as an assumption.

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

- **Use:** Enriching the assumptions of a goal with definitions or previously proved theorems.

14.3.3 Specialization

```
GEN_TAC : tactic
```

- **Summary:** Specializes a universally quantified theorem to an arbitrary value.

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

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

- **Use:** Solving universally quantified goals. `GEN_TAC` is often the first step of a goal directed proof. `STRIP_TAC` (see below) applies `GEN_TAC` to universally quantified goals.

14.3.4 Conjunction

```
CONJ_TAC : tactic
```

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

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

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

14.3.5 Discharging an assumption

```
DISCH_TAC : tactic
```

- **Summary:** Moves the antecedant of an implicative goal into the assumptions, leaving the consequent as the term component.

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

- **Use:** Solving goals of the form u `==>` v by assuming u and then solving v under the assumption. `STRIP_TAC` (see below) invokes `DISCH_TAC` on implicative goals.

14.3.6 Combined simple decompositions

```
STRIP_TAC : tactic
```

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

- **Use:** Useful for splitting a goal up into manageable pieces. Often the best thing to do first is `REPEAT STRIP_TAC`, where `REPEAT` is the tactical that repeatedly applies a tactic until it fails (see Section 14.4.7).

14.3.7 Substitution

```
SUBST_TAC : thm list -> tactic
```

- **Summary:** `SUBST_TAC[|-`u_1`=`v_1`;...;|-`u_n`=`v_n`]` changes each sub-term $t[u_1, \ldots, u_n]$ of the goal to $t[v_1, \ldots, v_n]$ by substitution.

- **Use:** Useful in situations where `REWRITE_TAC` does too much, or would loop.

14.3.8 Case analysis on a boolean term

```
ASM_CASES_TAC : term -> tactic
```

- **Summary:** `ASM_CASES_TAC` u , where u is a boolean-valued term, does case analysis on u.

$$\frac{t}{\{u\}t \qquad \{\sim u\}t}$$

- **Use:** Case analysis.

14.3.9 Case analysis on a disjunction

```
DISJ_CASES_TAC : thm_tactic
```

- **Summary:** `DISJ_CASES_TAC |-` u `\/` v splits a goal into two cases: one with u as an assumption and the other with v as an assumption.

$$\frac{t}{\{u\}t \qquad \{v\}t}$$

- **Use:** Case analysis. The tactic `ASM_CASES_TAC` is defined in ML by

```
let ASM_CASES_TAC t = DISJ_CASES_TAC(SPEC t EXCLUDED_MIDDLE)
```

  where `EXCLUDED_MIDDLE` is the theorem `|- !t. t \/ ~t`.

14.3.10 Rewriting

```
REWRITE_TAC : thm list -> tactic
```

- **Summary:** `REWRITE_TAC[`th_1`;`$\ldots$`;`th_n`]` transforms the term part of a goal by rewriting it with the given theorems th_1, ..., th_n, and the set of pre-proved standard tautologies.

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

where t' is obtained from t as described.

- **Use:** Advancing goals by using definitions and previously proved theorems (lemmas).
- **Some other rewriting tactics** (based on `REWRITE_TAC`) are:
  1. `ASM_REWRITE_TAC` adds the assumptions of the goal to the list of theorems used for rewriting.
  2. `PURE_ASM_REWRITE_TAC` is like `ASM_REWRITE_TAC`, but it doesn't use any built-in rewrites.
  3. `PURE_REWRITE_TAC` uses neither the assumptions nor the built-in rewrites.
  4. `FILTER_ASM_REWRITE_TAC` p `[`th_1`;`...`;`th_n`]` simplifies the goal by rewriting it with the explicitly given theorems th_1 , ..., th_n , together with those assumptions of the goal which satisfy the predicate p and also the standard rewrites.

14.3.11 Resolution by Modus Ponens

```
IMP_RES_TAC : thm -> tactic
```

- **Summary:** `IMP_RES_TAC` th does a limited amount of automated theorem proving in the form of forward inference; it 'resolves' the theorem th with the assumptions of the goal and adds any new results to the assumptions. The specification for `IMP_RES_TAC` is:

$$\frac{\{t_1, \ldots, t_m\}t}{\{t_1, \ldots, t_m, u_1, \ldots, u_n\}t}$$

where u_1, ..., u_n are derived by 'resolving' the theorem th with the existing assumptions t_1, ..., t_m. Resolution in HOL is not classical resolution, but just Modus Ponens with one-way pattern matching (not unification) and term and type instantiation. The general case is where th is of the canonical form

`|- !`$x_1 \ldots x_p$`.`v_1 `==>` v_2 `==>` ... `==>` v_q `==>` v

`IMP_RES_TAC` th then tries to specialize x_1, ..., x_p in succession so that v_1, ..., v_q match members of $\{t_1, \ldots, t_m\}$. Each time a match is found for some antecedent v_i, for i successively equal to 1, 2, ..., q, a term and type instantiation is made and the rule of Modus Ponens is applied. If all the antecedents v_i (for $1 \leq i \leq q$) can be dismissed in this way, then the appropriate instance of v is added to the

assumptions. Otherwise, if only some initial sequence $v_1, \ldots, v_k$ (for some k where $1 < k < q$) of the assumptions can be dismissed, then the remaining implication:

`|-` v_{k+1} `==>` $\ldots$ `==>` v_q `==>` v

is added to the assumptions.

For a more detailed description of resolution and `IMP_RES_TAC`, see *REFERENCE*. (See also the Cambridge LCF Manual [16].)

- **Use:** Deriving new results from a previously proved implicative theorem, in combination with the current assumptions, so that subsequent tactics can use these new results.

14.3.12 Identity

```
ALL_TAC : tactic
```

- **Summary:** The identity tactic for the tactical `THEN` (see Section 14.1). Useful for writing tactics.
- **Use:**
  1. Writing tacticals (see description of `REPEAT` in Section 14.4).
  2. With `THENL` (see Section 14.4.5); for example, if tactic T produces two subgoals T_1 is to be applied to the first while nothing is to be done to the second, then T `THENL[`T_1`;ALL_TAC]` is the tactic required.

14.3.13 Null

```
NO_TAC : tactic
```

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

14.3.14 Splitting logical equivalences

```
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.

14.3.15 Solving existential goals

```
EXISTS_TAC : term -> tactic
```

- **Summary:** `EXISTS_TAC "`u`"` reduces an existential goal `!`x`.` $t[x]$ to the subgoal $t[u]$.

$$\frac{!x.t[x]}{t[u]}$$

- **Use:** Proving existential goals.

- **Comment:** `EXISTS_TAC` is a crude way of solving existential goals, but it is the only built-in tactic for this purpose. A more powerful approach uses Prolog-style 'logic variables' (i.e. meta-variables) that can be progressively refined towards the eventual witness. Implementing this requires goals to contain an environment giving the binding of logic variables to terms. Details (in the context of LCF) are given in a paper by Stefan Sokołowski [17].

14.4 Tacticals

A *tactical* is not represented by a single ML type, but is in general an ML function that returns a tactic (or tactics) as result. Tacticals may take parameters, and this is reflected in the variety of ML types that the built-in tacticals have. Tacticals are used for building compound tactics. Some important tacticals in the HOL system are listed below. For a complete list of the tacticals in HOL see *REFERENCE*.

14.4.1 Alternation

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

The tactical `ORELSE` is an ML infix. If T_1 and T_2 are tactics, then the ML expression 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

```
(T1 ORELSE T2) g =  T1 g ? T2 g
```

14.4.2 First success

```
FIRST : tactic list -> tactic
```

The tactical `FIRST` applies the first tactic, in a list of tactics, that succeeds.

`FIRST` $[T_1;T_2;\ldots;T_n]$ = T_1 `ORELSE` T_2 `ORELSE` ... `ORELSE` T_n

14.4.3 Change detection

```
CHANGED_TAC : tactic -> tactic
```

`CHANGED_TAC` T g fails if the subgoals produced by T are just $[g]$; otherwise it is equivalent to T g. It is defined by the following, where `set_equal : * list -> * list -> bool` tests whether two lists denote the same set (i.e. contain the same elements).

```
letrec CHANGED_TAC tac g =
 let gl,p = tac g in
 if set_equal gl [g] then fail else (gl,p)
```

14.4.4 Sequencing

```
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 each subgoal produced by T_1. Its definition in ML is complex (and due to Milner) but worth understanding as an exercise in ML. It is an ML curried infix.

```
let ((T1:tactic) THEN (T2:tactic)) g =
 let gl,p = T1 g
 in
 let gll,pl = split(map T2 gl)
 in
 (flat gll, (p o mapshape(map length gll)pl));;
```

Some of the ML functions `map`, `split`, `o`, `length`, `flat` and `mapshape` were introduced in Chapter 7. Here are their definitions:

```
map : (* -> **) -> * list -> ** list
```

`map` f $[x_1;\ldots;x_n]$ = $[f\ x_1;\ldots;f\ x_n]$

```
split : (* # **) list -> (* list # ** list)
```

```
  split[(x1,y1);...;(xn,yn)]  =  ([x1;...;xn],[y1;...;yn])

$o : ((* -> **) # (*** -> *)) -> *** -> **$ (an infix)

  (f o g) x  =  f(g x)

length : * list -> int

  length[x1;...;xn]  =  n

flat : (* list) list -> * list

  flat[[x11;...;x1m1];[x21;...;x2m2];...;[xn1;...;xnmn]] =
    [x11;...;x1m1;x21;...;x2m2; ... ;xn1;...;xnmn]

mapshape : int list -> (* list -> **) list -> * list -> ** list

  mapshape
    [m1;...;mn]
    [f1;...;fn]
    [x11;...;x1m1;x21;...;x2m2; ... ;xn1;...;xnmn] =
  [f1[x11;...;x1m1];f2[x21;...;x2m2]; ... ;fn[xn1;...;xnmn]]
```

Suppose T_1 `g = (`gl`,`p`)` where gl`=[`g_1`;...;`g_n`]`. Suppose also that for i between 1 and n it is the case that T_2 `g`$_i$ `= ([`g_{i1}`;...;`g_{im_i}`],`p_i`)`. Then `split(map` T_2 gl`)` will evaluate to the pair `(`gll`,`pl`)` of a subgoal list and a proof function, where

gll `= [[`g_{11}`;...;`g_{1m_1}`];[`g_{21}`;...;`g_{2m_2}`]; ... ;[`g_{n1}`;...;`g_{nm_n}`]]`

and pl `= [`p_1`;...;`p_n`]`. Note that

`map length` gll `= [`m_1`;...;`m_n`]`

and that

`flat` gll `= [`g_{11}`;...;`g_{1m_1}`;`g_{21}`;...;`g_{2m_2}`; ... ;`g_{n1}`;...;`g_{nm_n}`]`

Suppose now that, for i between 1 and n, the theorems th_{i1}, ..., th_{im_i} achieve the goals g_{i1}, ..., g_{im_i}, respectively. It will follow that if T_2 is valid then for i between 1 and n the result of applying p_i to the list of theorems `[`th_{i1}`;...;`th_{im_i}`]` will be a theorem, th_i say, which achieves g_i. Now if T_1 is valid then p`[`th_1`;...;`th_n`]` will evaluate to a theorem, th say, that achieves the goal g. Thus

p
(mapshape
 (map length gll)
 pl
 [th_{11};...;th_{1m_1};th_{21};...;th_{2m_2}; ... ;th_{n1};...;th_{nm_n}]) =

p([p_1[th_{11};...;th_{1m_1}];p_2[th_{21};...;th_{2m_2}]; ... ;p_n[th_{n1};...;th_{nm_n}]]) =

p([th_1;...;th_n]) =

th

This shows that p o mapshape(map length gll)pl is a function that, when applied to a list of theorems respectively achieving flat gll, returns a theorem (namely th) that achieves g.

14.4.5 Selective sequencing

```
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 apply different tactics to different subgoals.

Here is the definition of THENL:

```
let ((T:tactic) THENL (Tl:tactic list)) g =
 let gl,p = T g
 in
 let gll,pl = (split(map (\(T,g). T g) Tgl)
                where Tgl = combine(Tl,gl) ? failwith `THENL`)
 in
 (flat gll, (p o mapshape(map length gll)pl))
```

The understanding of this procedure is left as an exercise!

14.4.6 Successive application

```
EVERY : tactic list -> tactic
```

The tactical EVERY applies a list of tactics one after the other.

EVERY [T_1;T_2;...;T_n] = T_1 THEN T_2 THEN ... THEN T_n

14.4.7 Repetition

```
REPEAT : tactic -> tactic
```

If T is a tactic then `REPEAT` T is a tactic that repeatedly applies T until it fails. It is defined in ML by:

```
letrec REPEAT T g = ((T THEN REPEAT T) ORELSE ALL_TAC) g
```

(The extra argument `g` is needed because ML does not use lazy evaluation.)

14.5 Tactics for manipulating assumptions

There are in general two kinds of tactics in HOL: those that transform the conclusion of a goal without affecting the assumptions, and those that do (also or only) affect the assumptions. The various tactics that rewrite are typical of the first class; those that do 'resolution' belong to the second. Often, many of the steps of a proof in HOL are carried out 'behind the scenes' on the assumptions, by tactics of the second sort. A tactic that in some way changes the assumptions must also have a justification that 'knows how' to restore the corresponding hypotheses of the theorem achieving the subgoal. All of this is explicit, and can be examined by a user moving about the subgoal-proof tree.[6] Using these tactics in the most straightforward way, the assumptions at any point in a goal-directed proof, i.e. at any node in the subgoal tree, form an unordered record of every assumption made, but not yet dismissed, up to that point.

In practice, the straightforward use of assumption-changing tactics, with the tools currently provided in HOL, presents at least two difficulties. The first is that assumption sets can grow to an unwieldy size, the number and/or length of terms making them difficult to read. In addition, forward-search tactics such as resolution often add at least some assumptions that are never subsequently used, and these have to be carried along with the useful assumptions; the straightforward method provides no ready way of intercepting their arrival. Likewise, there is no straightforward way of discarding assumptions after they have been used and are merely adding to the clutter. Although perhaps against the straightforward spirit, this is a perfectly valid strategy, and requires no more than a way of denoting the specific assumptions to be discarded. That, however, raises the more general problem of denoting assumptions in the first place. Assumptions are also denoted so that they can be manipulated: given as parameters, combined to draw inferences, etc. The only straightforward way to denote them in the existing system is to supply their quoted text. Though adequate, this method may result in bulky ML expressions; and it may take some effort to present the text correctly (with necessary type information, etc.).

[6] The current subgoal package makes this difficult, but the point still holds.

As always in HOL, there are quite a few ways around the various difficulties. One approach, of course, is the one intended in the original design of Edinburgh LCF, and advocates the rationale for providing a full programming language, ML, rather than a simple proof command set: that is for the user to implement new tactics in ML. For example, resolution tactics can be adapted by the user to add new assumptions more selectively; and case analysis tactics to make direct replacements without adding case assumptions. This, again, is adequate, but can involve the user in extensive amounts of programming, and in debugging exercises for which there is no system support.

Short of implementing new tactics, two other standard approaches are reflected in the current system. Both were originally developed for Cambridge LCF [15, 16]; both reflect fresh views of the assumptions; and both rely on tacticals that transform tactics. The two approaches are partly but not completely complementary.

The first approach, described in this section, implicitly regards the assumption set, already represented as a list, as a stack, with a *pop* operation, so that the assumption at the top of the stack can be (i) discarded and (ii) denoted without explicit quotation. (The corresponding *push* adds new assumptions at the head of the list.) The stack can be generalized to an array to allow for access to arbitrary assumptions.

The other approach, described in Section 14.5.2, gives a way of intercepting and manipulating results without them necessarily being added as assumptions in the first place. The two approaches can be combined in HOL interactions.

14.5.1 Theorem continuations with popping

The first proof style, that of popping assumptions from the assumption 'stack', is illustrated using its main tool: the tactical `POP_ASSUM`.[7]

```
POP_ASSUM : (thm -> tactic) -> tactic
```

Given a function f:`thm -> tactic`, the tactic `POP_ASSUM` f applies f to the (assumed) first assumption of a goal (i.e. to the top element of the assumption stack) and then applies the tactic created thereby to the original goal minus its top assumption:

`POP_ASSUM` f `([`t_1`;...;`t_n`],`t`) = `f `(ASSUME` t_1`) ([`t_2`;...;`t_n`],`t`)`

ML functions such as f, with type `thm -> tactic`, abbreviated to `thm_tactic`, are called theorem continuations, suggesting the fact that they take theorems and then continue the proof.[8] The use of `POP_ASSUM` can be illustrated by applying it to a particular tactic, namely `DISCH_TAC` (Section 14.3.5).

[7] The type of `POP_ASSUM` is actually more general than the type shown here. The present format is used simply for readability.

[8] There is a superficial analogy with continuations in denotational semantics.

```
   DISCH_TAC : tactic
```

On a goal whose conclusion is an implication $u \Rightarrow v$, DISCH_TAC reflects the natural strategy of attempting to prove v under the assumption u, the discharged antecedent. For example, suppose it were required to prove that $(n = 0) \Rightarrow (n \times n = n)$:

```
#g "(n = 0) ==> (n * n = n)";;                                          1
"(n = 0) ==> (n * n = n)"

() : void

#e DISCH_TAC;;
OK..
"n * n = n"
    [ "n = 0" ]
```

Application of DISCH_TAC to the goal produces one subgoal, as shown, with the added assumption. To engage the assumption as a simple substitution, the tactic SUBST1_TAC is useful (see *REFERENCE* for details).

```
   SUBST1_TAC : thm_tactic
```

SUBST1_TAC expects a theorem with an equational conclusion, and substitutes accordingly, into the conclusion of the goal. At this point in the session, the tactical POP_ASSUM is applied to SUBST1_TAC to form a new tactic. The new tactic is applied to the current subgoal.

```
#top_goal();;                                                           2
(["n = 0"], "n * n = n") : goal

#e(POP_ASSUM SUBST1_TAC);;
OK..
"0 * 0 = 0"
```

The result, as shown, is that the assumption is used as a substitution rule and then discarded. The one subgoal therefore has no assumptions on its stack. The two tactics used thus far could be combined into one using the tactical THEN:

```
#g "(n = 0) ==> (n * n = n)";;                                          1
"(n = 0) ==> (n * n = 0)"

() : void

#e(DISCH_TAC THEN POP_ASSUM SUBST1_TAC);;
OK..
"0 * 0 = 0"
```

The goal can now be solved by rewriting with a fact of arithmetic:

```
#e(REWRITE_TAC[MULT_CLAUSES]);;                                          2
Theorem MULT_CLAUSES autoloaded from theory `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))

OK..
goal proved
|- 0 * 0 = 0
|- (n = 0) ==> (n * n = n)
```

A single tactic can, of course, be written to solve the goal:

```
#g "(n = 0) ==> (n * n = n)";;                                           1
"(n = 0) ==> (n * n = n)"

() : void

#e(DISCH_TAC THEN POP_ASSUM SUBST1_TAC THEN REWRITE_TAC[MULT_CLAUSES]);;
Theorem MULT_CLAUSES autoloaded from theory `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))

OK..
goal proved
|- (n = 0) ==> (n * n = n)
```

This example illustrates how the tactical `POP_ASSUM` provides access to the top of the assumption 'stack' (a capability that is useful, obviously, only when the most recently pushed assumption is the very one required). To accomplish this access in the straightforward way would require some more awkward construct, with explicit assumptions:

```
#g "(n = 0) ==> (n * n = n)";;                                          1
"(n = 0) ==> (n * n = n)"

() : void

#e(DISCH_TAC);;
OK..
"n * n = n"
    [ "n = 0" ]

() : void

#e(SUBST1_TAC(ASSUME "n = 0"));;
OK..
"0 * 0 = 0"
    [ "n = 0" ]
```

In contrast to the above, the popping example also illustrates the convenient disappearance of an assumption no longer required, by removing it from the stack at the moment when it is accessed and used. This is valid because any theorem that achieves the subgoal will still achieve the original goal. Discarding assumptions is a separate issue from accessing them; there could, if one liked, be another tactical that produced a similar tactic on a theorem continuation to `POP_ASSUM` but which did not pop the stack.

Finally, `POP_ASSUM` f induces case splits where f does. To prove $(n = 0 \ \vee \ n = 1) \Rightarrow (n \times n = n)$, the function `DISJ_CASES_TAC` can be used. The tactic

`DISJ_CASES_TAC |-` p `\/` q

splits a goal into two subgoals that have p and q, respectively, as new assumptions.

```
#g "((n = 0) \/ (n = 1)) ==> (n * n = n)";;                             1
"(n = 0) \/ (n = 1) ==> (n * n = n)"

() : void

#e DISCH_TAC;;
OK..
"n * n = n"
    [ "(n = 0) \/ (n = 1)" ]

() : void

#backup();;
"(n = 0) \/ (n = 1) ==> (n * n = n)"
```

```
#e(DISCH_TAC THEN POP_ASSUM DISJ_CASES_TAC);;                                  2
OK..
2 subgoals
"n * n = n"
    [ "n = 1" ]

"n * n = n"
    [ "n = 0" ]

() : void

#backup();;
"(n = 0) \/ (n = 1) ==> (n * n = n)"

() : void

#e(DISCH_TAC THEN POP_ASSUM DISJ_CASES_TAC THEN POP_ASSUM SUBST1_TAC);;
OK..
2 subgoals
"1 * 1 = 1"

"0 * 0 = 0"
```

As noted earlier, `POP_ASSUM` is useful when an assumption is required that is still at the top of the stack, as in the examples. However, it is often necessary to access assumptions made at arbitrary previous times, in order to give them as parameters, combine them, etc. The stack approach can be extended to such cases by re-conceiving the stack as an array, and by use of the tactical `ASSUM_LIST`:

```
ASSUM_LIST : (thm list -> tactic ) -> tactic
```

where

`ASSUM_LIST` f `([`t_1`;...;`t_n`],`t`) =` f`([ASSUME` t_1`;...;ASSUME` t_n`])`

That is, given a function f, `ASSUM_LIST` f forms a new tactic by applying f to the list of (assumed) assumptions of a goal, then applies the resulting tactic to the goal. For example, a tactic of the form `ASSUM_LIST (\thl.` f `(el` i `thl))` applies the function f to the ith assumption of a goal to produce a new tactic, then applies the new tactic to the goal. Again, `ASSUM_LIST REWRITE_TAC` is a tactic that engages all of the current assumptions as rewrite rules. In this way, the array approach enables arbitrary assumptions to be accessed; and in particular, specific assumptions to be accessed by location using the function `el`.

To illustrate the use of `ASSUM_LIST`, suppose it were required to prove something different: that $(\forall m.\ m + n = m) \Rightarrow (n \times n = n)$. Suppose also that the arithmetic fact

ADD_INV_0 is already known: namely, that $\forall m\ n.\ (m + n = m) \Rightarrow (n = 0)$. After discharging the assumption, the conclusion of the theorem ADD_INV_0 is imported as an assumption, occupying first place in the array.

1
```
#g "(!m. m + n = m) ==> (n * n = n)";;
"(!m. m + n = m) ==> (n * n = n)"

() : void

#e(DISCH_TAC);;
OK..
"n * n = n"
    [ "!m. m + n = m" ]

() : void

#e(ASSUME_TAC ADD_INV_0);;
Theorem ADD_INV_0 autoloaded from theory `arithmetic`.
ADD_INV_0 = |- !m n. (m + n = m) ==> (n = 0)

OK..
"n * n = n"
    [ "!m. m + n = m" ]
    [ "!m n. (m + n = m) ==> (n = 0)" ]
```

The problem is now to combine the two assumptions to produce the obvious conclusion. That requires denoting them, for which ASSUM_LIST provides the means. Finally, ASSUME_TAC places the conclusion of the new result in the assumptions. (The ML function `el: int -> * list -> *` is used here to select a numbered element of a list.)

2
```
#e(ASSUM_LIST(\thl. ASSUME_TAC
                        (MP (SPECL ["m:num";"n:num"] (el 1 thl))
                            (SPEC "m:num"(el 2 thl)))));;
##OK..
"n * n = n"
    [ "!m. m + n = m" ]
    [ "!m n. (m + n = m) ==> (n = 0)" ]
    [ "n = 0" ]
```

The goal can now be solved as in the previous example.

To access the two particular assumptions in the straightforward way would again require quoting their text. To access all of them (to pass to REWRITE_TAC, for instance) would require quoting all of them.

ASSUM_LIST addresses the issue of accessing assumptions, but not the issue of discarding them. A related function generalizes POP_ASSUM to discard them as well:

```
POP_ASSUM_LIST : (thm list -> tactic ) -> tactic
```

POP_ASSUM_LIST resembles ASSUM_LIST except in removing all of the old assumptions of the subgoal, the way that POP_ASSUM removes the most recent. (Thus POP_ASSUM is no more than a special case of POP_ASSUM_LIST that selects the first element of those supplied and re-assumes the others.)

POP_ASSUM_LIST f ([t_1; ... ;t_n],t) = f [ASSUME t_1; ... ;ASSUME t_n] ([],t)

This is used when the existing assumptions have served their purpose and can be discarded, as in the current example:

3

```
#backup();;
"n * n = n"
    [ "!m. m + n = m" ]
    [ "!m n. (m + n = m) ==> (n = 0)" ]

() : void

#e(POP_ASSUM_LIST(\thl. ASSUME_TAC
                           (MP (SPECL ["m:num";"n:num"] (el 1 thl))
                               (SPEC "m:num"(el 2 thl)))));;
##OK..
"n * n = n"
    [ "n = 0" ]
```

This leaves only the one assumption vital to solving the goal, as before. In some contexts, the new result is required as an assumption, but here it can be used immediately:

4

```
#backup();;
"n * n = n"
    [ "!m. m + n = m" ]
    [ "!m n. (m + n = m) ==> (n = 0)" ]

() : void

#e(POP_ASSUM_LIST(\thl. SUBST1_TAC
                           (MP (SPECL ["m:num";"n:num"] (el 1 thl))
                               (SPEC "m:num"(el 2 thl)))));;
##OK..
"0 * 0 = 0"
```

POP_ASSUM_LIST can, of course, take any function of appropriate type, but is in fact often used in conjunction with the element-selecting functions. Function composition occasionally allows a more compact expression to be written.

The array view (of which the stack view is a special case) gives a way in which unnecessary assumptions can be dropped, and assumptions can be accessed, individually if necessary, using tacticals. Although this approach can be effective, as illustrated, it

does tend to rely on the ordering of the representation of the assumption set. (That is, POP_ASSUM necessarily does, while the other two provide the temptation!) A minor drawback of this reliance is that tactics are then sensitive to changes that alter the order or composition of the assumptions; for example, changes in the implementation of HOL, modifications of existing tactics, and so on. However, that sensitivity is not so serious in any one incarnation of HOL; there is a logical viewpoint that regards the assumptions (sequents) as ordered anyway. A more serious problem is that order-sensitive tactics are meaningful only during interactive sessions; to reconstruct the assumptions from the ML text and the original goal alone is generally difficult, and more so when assumptions are denoted by location. This means that (i) the resulting tactics cannot easily be generalized for use in other contexts, and (ii) the ML text does not supply useful documentation of the solution of the goal. Also, as shown in the last example, it it slightly unsatisfactory to push and subsequently pop assumptions, especially in immediate succession, where this could be avoided.

Two other tacticals that can be used to manipulate the assumption list are FIRST_ASSUM and EVERY_ASSUM. These are characterized by:

```
FIRST_ASSUM f ([t1; ... ;tn], t)  =
 (f(ASSUME t1) ORELSE ... ORELSE f(ASSUME tn)) ([t1; ... ;tn], t)

EVERY_ASSUM f ([t1; ... ;tn], t)  =
 (f(ASSUME t1) THEN ... THEN  f(ASSUME tn)) ([t1; ... ;tn], t)
```

14.5.2 Theorem continuations without popping

The idea of the second approach is suggested by the way the array-style tacticals supply a list of theorems (the assumed assumptions) to a function. These tacticals use the function to infer new results from the list of theorems, and then to do something with the results. In some cases, e.g. the last example, the assumptions need never have been made in the first place, which suggests a different use of tacticals. The original example for POP_ASSUM illustrates this: namely, to show that $(n = 0) \Rightarrow (n \times n = n)$. Here, instead of discharging the antecedent by applying DISCH_TAC to the goal, which adds the antecedent as an assumption and returns the consequent as the conclusion, and *then* supplying the (assumed) added assumption to the theorem continuation SUBST1_TAC and discarding it at the same time, a tactical called DISCH_THEN is applied to SUBST1_TAC directly. DISCH_THEN transforms SUBST1_TAC into a new tactic: one that applies SUBST1_TAC directly to the (assumed) antecedent, and the resulting tactic to a subgoal with no new assumptions and the consequent as its conclusion:

```
#DISCH_THEN;;                                                          1
- : (thm_tactic -> tactic)

#DISCH_THEN SUBST1_TAC;;
- : tactic

#g "(n = 0) ==> (n * n = n)";;
"(n = 0) ==> (n * n = n)"

() : void

#e(DISCH_THEN SUBST1_TAC);;
OK..
"0 * 0 = 0"
```

This gives the same result as the stack method, but more directly, with a more compact ML expression, and with the attractive feature that the term $n = 0$ is never an assumption, even for an interval of one step. This technique is often used at the moment when results are available; as above, where the result produced by discharging the antecedent can be immediately passed to substitution. If the result were only needed later, it *would* have to be held as an assumption. However, results can be manipulated when they are available, and their results either held as assumptions or used immediately. For example, to prove $(0 = n) \Rightarrow (n \times n = n)$, the result $n = 0$ could be reversed immediately:

```
#g "(0 = n) ==> (n * n = n)";;                                         1
"(0 = n) ==> (n * n = n)"

() : void

#e(DISCH_THEN(SUBST1_TAC o SYM));;
OK..
"0 * 0 = 0"
```

The justification of `DISCH_THEN SUBST1_TAC` is easily constructed from the justification of `DISCH_TAC` composed with the justification of `SUBST1_TAC`. The term $n = 0$ is assumed, to yield the theorem that is passed to the theorem continuation `SUBST1_TAC`, and it is accordingly discharged during the construction of the actual proof; but the assumption happens only internally to the tactic `DISCH_THEN SUBST1_TAC`, and not as a step in the tactical proof. In other words, the subgoal tree here has one node fewer than before, when an explicit step (`DISCH_TAC`) reflected the assumption.

On the goal with the disjunctive antecedent, this method again provides a compact tactic:

```
#g "((n = 0) \/ (n = 1)) ==> (n * n = n)";;                                    1
"(n = 0) \/ (n = 1) ==> (n * n = n)"

() : void

#e(DISCH_THEN(DISJ_CASES_THEN SUBST1_TAC));;
OK..
2 subgoals
"1 * 1 = 1"

"0 * 0 = 0"
```

This avoids the repeated popping and pushing of the stack solution, and likewise, gives a shorter ML expression. Both give a shorter expression than the direct method, which is:

```
DISCH_TAC
 THEN DISJ_CASES_TAC(ASSUME "(n = 0) \/ (n = 1)")
 THENL[SUBST1_TAC(ASSUME "n = 0");
       SUBST1_TAC(ASSUME "n = 1")]
```

To summarize, there are so far at least five ways to solve a goal (and these are often combined in one interaction): directly, using the stack view of the assumptions, using the array view with or without discarding assumptions, and using a tactical to intercept an assumption step. All of the following work on the goal $(n = 0) \Rightarrow (n \times n = n)$:

```
DISCH_TAC
 THEN SUBST1_TAC(ASSUME "n = 0")
 THEN REWRITE_TAC[MULT_CLAUSES]

DISCH_TAC
 THEN POP_ASSUM SUBST1_TAC
 THEN REWRITE_TAC[MULT_CLAUSES]

DISCH_TAC
 THEN ASSUM_LIST (SUBST1_TAC o el 1)
 THEN REWRITE_TAC[MULT_CLAUSES]

DISCH_TAC
 THEN POP_ASSUM_LIST (SUBST1_TAC o el 1)
 THEN REWRITE_TAC[MULT_CLAUSES]

DISCH_THEN SUBST1_TAC
 THEN REWRITE_TAC[MULT_CLAUSES]
```

Furthermore, all five induce the same sequence of inferences leading to the desired theorem; internally, no inference steps are saved by the economies in the ML text or the subgoal tree. In this sense, the choice is entirely one of style and taste; of how to organize

the decomposition into subgoals. The first expression illustrates the verbosity of denoting assumptions by text (the goal with the disjunctive antecedent gave a clearer example); but also the intelligibility of the resulting expression, which, of course, is all that is saved of the interaction, aside from the final theorem. The last expression illustrates both the elegance and the inscrutibility of using functions to manipulate intermediate results directly, rather than as assumptions. The middle three expressions show how results can be used as assumptions (discarded when redundant, if desired); and how assumptions can be denoted without recourse to their text. It is a strength of the LCF approach to theorem proving that many different proof styles are supported, (all in a secure way) and indeed, can be studied in their own right.

HOL provides several other theorem continuation functions analogous to `DISCH_THEN` and `DISJ_CASES_THEN`. (Their names always end with '`_THEN`', '`_THENL` or '`_THEN2`'.) Some of these do convenient inferences for the user. For example:

```
CHOOSE_THEN : thm_tactical
```

Where `thm_tactical` abbreviates `thm_tactic -> tactic`. `CHOOSE_THEN` f `(|- ?`x`.`$t[x]$`)` is a tactic that, given a goal, generates the subgoal obtained by applying f to `(`$t[x]$`|-`$t[x]$`)`. The intuition is that if `|- ?`x`.`$t[x]$ holds then `|-` $t[x]$ holds for some value of x (as long as the variable x is not free elsewhere in the theorem or current goal). This gives an easy way of using existentially quantified theorems, something that is otherwise awkward.

The new method has other applications as well, including as an implementation technique. For example, taking `DISJ_CASES_THEN` as basic, `DISJ_CASES_TAC` can be defined by:

```
let DISJ_CASES_TAC = DISJ_CASES_THEN ASSUME_TAC
```

Similarly, the method is useful for modifying existing tactics (e.g. resolution tactics) without having to re-program them in ML. This avoids the danger of introducing tactics whose justifications may fail, a particularly difficult problem to track down; it is also much easier than starting from scratch.

The main theorem continuation functions in the system are:

```
ANTE_RES_THEN
CHOOSE_THEN       X_CHOOSE_THEN
CONJUNCTS_THEN    CONJUNCTS_THEN2
DISJ_CASES_THEN   DISJ_CASES_THEN2   DISJ_CASES_THENL
DISCH_THEN
IMP_RES_THEN
RES_THEN
STRIP_THM_THEN
STRIP_GOAL_THEN
```

See *REFERENCE* for full details. For `INDUCT_THEN`,
see Section 11.10.4 and *REFERENCE*.

References

[1] P.B. Andrews, *An Introduction to Mathematical Logic and Type Theory: to Truth through Proof*, Computer Science and Applied Mathematics Series (Academic Press, 1986).

[2] 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.

[3] A. Church, 'A Formulation of the Simple Theory of Types', *Journal of Symbolic Logic* Vol. 5 (1940), pp. 56–68.

[4] T. Coquand, 'An Analysis of Girard's Paradox', in: *Proceedings of the ACM Conference on Logic in Computer Science*, Boston, June, 1986.

[5] G. Cousineau, M. Gordon, G. Huet, R. Milner, L. Paulson and C. Wadsworth, *The ML Handbook* (INRIA, 1986).

[6] J.A. Goguen, J.W. Thatcher, and E.G. Wagner, 'An initial algebra approach to the specification, correctness, and implementation of abstract data types', in: *Current Trends in Programming Methodology* edited by R.T. Yeh (Prentice-Hall, 1978) Vol. IV, pp 80–149.

[7] M. Gordon, 'HOL: A Machine Oriented Formulation of Higher-Order Logic', Technical Report No. 68, University of Cambridge Computer Laboratory, revised version (July 1985).

[8] 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.

[9] M. Gordon, R. Milner and C.P. Wadsworth, 'Edinburgh LCF: A Mechanised Logic of Computation', Lecture Notes in Computer Science, Vol. 78, (Springer-Verlag, 1979).

[10] F. K. Hanna and N. Daeche, 'Specification and Verification Using Higher-Order Logic: A Case Study', 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. 179–213.

[11] R. Milner, M. Tofte and R. Harper, *The Definition of Standard ML*, The MIT Press, 1990.

[12] A. Leisenring, *Mathematical Logic and Hilbert's ϵ-Symbol*, University Mathematical Series, (Macdonald & Co. Ltd., London, 1969).

[13] Z. Manna and R. Waldinger, *The Logical Basis for Computer Programming* (Addison-Wesley, 1985).

[14] T.F. Melham, 'Automating Recursive Type Definitions in Higher Order Logic', in: *Current Trends in Hardware Verification and Automated Theorem Proving*, edited by G. Birtwistle and P.A. Subrahmanyam (Springer-Verlag, 1989), pp. 341–386.

[15] L. Paulson, 'A Higher-Order Implementation of Rewriting', *Science of Computer Programming*, Vol. 3, (1983), pp. 119–149.

[16] L. Paulson, *Logic and Computation: Interactive Proof with Cambridge LCF*, Cambridge Tracts in Theoretical Computer Science 2 (Cambridge University Press, 1987).

[17] S. Sokołowski, 'A note on tactics in LCF', Technical Report CSR-140-83, Department of Computer Science, University of Edinburgh, 1983.

[18] M. Spivey, *The Z Notation*, (Prentice-Hall, 1989).

Appendix A

Version 2.0

This appendix summarizes the differences between HOL88 Version 2.0 and the last major release, which was Version 1.11 (Version 1.12 is essentially the Version 2.0 system with the Version 1.10 documentation).

Version 2.0 is primarily a rationalization of Version 1.11. Relics of LCF have been deleted and many 'internal' ML bindings have been made local to the functions that use them. It is hoped that nothing useful has been inadvertently removed.

The top-level directory of the HOL distribution sources has been reorganized. The details are in Chapter 1 of *TUTORIAL* ('Getting and Installing HOL'). An important addition is a directory `contrib` containing contributions from the HOL user community. The aim of this directory is to make it easy for the community to share theories, proofs, examples, tools, documents, and other material which may be of general interest. Unlike the library, there is no quality control on `contrib`.

Two major revisions of the theorem-proving tools in Version 2.0 are the complete reimplementations, by Tom Melham, of (i) the resolution rules and tactics and (ii) the type definition package.

The resolution tools have been rewritten so that they are more systematic and predictable. Unfortunately, this means that sometimes different results are produced, and so code using `RES_TAC`, `IMP_RES_TAC` etc. may need to be modified to run in Version 2.0. This is particularly likely if tactics have been used that depend on the positions of assumptions in goals (this is generally considered 'bad style'). The old resolution tools are available in the directory `contrib/resolve`.

The type definition package (i.e. the ML function `define_type`) has been rewritten to be faster and more robust. A number of identifier names have changed and its internal workings have been reorganized. A tool for mutually recursive type definitions is available in `contrib/mut_rec_types` (it was produced by students at Aarhus University).

Four libraries from Version 1.11 of HOL have been temporarily withdrawn, because the Cambridge group have been unable to rebuild them in Version 2.0. These are:

```
card   well_order   zet   csp
```

The first three are difficult to rebuild because of changes to resolution. They rely very heavily on the order in which the assumptions of goals appear. The fourth library uses the `set` library, which has been substantially modified. These four libraries have been moved to `contrib` until they are updated by their authors.

Version 2.0 provides parser and pretty-printer support for restricted quantification and set-theoretic notation.

The restricted quantification notation allows terms of the form $\mathsf{Q}x\mathtt{::}p.t[x]$, where Q is a quantifier and if $x : \alpha$ then p can be any term of type $\alpha \rightarrow bool$; this denotes the quantification of x over those values satisfying p. The qualifier `::` can be used with `\` and any binder, including user defined ones; the appropriate meanings are predefined for `\` and the built-in binders `!`, `?` and `@`. This provides a method of simulating subtypes and dependent types; the qualifying predicate p can be an arbitrary term containing parameters. For example: $\mathtt{!}w\mathtt{::}\mathsf{Word}(n)\mathtt{.}\ t[w]$, for a suitable constant Word, simulates a quantification over the 'type' of n-bit words. Experiments are needed to establish how satisfactory this approach is. The syntactic mechanism, although implemented initially to support restricted quantification, can also be used to support better approximations to notations like $\sum_{i=1}^{n} x_i$; this could, for example, be represented by $\mathsf{Sum}\ i\mathtt{::(1,}n\mathtt{).}\ x_i$, via a suitable definition of the constant Sum.

The set notation allows finite sets to be written in the form "$\{t_1, t_2, \ldots, t_n\}$". It also supports set abstraction notation of the form "$\{t|p\}$"; for example "`{2*x|x>0}`" and "`{(x,y)|x=y}`". Both these notations can be mounted onto any underlying set theory (or even onto theories of lists, bags etc.), although they were initially designed for use with the library `sets` (which was previously called `all_sets`). This library has been greatly extended.

Version 2.0 has libraries `parser` and `prettyp`, which provide parser and pretty-printer support for embedded languages. These are currently being used to provide theorem-proving tools for the hardware description languages ELLA, VHDL and SILAGE, and for various other special purpose languages. A mechanism called "syntax blocks" allows user-specified parsers to be invoked from ML. The development of methodologies for embedding application-specific languages in HOL is a major topic of research at Cambridge.

A facility is provided to support experiments with user-programmed quotation type-checkers for the HOL logic. This enables more type inference to be implemented.

Many new conversions are provided. The most useful ones are described in Section A.9 below.

In the sections that follow, those new things in Version 2.0 which are likely to be of most general use are described. Section A.11 list the ML bindings that have ben added and deleted in Version 2.0.

A.1 Directory reorganization

The help files and manual have been split apart into two separate directories. The top-level of the HOL distribution directory now contains:

- `Manual`: directory containing the manual sources;

- `help`: directory containing text files for online help, as well as for the *REFERENCE* part of the manual.

The search path mechanism has been revised in various ways.

Library pathnames are now no longer on the default search path, nor are they added to the search path when a library is loaded (it is the responsibility of the library's load-file to do this, if desired). As a consequence, the predefined ML value `Libraries` is no longer used.

The initial search path is now:

```
[``; `~/`; `directory/theories/`]
```

where *directory* is the site-specific absolute pathname in which the HOL distribution directory (`hol`) resides.

The ML functions `add_to_search_path` and `append_to_search_path` are no longer built-in. Users may define them, if required, by:

```
let add_to_search_path p = set_search_path(p.search_path())

let append_to_search_path p = set_search_path(search_path()@[p])
```

The `install` function now sets the search path to:

```
[``; `~/`; arg^`/theories/`]
```

where *arg* is the string argument to install.

A.2 Location of libraries

The function

```
library_pathname : void -> string
```

returns the internal pathname used by HOL to load libraries. This pathname, which is site-specific and is given an initial value when the system is built, should be the absolute pathname of the HOL system library directory. This pathname will typically have the form:

```
`directory/hol/Library`
```

where *directory* is the site-specific absolute pathname in which the HOL distribution directory ('hol') resides. The value returned by `library_pathname` can be changed by users only via the `install` function.

The string returned by `library_pathname` is primarily used in library load-files to update the HOL search path and help search path. For example, suppose that in a library `lib` there is a directory `help` which contains online help files specific to this library. The load-file `lib.ml` can then update the help search path as follows:

```
let path = library_pathname() ^ `/lib/help/`
in
set_help_search_path (path . help_search_path())
```

This will make the help files of the library `lib` available for online help whenever the library is loaded.

A.3 More flexible help system

The help system now uses `cat` rather than `more` as the default for displaying help files. This default can be changed with the ML function:

```
set_help : string -> string
```

This installs a new user-supplied help function, and returns the previous one as result. The effect of

```
help `file`
```

is to pipe the appropriate help file derived from *file*`.doc` into the current help function, with the top level of ML being the standard output. For example,

```
set_help `lpr`
```

will cause the help file to be printed instead of being displayed and

```
set_help `lpr -Pbarf`
```

will cause it to be printed to the printer `barf`.

Two new functions:

```
set_help_search_path : string list -> void
help_search_path     : void -> string list
```

have been added to the system for accessing the internal search path used by HOL to find online help files. The help search path has precisely the same format as the regular search path, and these two function are analogous to the ML functions `search_path` and `set_search_path`.

A.4 Syntax for restricted quantification

Syntactic support for restricted quantification and abstraction is now provided. This follows a suggestion discussed at the Second HOL Users Meeting and implements a method of simulating subtypes and dependent types with predicates.

Currently no derived rules are provided to support this notation, so any inferences will need to work on the underlying semantic representation.

The new syntax automatically translates as follows:

```
\v::P.B    <---->   RES_ABSTRACT P (\v.B)
!v::P.B    <---->   RES_FORALL   P (\v.B)
?v::P.B    <---->   RES_EXISTS   P (\v.B)
@v::P.B    <---->   RES_SELECT   P (\v.B)
```

Anything can be written between the binder and '::' that could be written between the binder and '.' in the old notation. See the examples below.

The flag `print_restrict` has default `true`, but if set to `false` will disable the pretty printing. This is useful for seeing what the semantics of particular restricted abstractions are.

The constants `RES_ABSTRACT`, `RES_FORALL`, `RES_EXISTS` and `RES_SELECT` are defined in the theory `bool` by:

```
RES_ABSTRACT P B  =  \x:*. (P x => B x | ARB:**)

RES_FORALL   P B  =  !x:*. P x ==> B x

RES_EXISTS   P B  =  ?x:*. P x /\ B x

RES_SELECT   P B  =  @x:*. P x /\ B x
```

where the constant `ARB` is defined in the theory `bool` by:

```
ARB  =  @x:*. T
```

User-defined binders can also have restricted forms, which are set up with the function:

```
associate_restriction : (string # string) -> *
```

If B is the name of a binder and `RES_`B is the name of a suitable constant (which must be explicitly defined), then executing:

```
associate_restriction('B', 'RES_B')
```

will cause the parser and pretty-printer to support:

B v::P. B `<---->` `RES_`*B P* `(\`*v. B*`)`

Note that associations between user defined binders and their restrictions are not stored in theory files, so they have to be set up for each HOL session (e.g. with a `hol-init.ml` initialization file).

Here is an example session:

```
#"!x y::P. x<y";;                                                        1
"!x y :: P. x < y" : term

#set_flag(‘print_restrict‘, false);;
true : bool

#"!x y::P. x<y";;
"RES_FORALL P(\x. RES_FORALL P(\y. x < y))" : term

#"?(x,y) p::(\(m,n).m<n). p=(x,y)";;
"RES_EXISTS
 (\(m,n). m < n)
 (\(x,y). RES_EXISTS(\(m,n). m < n)(\p. p = x,y))"
: term

#"\x y z::P.[0;x;y;z]";;
"RES_ABSTRACT P(\x. RES_ABSTRACT P(\y. RES_ABSTRACT P(\z. [0;x;y;z])))"
: term
```

A conversion that rewrites away the constants `RES_ABSTRACT`, `RES_FORALL`, `RES_EXISTS` and `RES_SELECT` is:

```
let RESTRICT_CONV =
 DEPTH_CONV
  (REWRITE_CONV (definition ‘bool‘ ‘RES_ABSTRACT‘) ORELSEC
   REWRITE_CONV (definition ‘bool‘ ‘RES_FORALL‘)   ORELSEC
   REWRITE_CONV (definition ‘bool‘ ‘RES_EXISTS‘)   ORELSEC
   REWRITE_CONV (definition ‘bool‘ ‘RES_SELECT‘))
 THENC DEPTH_CONV BETA_CONV
```

This is a bit unsatisfactory, as is shown by the artificial example below:

```
#let t = "!x y::P.?f:num->num::Q. f(@n::R.T) = (x+y)";;                  1
t = "!x y :: P. ?f :: Q. f(@n :: R. T) = x + y" : term

#RESTRICT_CONV t;;
|- (!x y :: P. ?f :: Q. f(@n :: R. T) = x + y) =
   (!x. P x ==> (!x'. P x' ==> (?x. Q x /\ (x(@x. R x /\ T) = x + x'))))
```

The variable `x` in the definitions of the constants `RES_ABSTRACT`, `RES_FORALL`, `RES_EXISTS` and `RES_SELECT` gets confused with the variable in the supplied term. This can be avoided by changing `RESTRICT_CONV` to perform explicit alpha-conversion. For example, by implementing a conversion:

RES_FORALL P (\v.$B[v]$) ----> !v. P v ==> $B[v]$

Dealing with the case when v is a variable structure is also desirable. For example:

```
#let t1 = "!(m,n)::P. m<n";;                                              2
t1 = "!(m,n) :: P. m < n" : term

#RESTRICT_CONV t1;;
|- (!(m,n) :: P. m < n) = (!x. P x ==> (\(m,n). m < n)x)
```

If anyone writes the desired conversions please let us know!

Here is an example of a user-defined restriction:

```
#new_binder_definition(`DURING`, "DURING(p:num#num->bool) = $!p");;       3
|- !p. $DURING p = $! p

#"DURING x::(m,n). p x";;
no restriction constant associated with DURING
skipping: x " ;; parse failed

#new_definition
# (`RES_DURING`, "RES_DURING(m,n)p = !x. m<=x /\ x<=n ==> p x");;
|- !m n p. RES_DURING(m,n)p = (!x. m <= x /\ x <= n ==> p x)

#associate_restriction(`DURING`,`RES_DURING`);;
() : void

#"DURING x::(m,n). p x";;
"DURING x :: (m,n). p x" : term

#set_flag(`print_restrict`,false);;
true : bool

#"DURING x::(m,n). p x";;
"RES_DURING(m,n)(\x. p x)" : term
```

A.5 Syntax for sets

The special purpose set-theoretic notations "{$t_1, t_2, \ldots, t_n$}" and "{$t|p$}" are now available. The normal interpretation of the former is the finite set containing $t_1, t_2, \ldots, t_n$ and the normal interpretation of the latter is the set of all ts such that p. These interpretations are predefined for the library `sets`, but the user can use the syntax for other purposes if he or she wishes using the functions:

```
define_finite_set_syntax      : (string # string) -> void
define_set_abstraction_syntax : string -> void
```

Executing:

```
define_finite_set_syntax(‘c1‘,‘c2‘)
```

causes "{$t_1, \ldots, t_n$}" to parse to:

"$c_1\ t_1\ (c_1\ t_2\ \cdots\ (c_1\ t_n\ c_2)\ \cdots\ ))$"

with failure if either c_1 or c_2 is not the name of a constant.

In the library `sets`, the empty set is `EMPTY` and the infixed function `INSERT` adds an element to a set. Executing:

```
define_finite_set_syntax(‘EMPTY‘,‘INSERT‘)
```

will cause

```
"{1,2,3,4}"
```

to parse to

```
"1 INSERT (2 INSERT (3 INSERT (4 INSERT EMPTY)))"
```

Executing:

```
define_set_abstraction_syntax ‘c‘
```

causes "{t|p}" to parse to:

"c(\(x_1,...,x_n).(t,p))"

where $x_1, \ldots, x_n$ are the free variables occurring in both t and p. If there are no such free variables then an error results. The order in which the variables are listed in the variable structure of the paired abstraction is an unspecified function of the structure of t (it is approximately left to right). Failure if c is not the name of a constant.

For example, if the library `sets` (i.e. what used to be `all_sets`) is loaded, then

```
define_set_abstraction_syntax ‘GSPEC‘
```

will cause

```
"{x+y | (x < y) /\ (y < z)}"
```

to parse to:

```
"GSPEC(\(x,y). ((x+y), (x < y) /\ (y < z)))"
```

where `GSPEC` is defined by:

```
|- !f. GSPEC f = SPEC(\x. ?y. x,T = f y)
```

and `SPEC` abstracts a predicate to a set (it is the abstraction bijection used in the definition of the type operator `set`). Other examples are:

```
"{x+y+z | (x < y) /\ (y < z)}"
```

will parse to:

```
"GSPEC(\(x,y,z). (x+(y+z), (x < y /\ y < z)))"
```

and

```
"{x+y+w | (x < y) /\ (y < z)}"
```

will parse to:

```
"GSPEC(\(x,y). (x+(y+w), (x < y /\ y < z)))"
```

Note that the precedence of comma is increased in the contexts "`{`$\cdots$`}`" and "`{`$\cdots$`|`". Terms will be printed in set notation if the flag `print_set` is `true`. Note that

"c`(\(`x_1`,...,`x_n`).(`t`,`p`))"

will only print as "`{`t`|`p`}`" if the variables x_1, ... , x_n occur free in both t and p (and `print_set` is `true`) .

A.6 Revised set theory libraries

In previous versions of the system, there were three libraries dealing with elementary set theory:

(i) `sets`: a theory of finite sets (Windley and Leveilley);

(ii) `all_sets`: a theory of infinite and finite sets (Windley and Leveilley);

(iii) `set`: a theory of predicates-as-sets (Kalker).

Each of these libraries is useful for certain applications, and all three will therefore continue to be supported. To better reflect the contents of these libraries, they have been renamed as follows:

(i) `sets` is now called `finite_sets`;

(ii) `all_sets` is now called `sets`;

(iii) `set` is now called `pred_sets`.

The `sets` library (formerly, `all_sets`) has been extensively revised and extended. The library has been restructured, and several additional built-in theorems are available. Parser and pretty-printer support for the notation:

- $\{x_1, \ldots, x_n\}$ denotes a finite set.
- $\{x \mid P[x]\}$ denotes the set of all x such that $P[x]$.
- $\{t[x] \mid P[x]\}$ denotes the set of all $t[x]$ such that $P[x]$.

is now provided by the parser/pretty-printer extension mentioned above.

Proof support is supplied for this notation in the form of a conversion

```
SET_SPEC_CONV : conv
```

which implements the axiom of specification for generalized set specifications.

`SET_SPEC_CONV` accepts two types of input.

`SET_SPEC_CONV "`t `IN {`v `|` $p[v]$`}"` (where v is a variable)

this returns the theorem:

`|-` t `IN` $\{v \mid p[v]\}$ `=` $p[t/v]$

Furthermore

`SET_SPEC_CONV "`t_1 `IN` $\{t_2[x_1, \ldots, x_n] \mid p[x_1 \ldots, x_n]\}$`"`

returns the theorem

`|-` t_1 `IN` $\{t_2[x_1, \ldots, x_n] \mid p[x_1, \ldots, x_n]\}$
`=`
`?`$x_1 \cdots x_n$`.` $(t_1 = t_2[x_1, \ldots, x_n])$ `/\` $p[x_1, \ldots, x_n]$

```
|- x IN {n | n < m}  =  x < m

|- 5 IN {x+y | x < 2 /\ y < 3}  =  ?x y. (5 = x + y) /\ x < 2 /\ y < 3
```

The library `sets` also makes available an induction principle for proving properties of finite sets.

```
SET_INDUCT_TAC : tactic
```

```
 "!s. FINITE s ==>  P[s]"
============================     SET_INDUCT_TAC
 P[EMPTY]   P[x INSERT t]
              [ "FINITE t" ]
              [ "P[s]"]
              [ "~x IN t"]
```

The file `Libaries/sets/COMPAT.READ.ME` provides an index to theorem names in the new library and other compatibility information. This is intended to help users update proof scripts based on the old version of the `all_sets` library.

A.7 Syntax blocks

A syntax block starts with a keyword and ends with a terminator and is associated with a function on strings. When such a block is encountered in the input stream, all the characters between the start keyword and the terminator are made into a string to which the associated function is applied. Syntax blocks can be used with the parser generator library `parser` to parse user-specified languages into HOL terms. See the documentation of the library for details and examples.

The ML function:

```
new_syntax_block : string # string # string -> void
```

declares a new syntax block. The first argument is the start keyword of the block, the second argument is the terminator and the third argument is the name of a function having a type which is an instance of `string->*`. The effect of

```
new_syntax_block(`XXX`, `YYY`, `f`);;
```

is that if subsequently

```
XXX   ...   YYY
```

occurs in the input, then it is as though

```
f ` ... `
```

were input instead. For example:

```
#let foo s = print_string `Hello: `; print_string s; print_newline();;
foo = - : (string -> void)

#new_syntax_block(`<<`,`>>`, `foo`);;
() : void

#<< Mike >>;;
Hello: Mike
() : void
```

The following function is useful with syntax blocks, because it enables backslash (\) to be included in strings.

```
set_string_escape : int -> int
```

This changes the escape character in strings to be the character with the supplied ascii code (initially this is `92`, viz. '\'). The old escape character is returned.

A.8 User-programmable quotation typechecker

The typechecker for HOL quotations contains a number of arbitrary design decisions. Several people have suggested changes, e.g. that full Hindley/Milner type inference be performed. Rather than try to create a single new improved typechecker, a facility is now provided that enables the user to write his or her own one and then to install it in the system.

The ML abstract type `preterm` represents the parse trees of HOL terms. A typechecker is a function of type `preterm->term`. If the flag `preterm` is set to `true` (the default is `false`), then HOL will use whatever ML function is currently bound to the name `preterm_handler` as the quotation typechecker. The way this works is that when `preterm` is true the parser produces a preterm rather than a term, and then wraps a call of `preterm_handler` around the quotation. Other uses of preterms are possible, for example as patterns for describing terms.

The definition of the ML type `preterm` is:

```
rectype preterm =
  preterm_var      of string                    Variables
| preterm_const    of string                    Constants
| preterm_comb     of preterm # preterm         Combinations
| preterm_abs      of preterm # preterm         Abstractions
| preterm_typed    of preterm # type            Explicit typing
| preterm_antiquot of term                      Antiquotation
```

The function:

```
preterm_to_term : preterm -> term
```

invokes the standard HOL typechecker on a preterm and returns the resulting typechecked term (or causes the standard error message).

The following is a rather contrived example:

```
#letref p_reg = preterm_var `x`;;                                        1
p_reg = preterm_var `x` : preterm

#let preterm_handler p = p_reg:=p;
                         print_string `Typechecking ... `;
                         print_newline();
                         preterm_to_term p;;
preterm_handler = - : (preterm -> term)
```

```
#set_flag(`preterm`,true);;                                              2
false : bool

#"x+y";;
Typechecking ...
"x + y" : term

#p_reg;;
preterm_comb((preterm_comb((preterm_const `+`), preterm_var `x`)),
             preterm_var `y`)
: preterm
```

Different top-level typechecking can be defined by using a different definition of the function `preterm_handler`. Note that typechecking is purely a 'user interface' feature, so changing the typechecker does not compromise the logical soundness of HOL.

A.9 New conversions

Many new conversions have been added to the system. Only those likely to be of general interest are listed here.

A.9.1 Generalized beta-reduction

A new conversion:

```
PAIRED_BETA_CONV : conv
```

has been added to do generalized beta-conversion of tupled lambda abstractions applied to tuples.

Given the term:

"(\(x_1,...,x_n).t) (t_1,...,t_n)"

`PAIRED_BETA_CONV` proves that:

|- (\(x_1,...,x_n). t[x_1,...,x_n]) (t_1,...,t_n) = t[t_1,...,t_n]

The conversion works for arbitrarily nested tuples. For example:

```
#PAIRED_BETA_CONV "(\((a,b),(c,d)). [a;b;c;d]) ((1,2),(3,4))";;                1
|- (\((a,b),c,d). [a;b;c;d])((1,2),3,4) = [1;2;3;4]
```

A.9.2 Arithmetical conversions

The conversion:

```
   ADD_CONV : conv
```

does addition by formal proof. If n and m are numerals then `ADD_CONV "`n `+` m`"` returns the theorem `|-` n `+` m `=` s, where s is the numeral denoting the sum of n and m. For example:

```
#ADD_CONV "1 + 2";;                                                            1
|- 1 + 2 = 3

#ADD_CONV "0 + 1000";;
|- 0 + 1000 = 1000

#ADD_CONV "101 + 102";;
|- 101 + 102 = 203
```

A new conversion has been added for deciding equality of natural number constants.

```
   num_EQ_CONV : conv
```

If n and m are terms constructed from numeral constants and the successor function `SUC`, then: `num_EQ_CONV "`n`=`m`"` returns:

`|- (`n`=`m`) = T` if n and m represent the same number,
`|- (`n`=`m`) = F` if n and m represent different numbers.

In addition, `num_EQ_CONV "`t `=` t`"` returns: `|- (`t`=`t`) = T`

A.9.3 List processing conversions

Two new conversions for lists are now built-in:

```
   LENGTH_CONV : conv
   list_EQ_CONV: conv
```

`LENGTH_CONV`: computes the length of a list. A call to:

`LENGTH_CONV "LENGTH[`t_1`;`$\ldots$`;`t_n`]"`

generates the theorem:

```
|- LENGTH [t1;...;tn] = n
```

The other conversion, `list_EQ_CONV`, proves or disproves the equality of two lists, given a conversion for deciding the equality of elements. A call to:

```
list_EQ_CONV conv "[u1;...;un] = [v1;...;vm]"
```

returns: `|- ([u1;...;un] = [v1;...;vm]) = F` if:

(i) `(n=m)` or

(ii) *conv* proves `|- (ui = vi) = F` for any $1 \leq i \leq m$.

`|- ([u1;...;un] = [v1;...;vm]) = T` is returned if:

(i) `(n=m)` and u_i is syntactically identical to v_i for $1 \leq i \leq m$, or

(ii) `(n=m)` and *conv* proves `|- (ui=vi)=T` for $1 \leq i \leq n$.

A.9.4 Simplifying `let`-terms

A conversion for reducing `let`-terms is now provided.

```
let_CONV : conv
```

Given a term:

"`let` v_1 `=` t_1 `and` $\cdots$ `and` v_n `=` t_n `in` $t[v_1, \ldots, v_n]$"

`let_CONV` proves that:

`|- let` v_1 `=` t_1 `and` $\cdots$ `and` v_n `=` t_n `in` $t[v_1, \ldots, v_n]$ `=` $t[t_1, \ldots, t_n]$

The v_i's can take any one of the following forms:

(i) Variables: `x` etc.

(ii) Tuples: `(x,y)`, `(a,b,c)`, `((a,b),(c,d))` etc.

(iii) Applications: `f (x,y) z`, `f x` etc.

Variables are just substituted for. With tuples, the substitution is done component-wise, and function applications are effectively rewritten in the body of the `let`-term.

```
#let_CONV "let x = 1 in x+y";;                                          1
|- (let x = 1 in x + y) = 1 + y

#let_CONV "let (x,y) = (1,2) in x+y";;
|- (let (x,y) = 1,2 in x + y) = 1 + 2

#let_CONV "let f x = 1 and f y = 2 in (f 10) + (f 20)";;
|- (let f x = 1 and f y = 2 in (f 10) + (f 20)) = 2 + 2

#let_CONV "let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))";;
|- (let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))) =
   (((0 + 2) + 1) + 2) + 1

#CONV_RULE(DEPTH_CONV ADD_CONV)it;;
|- (let f x = x + 1 and g x = x + 2 in f(g(f(g 0)))) = 6

#let_CONV "let f x y = x+y in f 1";;  % NB: partial application %
|- (let f x y = x + y in f 1) = (\y. 1 + y)
```

A.9.5 Skolemization

Two conversions are provided for a higher-order version of Skolemization (using existentially quantified function variables rather than first-order Skolem constants).

The conversion

```
X_SKOLEM_CONV : term -> conv
```

takes a variable parameter, f say, and proves:

`|- (!`x_1 ... x_n`. ?`y`. `$t[x_1,\ldots,x_n,y]$ `  =  (?`f`. !`x_1 ... x_n`. `$t[x_1,\ldots,x_n,f\ x_1 \ldots\ x_n]$

for any input term `!`x_1 ... x_n`. ?`y`.` $t[x_1,\ldots,x_n,y]$. Note that when $n = 0$, this is equivalent to alpha-conversion:

`|- (?`y`.` $t[y]$`) = (?`f`.` $t[f]$`)`

and that the conversion fails if there is already a free variable f of the appropriate type in the input term. For example:

```
X_SKOLEM_CONV "f:num->*" "!n:num. ?x:*. x = (f n)"
```

will fail. The conversion `SKOLEM_CONV` is like `X_SKOLEM_CONV`, except that it uses a primed variant of the name of the existentially quantified variable as the name of the skolem function it introduces. For example:

```
SKOLEM_CONV "!x. ?y. P x y"
```

proves that:

```
|- ?y. !x. P x (y x)
```

A.9.6 Quantifier movement conversions

A complete and systematically-named set of conversions for moving quantifiers inwards and outwards through the logical connectives ~, /\, \/, and ==> has been added to the system. The naming scheme is based on the following atoms:

```
<quant> := FORALL | EXISTS
<conn>  := NOT | AND | OR | IMP
[dir]   := LEFT | RIGHT          (optional)
```

The conversions for moving quantifiers inwards are called:

```
<quant>_<conn>_CONV
```

where the quantifier <*quant*> is to be moved inwards through <*conn*>.

The conversions for moving quantifiers outwards are called:

```
[dir]_<conn>_<quant>_CONV
```

where <*quant*> is to be moved outwards through <*conn*>, and the optional [*dir*] identifies which operand (left or right) contains the quantifier. The complete set is:

```
NOT_FORALL_CONV    |- ~(!x.P) = ?x.~P
NOT_EXISTS_CONV    |- ~(?x.P) = !x.~P
EXISTS_NOT_CONV    |- (?x.~P) = ~!x.P
FORALL_NOT_CONV    |- (!x.~P) = ~?x.P


FORALL_AND_CONV         |- (!x. P /\ Q) = (!x.P) /\ (!x.Q)
AND_FORALL_CONV         |- (!x.P) /\ (!x.Q) = (!x. P /\ Q)
LEFT_AND_FORALL_CONV    |- (!x.P) /\ Q = (!x'. P[x'/x] /\ Q)
RIGHT_AND_FORALL_CONV   |- P /\ (!x.Q) = (!x'. P /\ Q[x'/x])


EXISTS_OR_CONV          |- (?x. P \/ Q) = (?x.P) \/ (?x.Q)
OR_EXISTS_CONV          |- (?x.P) \/ (?x.Q) = (?x. P \/ Q)
LEFT_OR_EXISTS_CONV     |- (?x.P) \/ Q = (?x'. P[x'/x] \/ Q)
RIGHT_OR_EXISTS_CONV    |- P \/ (?x.Q) = (?x'. P \/ Q[x'/x])


FORALL_OR_CONV
  |- (!x.P \/ Q) = P \/ !x.Q            [x not free in P]
  |- (!x.P \/ Q) = (!x.P) \/ Q          [x not free in Q]
  |- (!x.P \/ Q) = (!x.P) \/ (!x.Q)     [x not free in P or Q]


OR_FORALL_CONV
  |- (!x.P) \/ (!x.Q) = (!x.P \/ Q)     [x not free in P or Q]
```

```
LEFT_OR_FORALL_CONV    |- (!x.P) \/ Q = !x'. P[x'/x] \/ Q
RIGHT_OR_FORALL_CONV   |- P \/ (!x.Q)  = !x'. P \/ Q[x'/x]

EXISTS_AND_CONV
  |- (?x.P /\ Q) = P /\ ?x.Q            [x not free in P]
  |- (?x.P /\ Q) = (?x.P) /\ Q          [x not free in Q]
  |- (?x.P /\ Q) = (?x.P) /\ (?x.Q)     [x not free in P or Q]

AND_EXISTS_CONV
  |- (?x.P) /\ (?x.Q) = (?x.P /\ Q)     [x not free in P or Q]

LEFT_AND_EXISTS_CONV    |- (?x.P) /\ Q = ?x'. P[x'/x] /\ Q
RIGHT_AND_EXISTS_CONV   |- P /\ (?x.Q)  = ?x'. P /\ Q[x'/x]

FORALL_IMP_CONV
  |- (!x.P ==> Q) = P ==> !x.Q          [x not free in P]
  |- (!x.P ==> Q) = (?x.P) ==> Q        [x not free in Q]
  |- (!x.P ==> Q) = (?x.P) ==> (!x.Q)   [x not free in P or Q]

LEFT_IMP_FORALL_CONV   |- (!x.P) ==> Q = !x'. P[x/'x] ==> Q
RIGHT_IMP_FORALL_CONV  |- P ==> (!x.Q) = !x'. P ==> Q[x'/x]

EXISTS_IMP_CONV
  |- (?x.P ==> Q) = P ==> ?x.Q          [x not free in P]
  |- (?x.P ==> Q) = (!x.P) ==> Q        [x not free in Q]
  |- (?x.P ==> Q) = (!x.P) ==> (?x.Q)   [x not free in P or Q]

LEFT_IMP_EXISTS_CONV   |- (?x.P) ==> Q = !x'. P[x/'x] ==> Q
RIGHT_IMP_EXISTS_CONV  |- P ==> (?x.Q) = ?x'. P ==> Q[x'/x]
```

A.10 New built-in theorems

Quite a few new theorems have been added to HOL. Some are listed below, others can be found in the libraries and in `contrib`.

A.10.1 New arithmetical theorems

The built-in theory arithmetic has been augmented with a minimal theory of the functions `MOD` and `DIV` on the natural numbers. The old definitions of `MOD` and `DIV`:

```
MOD         |- MOD k n = @r. ?q. (k = (q * n) + r) /\ r < n
DIV         |- DIV k n = @q. (k = (q * n) + (k MOD n))
```

have been deleted, and replaced by the constant specification:

```
DIVISION    |- !n. 0 < n ==>
                (!k. (k = ((k DIV n) * n) + (k MOD n)) /\ (k MOD n) < n)
```

the validity of which is justified by the division algorithm:

```
DA          |- !k n. 0 < n ==> (?r q. (k = (q * n) + r) /\ r < n)
```

The following theorems are also available:

```
MOD_ONE      |- !k. k MOD (SUC 0) = 0
DIV_LESS_EQ  |- !n. 0 < n ==> (!k. (k DIV n) <= k)
DIV_UNIQUE   |- !n k q. (?r. (k = (q * n) + r) /\ r < n) ==> (k DIV n = q)
MOD_UNIQUE   |- !n k r. (?q. (k = (q * n) + r) /\ r < n) ==> (k MOD n = r)
DIV_MULT     |- !n r. r < n ==> (!q. ((q * n) + r) DIV n = q)
LESS_MOD     |- !n k. k < n ==> (k MOD n = k)
MOD_EQ_0     |- !n. 0 < n ==> (!k. (k * n) MOD n = 0)
ZERO_MOD     |- !n. 0 < n ==> (0 MOD n = 0)
MOD_MULT     |- !n r. r < n ==> (!q. ((q * n) + r) MOD n = r)
MOD_TIMES    |- !n. 0 < n ==> (!q r. ((q * n) + r) MOD n = r MOD n)
MOD_PLUS     |- !n. 0 < n ==>
                    (!j k. ((j MOD n) + (k MOD n)) MOD n = (j + k) MOD n)
MOD_MOD      |- !n. 0 < n ==> (!k. (k MOD n) MOD n = k MOD n)
```

The following miscellaneous arithmetical theorems are now pre-proved.

```
ZERO_LESS_EQ         |- !n. 0 <= n
LESS_EQ_MONO         |- !n m. (SUC n <= SUC m) = (n <= m)
LESS_OR_EQ_ADD       |- !n m. n < m \/ ?p. n = p+m
SUC_NOT              |- !n. ~(0 = SUC n)
SUB_MONO_EQ          |- !n m. (SUC n) - (SUC m) = n - m
SUB_PLUS             |- !a b c. a - (b + c) = (a - b) - c
INV_PRE_LESS         |- !m n. 0 < m /\ 0 < n ==> ((PRE m)<(PRE n) = m<n)
INV_PRE_LESS_EQ      |- !n. 0 < n ==> !m. ((PRE m)<=(PRE n) = m<=n)
SUB_LESS_EQ          |- !n m. (n - m) <= n
SUB_EQ_EQ_0          |- !m n. (m - n = m) = (m = 0) \/ (n = 0)
SUB_LESS_0           |- !n m. m < n = 0 < (n - m)
SUB_LESS_OR          |- !m n. n < m ==> n <= (m - 1)
LESS_SUB_ADD_LESS    |- !n m i. i < (n - m) ==> (i + m) < n
TIMES2               |- !n. 2 * n = n + n
LESS_MULT_MONO       |- !m i n. ((SUC n) * m) < ((SUC n) * i) = m < i
MULT_MONO_EQ         |- !m i n. ((SUC n) * m = (SUC n) * i) = (m = i)
ADD_SUB              |- !a c. (a + c) - c = a
LESS_EQ_ADD_SUB      |- !c b. c <= b ==> (!a. (a + b) - c = a + (b - c))
SUB_EQUAL_0          |- !c. c - c = 0
LESS_EQ_SUB_LESS     |- !a b. b <= a ==> (!c. (a - b) < c = a < (b + c))
NOT_SUC_LESS_EQ      |- !n m. ~(SUC n) <= m = m <= n
SUB_SUB              |- !b c. c <= b ==> (!a. a - (b - c) = (a + c) - b)
LESS_IMP_LESS_ADD    |- !n m. n < m ==> (!p. n < (m + p))
LESS_EQ_IMP_LESS_SUC |- !n m. n <= m ==> n < (SUC m)
```

```
SUB_LESS_EQ_ADD      |- !m p. m <= p ==> (!n. (p - m) <= n = p <= (m + n))
SUB_CANCEL           |- !p n m. n <= p /\ m <= p ==>
                                ((p - n = p - m) = (n = m))
CANCEL_SUB           |- !p n m. p <= n /\ p <= m ==>
                                ((n - p = m - p) = (n = m))
NOT_EXP_0            |- !m n. ~((SUC n) EXP m = 0)
ZERO_LESS_EXP        |- !m n. 0 < ((SUC n) EXP m)
ODD_OR_EVEN          |- !n. ?m. (n = (SUC(SUC 0)) * m) \/
                                (n = ((SUC(SUC 0)) * m) + 1)
LESS_EXP_SUC_MONO    |- !n m.
                         ((SUC(SUC m)) EXP n)<((SUC(SUC m)) EXP (SUC n))
LESS_EQUAL_ANTISYM   |- !n m. n <= m /\ m <= n ==> (n = m)
LESS_ADD_SUC         |- !m n. m < (m + (SUC n))
```

A.10.2 New theorems about lists

```
LIST_NOT_EQ: |- !l1 l2. ~(l1 = l2) ==> (!h1 h2. ~(CONS h1 l1 = CONS h2 l2))
NOT_EQ_LIST: |- !h1 h2. ~(h1 = h2) ==> (!l1 l2. ~(CONS h1 l1 = CONS h2 l2))
EQ_LIST      |- !h1 h2. (h1 = h2) ==>
                        (!l1 l2. (l1 = l2) ==> (CONS h1 l1 = CONS h2 l2))
```

A.10.3 New propositional theorems

```
LEFT_AND_OVER_OR   |- !t1 t2 t3. t1 /\ (t2 \/ t3) = t1 /\ t2 \/ t1 /\ t3
RIGHT_AND_OVER_OR  |- !t1 t2 t3. (t2 \/ t3) /\ t1 = t2 /\ t1 \/ t3 /\ t1
LEFT_OR_OVER_AND   |- !t1 t2 t3. t1 \/ t2 /\ t3 = (t1 \/ t2) /\ (t1 \/ t3)
RIGHT_OR_OVER_AND  |- !t1 t2 t3. t2 /\ t3 \/ t1 = (t2 \/ t1) /\ (t3 \/ t1)


OR_IMP_THM         |- !t1 t2. (t1 = t2 \/ t1) = t2 ==> t1
NOT_IMP            |- !t1 t2. ~(t1 ==> t2) = t1 /\ ~t2
COND_ID            |- !b t. (b => t | t) = t
DISJ_ASSOC         |- !t1 t2 t3. t1 \/ t2 \/ t3 = (t1 \/ t2) \/ t3
```

A.11 Additions and deletions

The ML identifiers that have been added and deleted in Version 2.0 are listed here. The reasons for the changes are given in the files:

```
hol/Version.2.0
hol/Versions/Version.1.12
hol/Versions/Versions.pre.1.12
```

The file `Version.`m`.`n (in the sources) documents HOL Version $m.n$ and may contain details not in *DESCRIPTION* or *REFERENCE*.

A.11.1 Additions

Here is a list of all the new functions in Version 2.0, the ones that are not described in this appendix or earlier in *DESCRIPTION* should be documented in *REFERENCE*.

```
get_const_type          parity                  is_type
mk_let                  dest_let                is_let
bndvar                  body                    is_axiom

mk_cons                 dest_cons               is_cons
mk_list                 dest_list               is_list
list_mk_pair            strip_mk_pair

ancestors               ancestry

prove_rep_fn_one_one    prove_rep_fn_onto
prove_abs_fn_one_one    prove_abs_fn_onto

define_finite_set_syntax  define_set_abstraction_syntax
associate_restriction

X_FUN_EQ_CONV           COND_CONV               PAIRED_BETA_CONV
BOOL_EQ_CONV            ADD_CONV                LENGTH_CONV
list_EQ_CONV            num_EQ_CONV             let_CONV
EXISTS_LEAST_CONV       X_SKOLEM_CONV           SKOLEM_CONV
NOT_FORALL_CONV         NOT_EXISTS_CONV
EXISTS_NOT_CONV         FORALL_NOT_CONV
FORALL_AND_CONV         AND_FORALL_CONV
LEFT_AND_FORALL_CONV    RIGHT_AND_FORALL_CONV
EXISTS_OR_CONV          OR_EXISTS_CONV
LEFT_OR_EXISTS_CONV     RIGHT_OR_EXISTS_CONV
FORALL_OR_CONV          OR_FORALL_CONV
LEFT_OR_FORALL_CONV     RIGHT_OR_FORALL_CONV
EXISTS_AND_CONV         AND_EXISTS_CONV
LEFT_AND_EXISTS_CONV    RIGHT_AND_EXISTS_CONV
FORALL_IMP_CONV         EXISTS_IMP_CONV
LEFT_IMP_FORALL_CONV    RIGHT_IMP_FORALL_CONV
LEFT_IMP_EXISTS_CONV    RIGHT_IMP_EXISTS_CONV

last                    butlast

getenv                  library_pathname        is_ml_infix
set_help_search_path    help_search_path

EQF_INTRO               ISPEC                   ISPECL
EQF_ELIM                EXISTENCE               IMP_CONJ
EXIST_IMP

REPEAT_GTCL             AP_THM_TAC
```

A.11.2 Deletions

The large number of deletions listed below are mostly the result of a "spring cleaning" of the HOL sources. It is hoped that nothing in use has been deleted by mistake—please let us know if you want anything back.

```
abbrev_ty_def            EXPAND_TY_DEFINITION
pp_constants             pp_axioms                pp_theorem
pp_delete_theorem        pp_theorems
new_pp_theory            close_pp_theory
load_pp_theory           extend_pp_theory         new_pp_predicate
mk_quant                 mk_bool_comb space       ascii_ize
dest_quant               dest_bool_comb           truth
mk_iff                    dest_iff                is_iff

TOTALLY_AD_HOC_LEMMA

AND_CLAUSE1              AND_CLAUSE2              AND_CLAUSE3
AND_CLAUSE4              AND_CLAUSE5
OR_CLAUSE1               OR_CLAUSE2               OR_CLAUSE3
OR_CLAUSE4               OR_CLAUSE5
IMP_CLAUSE1              IMP_CLAUSE2              IMP_CLAUSE3
IMP_CLAUSE4              IMP_CLAUSE5
EQ_CLAUSE1               EQ_CLAUSE2               EQ_CLAUSE3
EQ_CLAUSE4
COND_CLAUSE1             COND_CLAUSE2

paired_mem               paired_tryfind           paired_aconv_term
paired_map               paired_filter            paired_subst_term
paired_forall            paired_mapfilter         paired_subst_occs_term
paired_exists            paired_rev_itlist        paired_term_freein_term
paired_find              paired_term_match        paired_type_in_term
paired_type_in_type      paired_set_left          paired_axiom
paired_inst_type         paired_set_right         paired_inst_term
paired_cons              paired_variant           paired_eq
paired_form_match        paired_aconv_form        paired_subst_form
paired_subst_occs_form   paired_term_freein_form  paired_term_freein_form
paired_type_in_form      paired_inst_form

new_curried_infix        curried_infixes          paired_infixes

rename_form              rename_term              print_form
mk_inequiv               dest_inequiv             is_inequiv
is_equiv                 dest_equiv               mk_equiv
gen_all                  new_predicate
make_tuple               list_variant             nil
diagonal                 eqfst                    eqsnd
flip                     mk_tuple                 dest_tuple
```

```
OLD_IMP_RES_TAC          HOL_IMP_RES_THEN          HOL_STRIP_THM_THEN
OLD_RES_TAC              HOL_RES_THEN              HOL_STRIP_ASSUME_TAC
OLD_IMP_RES_THEN         RES_CANON_FUN             HOL_MATCH_MP
OLD_RES_THEN             MULTI_MP                  HOL_RESOLVE_THEN
MP_CHAIN                 REDEPTH_CHAIN             SUB_CHAIN
ONCE_DEPTH_CHAIN         DSPEC                     DSPECL


print_tok_ty             print_tok_thm             apply_type_op
print_all_tok_thm
int_of_tok               (replaced by int_of_string)
tok_of_int               (replaced by string_of_int)
int_to_term              term_to_int
form_class               form_frees
form_vars                form_tyvars
term_frees               (equivalent to frees)
term_vars                (equivalent to vars)
term_tyvars              (equivalent to tyvars)
term_class


seg                      (now in library `eval`)
word_seg                 (now in library `eval`)
word_el                  (now in library `eval`)
truncate                 (now in library `eval`)
upto
keyword


LESS_OR                  (replaced by LESS_SUC_EQ)
GSUBS


DEPTH_EXISTS_CONV        DISJS_CONV                BINOP_CONV
ANTE_FORALL_CONV
BOOL_EQ_CONV             (renamed bool_EQ_CONV)
MOVE_EXISTS_OUT_CONV     (replaced by RIGHT_AND_EXISTS_CONV)


is_string                is_int                    is_cons
obj_of_string            obj_of_int                cons
string_of_obj            int_of_obj
left                     right
set_left                 set_right
eq                       lisp_eval
make_set                 (renamed to setify)
thenf                    orelsef                   all_fun
no_fun                   first_fun                 every_fun
repeatf                  mapshape
```

```
BETA_RED_CONV             LENGTH_MAP_CONV          make_definitions
BINDER_CONV               LEN_SIMP_CONV            mk_def
CHOOSE_RULE               LHS_CONV                 mk_defs
COND_F                    MAP_ELIM                 mk_fn_ty
COND_T                    MOVE_FORALL_IN           mk_fun_test
CONJS_CONV                OR_ELIM_CONV             mk_funct
DEPTH_FORALL_CONV         OR_IMP                   mk_injs
DO_ASM                    PROJ                     mk_new_vars
ELIM_ANTE_EQNS_CONV       RM_LEN_CONV              mk_subset_pred
ELIM_ANTE_EQN_CONV        SIMP_CONV                mk_sum_ty
ELIM_LEN_CONV             SIMP_ISL1                mk_test
ELIM_MAP_CONV             SIMP_ISL2                mk_tl_spec
EQ_ANTE_ELIM              SIMP_RHS                 mk_tuple_ty
EXISTS_RULE               TEST_CONV                pars_ty

E_I_CONV                  clean                    parse_type
FN_BETA                   cleanup                  proj
FN_CASES_SIMP             do_abs                   prove_existence_thm
FN_CASE_SIMP              extract_list             strip_cases
FORALL_AND_CONV           extract_tuple            strip_tok
FORALL_CONJ_CONV          gen_names                strip_val
FOUR_BINDER_CONV          infix_ty                 strip_vty_tok
FST_SND_SIMP              is_char                  sub_conv
HD_TL_SIMP                list_gen_alpha           ty_case
variant_tyvar             rotate_left              rotate_right
derive_existence_thm      instantiate_existence_thm
mk_fn                     closeup

CLOSE_UP                  save_open_thm
HOLdir                    set_hol_lib_dir
load_from_lib             loadt_from_lib           loadf_from_lib
loadx                     theories_dir_pathname

IFF_DEF  I                FF_EQ_THM1               IFF_EQ_THM2
IFF_EQ                    IFF_EQ_RULE              CONJ_IFF
IFF_CONJ                  IFF_THEN2                IFF_THEN
IFF_TAC                   EXISTS_REFL_TAC

define_new_type_isomorphisms    (renamed define_new_type_bijections)

mk_rewrites               mk_rewritesl             REWRITES_CONV
```

Index

- (abbreviation, of HOL theory part names), 168
+ (addition, in HOL logic), 183
+ (addition, in ML), 25, 29, **53**
^ (antiquotation, in HOL logic), 154
:= (assignment, in ML), 8, 25, 36
= (binding operator, in ML), 24
(···) (brackets, in ML), 24, 26
@ (choice function, in HOL logic), 145, 168, 169, 176
%··· % (comment markers, in ML), 27
%<··· >% (comment markers, nested, in ML), 27
/\ (conjunction, in HOL logic), 145, 169, 176
& (conjunction, in ML), 25, 29, 36
+ (disjoint union type operator, in HOL logic), 189
+ (disjoint union type operator, in ML), 21, 40
\/ (disjunction, in HOL logic), 145, 169, 176
/ (division, in ML), 25, 29, **53**
= (equality, in HOL logic), 145, 168
= (equality, in ML), 25, 29, **53**
$ (escape, in HOL logic parser), 145, 162
$ (escape, in ML parser), 55
\ (escape, in ML strings), 29, 209
? (existential quantifier, in HOL logic), 145, 169, 176
?! (exists unique, in HOL logic), 169, 176
? (failure trap, in ML), 15, 26
!! (failure trap, iterative selective, in ML), 17, 26
! (failure trap, iterative, in ML), 17, 26
?? (failure trap, selective, in ML), 16, 26
\ (function abstraction binder, in HOL logic), 146, 209
\ (function abstraction binder, in ML), 14, 26, 38
o (function composition operator, in HOL logic), 186–187
o (function composition operator, in ML), **56**, 273
-> (function type operator, in HOL logic), 144, 176
-> (function type operator, in ML), 40
>= (greater or equal, in HOL logic), 184
> (greater than, in HOL logic), 184
> (greater than, in ML), 25, 29, **53**
==> (implication, in HOL logic), 145, 168, 176
<= (less or equal, in HOL logic), 184
< (less than, in HOL logic), 183
< (less than, in ML), 25, 29, **53**
@ (list concatenation, in ML), 25, 29–30, **53**, 54, 58
. (list constructor, in ML), 24, 25, 29, **53**
[··· ; ···] (lists, in HOL logic), 184–186
[··· ; ···] (lists, in ML), 12, 26
- (minus, in ML), 25, 29
* (multiplication, in HOL logic), 184
* (multiplication, in ML), 25, 29, **53**
~ (negation, in HOL logic), 145, 169, 176

- ('non-printable' symbol, in ML), 9, 20, 50
, (pair constructor, in HOL logic), 177
, (pair constructor, in ML), 24, 25, 29, 36
(product function operator, in ML), **56**
(product type operator, in HOL logic), 177
(product type operator, in ML), 9, 40
(prompt, in ML), 7, 85
; (sequencing, in ML), 11, 26
^ (string concatenation, in ML), **66**
‘ ... ‘ (string markers, in ML), 13, 29
- (subtraction, in HOL logic), 184
- (subtraction, in ML), 25, 29, **53**
"..." (term quotes, in ML), 142–144
;; (terminator, of ML expressions), 7
|- (theorem marker, in HOL logic), 157
: (type constraint, in HOL logic), 140, 142
: (type constraint, in ML), 24, 25, 32
":..." (type quotes, in ML), 142–144
*, **, ... (type variables, in HOL logic), 144
*, **,... (type variables, in ML), 14, 40
! (universal quantifier, in HOL logic), 145, 169, 176
<< (value comparison, in ML), **81**
(), *see* void
_ (wildcard, in ML), 22, 24, 32, 34, 43
0 (zero, in HOL logic), 179

abbreviation of types, in HOL logic, 174
abbreviation of types, in ML, 17, 24, 40, 47
abort_when_fail, **84**
ABS, 116, 117, **172**
ABS_..., 193
abs_..., 20, 49
ABS_CONV, **245**
ABS_REP_THM, 179
absrectype...with..., 24, 49–50
abstract types, in ML, 19–20, 40, 49–50
  listing current, 82
  printing of, 86–87
  reasons for, 48
abstract-typed values, in ML
  printing of, 20, 50
abstraction rule, in HOL logic
  abstract form of, 116
  ML function for, 172
abstype...with..., 19–20, 49–50
ACCEPT_TAC, 256, **265**
achievement, of goals, 256
ADD, 183, 184
ADD_ASSUM, **223**
ADD_CONV, **247**, 302
ALL_CONV, **244**
ALL_TAC, 258, **270**
allowed_constant, **206**
alphanumerics, in ML, 27–28
alternation
  of conversions, 239–240
  of tactics, 271
ancestry, of HOL system theories, 158, 175
and, 24
AND_DEF, 169
antiquotation, in HOL logic terms, 154, 174
AP_TERM, 223, **225**
AP_THM, 223, **225**
append, **58**
APPEND, the constant in HOL logic, 186
APPEND, the theorem in HOL logic, 186
append_openw, **69**
application, *see* function application
ARB, 147, 170, 177
arithmetic, 184
arithmetic, in HOL logic, 184
arithmetic, in ML, 82
array, of assumptions, 280

ASCII, 26, 91
`ascii`, **52**
`ascii_code`, **52**
`ASM_CASES_TAC`, 268
`ASM_REWRITE_TAC`, 253, 269
assignable variables, in ML, 8, 11, 31, 33, 36, 47
assignments, in ML
  evaluation of, 36
  simultaneous, 9
  syntax of, 25
  type checking of, 44
`assoc`, **62**
`associate_restriction`, **147**, 293
associativity, of constructs in ML, 10, 23, 26
  of infixes, 29
`ASSUM_LIST`, 280, 285
`ASSUME`, **171**, 218
`ASSUME_TAC`, **266**
assumption introduction, in HOL logic
  abstract form of, 116
  ML function for, 171
assumption list, of goal, 256
assumptions
  as array, 280
  as stack, 276
  compared methods of handling, 285
  denoting of, in proofs, 275, 278, 281
  discarding of, in proofs, 275, 277, 279
  explicit, 278
  importance of ordering of, 283
  internal, 284
  of sequents, 114
  of theorems, *see* hypothesis
  role of, in goal directed proof, 275
  tactic for adding, 266
atomic types, in HOL logic, 101
`autoload`, **76**
`autoload_theory`, **79**, **211**
autoloading, *see* automatic loading
automated derivation
  of case analysis theorems, 201
  of characterizing theorems for recursive types, 193
  of distinctness theorems, 201
  of one-to-one theorems, 201
  of recursive definitions, 196
  of structural induction tactics, 200
  of structural induction theorems, 199
automatic loading, of HOL theory components, 211
`auxiliary` library, 91
`axiom`, **168**
axiom of choice, 100
axiom of infinity, 169, 170
  abstract form of, 121
axiom of replacement, 100
axioms
  abstract form of HOL logic's, 121
  as inference rules, 117
  autoloading of, in HOL system, 211
  declaration of, in HOL logic, 160
  dispensibility of adding, 123, 170
  formal semantics of HOL logic's, 120–121
  in `bool` theory, 168, 178
  in `ind` theory, 170
  in `list` theory, 185
  in natural deduction, 156
  in `num` theory, 179
  non-primitive, of HOL logic, 168, 170, 178, 179, 185
  primitive, of HOL logic, 168–170
  retrieval of, in HOL system, 167–168
`axioms`, **167**
axioms, in a HOL theory, 118

`B`, **56**
`b`, the subgoal package function, 265

backtrace, 67
`backup`, **264**
`backup_limit`, 264
`bags` library, 91
`BASIC-HOL`, 175, 179
`basic-rewrites`, 252
beta-conversion, in HOL logic
  abstract form of, 116
  ML function for, 171
  not expressible as a theorem, 222, 238
`BETA_CONV`, 116, 117, **171**, 237
`BETA_RULE`, **242**
bijection of types, in HOL logic, 165
binary tree theory, in HOL logic, 195
`BINDERS`, 177
`binders`, **167**
binders, in HOL logic, 146, 163
  declaration of, 160
  indicator functions for, 206
  parsing of, 150
  under interface maps, 205
binding power, in ML, 26
bindings, in ML
  abstract type, 19–20, 49
  concrete type, 17–19, 48
  effect of declarations on, 31
  evaluation of, 33–34
  in declarations, 7–8
  in function definitions, 11, 33–34
  kinds of, 33–34
  multiple, 34
  predeclared, 51–52
  scope of, 31
  simple, 33
  simultaneous, 8
  syntax of, 24
`bndvar`, 142
BNF syntax, explanation of, 23
`body`, 142
`bool`, the HOL theory, 168, 169, 175–178
`bool`, the type in HOL logic, 140, 142, 168
`bool`, the type in ML, 40, 41
`BOOL_CASES_AX`, 169
  abstract form of, 121
boolean values, *see* truth values
bound variables, in HOL logic, 106
breaks, in ML pretty printing, 73
`butlast`, **59**

`C`, **56**
`cached_theories`, **212**
caching, 211
`can`, **57**
canonical contexts, in HOL logic
  of terms, 107
  of types, 103
cardinals library, 91
carriage return
  after ML expressions, 7
  in ML expressions, 26
  in ML strings, 29
Carroll, J., 3
Cartesian products, *see* product types
case analysis, in HOL logic
  as instance of induction, 199, 202
  tactics for, 268
  theorems for, 201
`case` ··· `of` ··· `|` ···, 26
  relation to abstraction, 19, 26
`CB`, **56**
`CCONTR`, 223, **236**
Centaur, 83, 93
`CHANGED_CONV`, **244**
`CHANGED_TAC`, **272**
characteristic function, of type definitions, 129
characteristic predicate, of type definitions, 164, 169
characterizing theorem
  for defined types, 193

for finitely-branching ordered trees, 188
for lists, 185
for numbers, 180
use of, in deriving induction, 200
choice axiom, 169
abstract form of, 121
choice operator, in HOL logic, 168, 176
abstract form of, 112
formal semantics of, 112
inference rules for, 230, 231
primitive axiom for, 169
syntax of, 145
CHOOSE, 223, **232**
CHOOSE_THEN, **286**
chop_list, **63**
Church, A., 99, 168, 169
close, **70**
close_theory, **161**, 167
Co, **56**
combin, 186
combinations, in HOL logic
abbreviation for multiple, 146
abstract form of, 105
constructor for, 141
destructor for, 141
formal semantics of, 108
quotation of, 144
combinations, in ML, 36, 44
combinators, in HOL logic, 186
combinators, in ML, 55–56
combine, **64**
COMMA_DEF, 178
comments, in ML, 27
Common Lisp HOL, 3, 72, 76, 84
compilation, in ML, 84
compile, **72**
compile_on_the_fly, 72, **84**
compilef, **72**
compilet, **72**
compound monotypes, in ML, 41
compound tactics, in HOL system, 271
compound types, in HOL logic
abstract form of, 101
constructors for, 140, 144
destructors for, 140
computation phase, of proof search, 262
concat, **66**
concatenation, of lists
in HOL logic, 186
in ML, 25, 29, 30, 53, 54, 58
concatenation, of ML strings, 66
concatl, **66**
concl, **156**
conclusions
of inference rules, 115, 171
of sequents, 114, 156
of theorems, 156
concrete recursive types, in HOL logic, 191
concrete types, in ML, 17, 40, 48
COND, 145, 146, 149, 170, 177
COND_DEF, 170
conditional/loop construct, in ML, 11–12, 25, 44
evaluation of, 36–37
syntax of, 25
type checking of, 44
conditional predicate, in HOL logic, 145, 149
definitional axiom for, 170
conditionals, in HOL logic, 145, 149, 170
CONJ, 223, **233**
CONJ_TAC, 258, **267**
ML implementation of, 260
CONJUNCT1, 223, **234**
CONJUNCT2, 223, **234**
conjunction, in HOL logic, 176
abstract form of, 119, 124
constructor for, 145
defined in terms of primitives, 169

definitional axiom for, 169
formal semantics of, 120
inference rule for, 233
syntax of, 145
tactic for splitting of, 267
conjunction, in ML, 25, 36
CONS, 184
CONS_11, 185
consistency, of HOL logic, 157, 257–258
under constant specification, 125, 126
under type definition, 128
consistent theory, 122
constant definition, 123
constant definition extension, of HOL logic
abstract form of, 123
ML function for, 161–162
constant expressions, in ML, 29, 33, 35, 43
constant specification, 125
constant specification extension, of HOL logic
abstract form of, 125
ML function for, 163
constant specifications, in HOL logic
autoloading of, 211
constants, **167**
constants, in HOL logic
abbreviational, 177
abstract form of, 105
constructor for, 141
declaration of, 160
destructor for, 141
for pairs, 177
for type definitions, 177
formal semantics of, 108
hiding status of, 210
indicator function for, 206
logical, 176
primitive logical, 168
primitive, abstract form of, 111
supressing parsing behaviour of, 145
system, 177
constructions, in ML, 24, 34, 36, 48
constructors, in ML, 25, 29
type checking of, 43
constructors, of concrete types in HOL logic
proving distinct, 201
proving one-to-one, 201
contexts, in semantics of HOL logic, 106
contradiction rule, in HOL logic, 236
CONV_RULE, **242**
CONV_TAC, **242**
conversion and unwinding library, 91
conversions
application specific, 237
as families of equations, 238
functions for building, 243–246
implementation of, in ML, 245–246
operators on, 239–243
purpose of, 237
quantifier movement, 250, 305
reduction by, 239
convert library, 91
core images, saving of, 76
counting inferences, in HOL proofs, 203, 213, 218, 219, 221
current_theory, **159**
CURRY, 177
curry, **57**
currying, in ML, 38, 52
in primitive recursive definitions, 182
in recursive definitions, 196
customization of HOL system, 83, 85, 209–210

debugging, of tactics, 261, 263
declarations, in ML
at top level, 31
effect on it, 8

evaluation of, 38
local, 8, 26, 38
relation to function abstraction, 14
syntax of, 24
type checking of, 45
declared constants, in HOL logic, 145, 162
decomposition phase, of proof search, 262
deductive systems, 115, 156
`..._DEF`, 182
default print depth, for HOL logic, 212
default types, in HOL logic, 204–205
`define_finite_set_syntax`, **152**, 295
`define_new_type_bijections`, **165**
`define_set_abstraction_syntax`, **152**, 295
`define_type`, **191**, 192, 196, 200
defined types in ML
flag for printing, 84
printing of, 82
defining mechanisms, for HOL logic, 161, 162
`definition`, **168**
definitional axioms, 164, 190
definitional extension, of HOL logic, 161
definitional theories, 170
`definitions`, **167**
definitions, adding to HOL logic, 162
`delete_cache`, **211**
denoting assumptions, 275, 278, 281
dependent types in HOL logic, 146, 293
`DEPTH_CONV`, **241**, 246
search strategy of, 241
use of, in rewriting tools, 252
derived rules, in HOL logic
importance of, 218
justification of, 219
list and derivations of some, 223–236
list of axiomatic, 223
pre-defined, 221–222
`dest_abs`, **141**
`dest_comb`, **141**
`dest_const`, **141**
`dest_thm`, **156**
`dest_type`, **140**
`dest_var`, **141**
`dest_vartype`, **140**
Dhingra, I., 3
discarding assumptions, 275, 277, 279
`DISCH`, 117, **173**
`DISCH_TAC`, **267**, 276
discharging assumptions, in HOL logic
abstract form of, 117
ML function for, 173
`DISJ1`, 223, **234**
`DISJ2`, 223, **235**
`DISJ_CASES`, 223, **235**
`DISJ_CASES_TAC`, **268**
disjoint union theory, in HOL logic, 189
disjoint union, in ML, 21, 40
disjunction, in HOL logic, 176
abstract form of, 119, 124
constructor for, 145
defined in terms of primitives, 169
definitional axiom for, 169
formal semantics of, 120
inference rule for, 234–236
syntax of, 146
tactic for case splits on, 268
disjunction, in ML, 25, 36
`distinct`, **65**
`DIV`, 184
`do`, 25, 29, **53**
dollared infixes, in ML, 53, 55
draft mode, in HOL system, 159, 161, 167, 193, 206
`draft_mode`, **206**

`e`, the subgoal package function, 265
`el`, **58**
`EL`, the constant in HOL logic, 186
`EL`, the theorem in HOL logic, 186

emacs, 85
empty, *see* `void`
environments, in ML evaluations, 31, 35–38
epsilon operator, 112, 169
`EQ_IMP_RULE`, 223, **226**
`EQ_MP`, 223, **225**
`EQ_TAC`, **270**
`EQT_ELIM`, **226**
`EQT_INTRO`, 223, **227**
equality, in HOL logic, 168, 169, 176
  abstract form of, 112
  formal semantics of, 111
  `MP` rule for, 225
  other rules for, 226–228
  primitive axiom for, 169
  symmetry rule for, 224
  syntax of, 145
  tactic for splitting, 270
  transitivity rule for, 224
equality, in ML
  at function types, 53
  syntax of, 25
  weakness of, 53
equational theorems, in HOL logic
  produced by conversions, 237, 238
  use of in rewriting, 220
equiv., 23
`ETA_AX`, 169
  abstract form of, 121
`ETA_CONV`, 223, **229**
`eval` library, 91
evaluation, of ML constructs, 31
  control flow in, 32, 36–37
  counting inference invoked by, 84
  termination of, 35
  timing of, 84
`EVERY`, the HOL constant, 186
`EVERY`, the ML function, **274**
`EVERY_ASSUM`, 283
`EVERY_CONV`, **244**
`EVERY_DEF`, 186
existential quantifier, in HOL logic, 176
  abbreviation for multiple, 119, 146
  abstract form of, 112, 119, 124
  defined in terms of primitives, 169
  definitional axiom for, 169
  formal semantics of, 120
  in infinity axiom, 170
  inference rules for, 231–232
  syntax of, 145
  tactic for, 271
`EXISTS`, 223, **231**
`exists`, **61**
exists unique, in HOL logic, 169
  defined in terms of primitives, 169
  definitional axiom for, 169
`EXISTS_DEF`, 169
`EXISTS_TAC`, **271**
`EXISTS_UNIQUE_DEF`, 169
exiting of HOL system, 161
`EXP`, 184
`expand`, **263**
`explode`, **52**
expressions, in ML
  carriage return after, 7
  evaluation of, 35–38
  syntax of, 25
  terminator for, 7
  type checking of, 44
`EXT`, 223, **230**
`extend_theory`, **167**
extension, of HOL logic, 190
  abstract form of, 122
  by constant definition, 123–124, 161
  by constant specification, 125–127, 163
  by type definition, 128–130, 164–165, 190–202
  by type specification, 130–134
  definitional, 161

extension, of theory, 122
extensionality rule, in HOL logic, 230

F, 176
    abstract form of, 119, 124
    axiom for, 157, 169
    defined in terms of primitives, 169
    definitional axiom for, 169
    formal semantics of, 120
    rules of inference for, 236
F_DEF, 169
fail, 25
failure strings, in ML, 15–17
    for division by zero, 53
    of general purpose functions, 55
    of primitive identifiers, 51
failure, in ML, 15–17, 31–32, 35–37
    backtrace functions for, 67
    catch-all trap for, 16
    constructs for, 36–37
    explicit, 31
    implicit, 31
    iterative selective trapping of, 17, 26, 37, 45
    iterative trapping of, 17, 26, 37, 45
    selective trapping of, 16, 26, 37, 45
    side effects of, 35
    similar, 31
    test for, 57
    to cause exit from HOL, 84
    trapping of, 15, 26, 37, 45
failure, of tactics, 257–258, 261–263, 286
    debugging, 261
failwith, 15, 25, 31, 36, 44
false, 29, 43
falsity, in HOL logic, *see* F
families of inferences, in HOL logic, 171, 222, 238
fast_arith, **82**
FCOND, 177
file_load_msg, **84**
filter, **62**
FILTER_ASM_REWRITE_TAC, 269
find, **61**
find_file, **70**
find_ml_file, **71**
find_theory, **71**
finite_sets library, 91
FIRST, **272**
FIRST_ASSUM, 283
FIRST_CONV, **243**
fixed point theory library, 91
fixpoints library, 91
flags, for subgoal package, 265
flags, in ML, 84, 203
flat, **63**, 273
FLAT, the constant in HOL logic, 186
FLAT, the theorem in HOL logic, 186
follows from, in natural deduction, 115, 156
forall, **61**
FORALL_DEF, 169
form feed
    in ML expressions, 26
    in ML strings, 29
formulas as terms, in HOL logic, 112, 156
forward proof
    compared to goal directed, 255
    interfacing to goal directed, 265
Franz Lisp HOL, 3, 72, 76, 84
free variables, in HOL logic, 227, 229–232
    abstract form of, 106
free variables, in ML
    binding of, 38
    in function definitions, 11
fst, **51**
FST, the axiom in HOL logic, 178
FST, the constant in HOL logic, 177
    definition of, 178
FST_DEF, 178

`fun`···`|`···, 18
evaluation of, 38
syntax of, 26
type checking of, 45
function (pattern) selection abstraction, in ML, 18, 26, 38, 45
function abstraction, in HOL logic, 142
abbreviation for multiple, 150
abstract form of, 105
changing symbol for, 209–210
constructor for, 141
destructor for, 141
formal semantics of, 109
inference rules for, 172
paired, 150–151, 203
pretty printing, of paired, 203
relation to `let`-terms, 151
subterms of, 150
symbol for, 144
syntax of, 144
type checking of, 144
uncurrying, in paired, 150–151
function abstraction, in ML
evaluation of, 38
relation to declarations, 14
syntax of, 26
type checking of, 45
function application, in HOL logic
abstract form of, 105
associativity of, 106
constructor for, 141, 144
destructor for, 141
formal semantics of, 108
inference rules for, 225
syntax of, 144
function application, in ML, 9
associativity of, 10
evaluation of, 36
iteration of, 57
partial, 10
precedence of, 9
syntax of, 25
type checking of, 44
function composition, in HOL logic, 186–187
function composition, in ML, 56, 273
function types, in HOL logic
abstract form of, 101
constructors for, 140
destructors for, 140
formal semantics of, 103
function types, in ML, 40, 44
functions in ML, printing of, 9
`funpow`, **57**

`g`, the subgoal package function, 265
`GEN`, 223, **228**
`GEN_TAC`, **266**
generalization rule, in HOL logic, 228
generic types, in HOL logic, 160
generic types, in ML, 42
`get_flag_value`, **85**
`get_state`, **264**
`goal`, 257
goal directed proof search
composition phase of, 262
concepts of, 256–259
decomposition phase of, 262
generation of proofs by, 260
reason for, 255
goals, in HOL system, 256
Gordon, M., 3
`group` library, 91
group theory library, 91

has type *vs* possesses type, 41–42
`hd`, 12, 13, **51**
`HD`, the constant in HOL logic, 184
`HD`, the definition in HOL logic, 186
`help`, **87**, 292
help system, 87–88, 292

help_search_path, **88**, 292
hide_constant, **210**
Hilbert, D., 169
Hoare logic library, 91
HOL, 158, 168, 175
HOL system
  adjustment of user interface of, 202–210
  caching theories in, 211
  changeable symbols of, 209–210
  customization of, 85–87
  flags in, 84, 203
  help search paths for, 88
  hiding constants in, 210
  interface maps in, 205–209
  new interface for, 93–95
  non-Unix versions of, 80
  root directory of, 89
  search paths for, 89
  sticky types in, 204–205
  timing and counting inferences in, 213
  typical work in, 158
  version number of, 81
hol-init.ml, 80
HOL_DEFINITION, 177
host_name, **81**
Huet, G., 3, 176, 220
hyp, **156**
hypotheses
  of inference rules, 115
  of sequents, 115, 156
  of theorems, 156

I, constant in HOL logic, 186
I, the constant in ML, **55**
I_DEF, 186
I_o_ID, 187
I_THM, 187
identifiers, in HOL logic, 137–138
identifiers, in ML
  binding of, 51–54
  dollared, 53–54
  evaluation of, 35
  in expression evaluation, 34
  lexical aspects of, 27–28
  predeclared, 51–54
  type checking of, 43
identity tactic, 258, 270
if ··· loop ··· else ···, 12
  evaluation of, 36–37
  syntax of, 25
  type checking of, 44
if ··· loop ··· loop ···, 25, 36–37
if ··· then ··· else ···, 11–12, 25, 36
if ··· then ··· loop ···, 11–12, 25, 36
iff, in HOL logic, 176
  definitional axiom for, 169
IMP_ANTISYM_AX, 169
  abstract form of, 121
IMP_RES_TAC, **269**
implication, in HOL logic, 168, 176
  abstract form of, 112
  formal semantics of, 111
  inference rules for, 224, 226
  primitive axiom for, 169
  syntax of, 145
  tactics for, 267, 269
implode, **52**
in, 26
ind, the theory, 170, 175
ind, the type, 170
individuals, 170
INDUCT, 179
INDUCT_TAC, 179, 200, 258
INDUCT_THEN, **200**
INDUCTION, 179
induction rule
  for concrete recursive types, 199
  for lists, in HOL logic, 185
  for numbers, in HOL logic, 179

for `trees`, in HOL logic, 188
structural, derivation of, 198–199
induction tactics, 258
derivation of, 200–201
inference rules, of HOL logic
abstract form of primitive, 115
derived, 223–236
formal semantics of, 117
primitive, 171–173
some not properly derived, 222–223
inference schemes, in HOL logic, 171
inference, in natural deduction, 156
inferences, in HOL logic
as ML function applications, 218
counting of, 203, 213
in derived rules, 218
in goal-directed proof search, 256, 262
notation for, 171
`INFINITY_AX`, 170
abstract form of, 121
`infixes`, **167**
infixes, in HOL logic, 145, 162
declaration of, 162
in recursive type definitions, 196
indicator functions for, 206
pretty printing of, 203
primitive recursive definitions, 183
under interface maps, 205
infixes, in ML, 29, 36, 53, 55
currying of, 52
`INIT`
abstract form of, 121
as `BASIC_HOL`, 179
formal semantics of, 121
initial theory, of HOL logic, 179
abstract form of, 121
initiality, 193
initialization of HOL, 80
`inject_input`, **79**
`inl`, 21, **51**
`INL`, the constant in HOL logic, 189
`INL`, the theorem in HOL logic, 190
`inr`, 21, **51**
`INR`, the constant in HOL logic, 189
`INR`, the theorem in HOL logic, 190
`INST_TYPE`, 117, **173**, 220
`install`, **89**
`int`, 7, 40
`int_of_string`, **52**
`integer` library, 91
integers (as quotient types) library, 91
integers (from groups) library, 91
integers, in ML, 29, 43
interface maps, 203, 205–209
in printing, 84
interface, new, 93–95
`interface_map`, **206**
`interface_print`, **84**, 203, 206
`intersect`, **64**
`INV_SUC`, 179
`is_alphanum`, **28**, **139**
`is_binder`, **206**
`is_constant`, **206**
`is_hidden`, **210**
`is_infix`, **206**
`is_letter`, **28**, **138**
`IS_PAIR`, 177
`IS_PAIR_DEF`, 178
`isl`, 21, **51**
`ISL`, the constant in HOL logic, 189
`ISL`, the theorem in HOL logic, 189
`ISL_OR_ISR`, 189
isomorphism of types, in HOL logic, 179, 190
`isr`, 21
`ISR`, the constant in HOL logic, 189
`ISR`, the theorem in HOL logic, 189
`it`, 7, 8
iterated function application, in ML, 57
`itlist`, **60**

justifications, in goal-directed proof search, 256
THEN example of, 274

K, the constant in HOL logic, 186
K, the constant in ML, **55**
K_DEF, 186
K_THM, 187
KI, **56**

labelled tree theory, in HOL logic, 187–188
lambda expressions, in ML, 14–15
extended form of, 18
lambda symbol, in HOL logic, resetting, 209
lambda terms, in HOL logic, 105
last, **59**
LCF, 117, 142, 144, 168, 176, 177, 286
Cambridge, 3, 220, 238, 262
Edinburgh, 3, 220, 255, 276
Leisenring, A., 169
lemmas, 255, 269
length, **58**, 273
LENGTH, the constant in HOL logic, 186
LENGTH, the theorem in HOL logic, 186
LENGTH_CONV, **248**, 302
LESS, 183
less than, in HOL logic, 183
LET, 151, 170, 177
let ... in ..., 7–8
evaluation of, 33, 38
syntax of, 24, 26
type checking of, 43, 45
let, in ML, 7
let-terms, in HOL logic
as abbreviations, 151
constant for, 169, 177
definitional axiom for, 169
pretty printing of, 203
simplification of, 248, 303
let_after, **77**
let_before, **77**
let_CONV, **248**, 303
LET_DEF, 170
letrec ... in ..., 11
evaluation of, 33, 38
syntax of, 24, 26
type checking of, 43, 45
letref ... in ..., 8
evaluation of, 33, 38
syntax of, 24, 26
type checking of, 43, 45
letters, in ML, 27
lettype, 17, 24, 40, 47–48
lhs, 142
libraries
directory of, 92
installation of, 93
load files for, 90, 92
loading of, 84
libraries, **91**
library_pathname, **92**, 291
line feed
in ML expressions, 26
in ML strings, 29
link, **81**
Lisp
accessing of, from HOL system, 82
core images, 76
lisp, **82**
list concatenation, in ML, 12
parser status of, 54
list constructor, in ML, 12, 29, 53
in expression evaluation, 35
syntax of, 24, 25
type checking of, 44
list definitions, in HOL logic, 186
list theorems, in HOL logic, 185–186
list theory, in HOL logic, 184–186
list, the type operator in HOL logic, 184

list, the type operator in ML, 40
list_Axiom, 185
list_CASES, 185
list_EQ_CONV, **248**, 302
LIST_INDUCT, 185
list_INDUCT, 185
LIST_INDUCT_TAC, 185
list_mk_abs, **146**
list_mk_comb, **146**
list_mk_conj, 146
list_mk_disj, 146
list_mk_exists, **146**
list_mk_forall, **146**
list_mk_imp, 146
list_mk_pair, 146
lists, in HOL logic, 184–186
lists, in ML
    evaluation of, 38
    in expression evaluation, 35
    syntax of, 24, 26
    type checking of, 45
load, **71**, 91
load_library, **90**
load_theory, **167**
loadf, **71**
loading, of HOL theories, 211
loadt, **71**
locations, 31, 36
LOG, 119
    formal semantics of, 120
logical constants, in HOL logic, 169, 176
...ltree, 188
ltree, the HOL theory, 187–188
ltree_Axiom, 188
ltree_Induct, 188

map, 14, **59**, 272
MAP, the constant in HOL logic, 186
MAP, the theorem in HOL logic, 186
MAP_EVERY, 200
mapfilter, **62**
mapshape, 273
matching, of patterns, in ML, 34–35
max_print_depth, **75**, **212**
Melham, T., 3, 187
mem, **61**
Milner, R., 3, 137, 143, 220, 255, 272
MIN, 118
minimal theory, of HOL logic, 118
MK_ABS, 223, **244**
mk_abs, **141**, 144
MK_COMB, 223, **244**
mk_comb, **141**, 144
mk_cond, 145
mk_conj, 145
mk_cons, 146
mk_const, **141**, 144
mk_disj, 145
mk_eq, 145
mk_exists, 145
mk_forall, 145
mk_imp, 145
mk_let, 145
mk_list, 146
mk_neg, 145
MK_PAIR, 177
mk_pair, 146
MK_PAIR_DEF, 178
mk_select, 145
mk_type, **140**, 144
mk_var, **141**, 144
mk_vartype, **140**, 144
ML
    arithmetic in, 82
    CAML version of, 3
    constant expressions in, 29
    HOL version of, 3
    identifiers in, 27–30
    original version of, 3
    pretty printer for, 72–75, 212

prompt character of, 7, 84, 85
purpose of, in HOL system, 276
short form of syntax of, 32
special symbols of, 27–28
Standard, 3
syntax of bindings in, 24
syntax of declarations in, 24
syntax of expressions in, 25–26
syntax of patterns in, 24
top level expression terminator in, 7
top level of, 7, 31, 42, 50, 84, 85
type checking of, 32, 42–45
versions of, 3
ml_curried_infix, **52**
ML_eval, **79**
ml_paired_infix, **52**
MOD, 184
Modus Ponens, in HOL logic
abstract form of, 117
ML function for, 173
monotonicity, in deductive systems, 115
Morris, L., 3, 47
MP, 117, **173**
MULT, 184
multiple bindings, in ML, 34

natural deduction, 156
natural numbers, *see* number theory
negation, in HOL logic
abstract form of, 119, 124
constructor for, 145
definitional axiom for, 169
formal semantics of, 120
syntax of, 145
new_..., 167
new_alphanum, **28**, **139**
new_axiom, **160**
new_binder, **160**
new_binder_definition, **162**
new_constant, **160**
new_definition, **162**
new_flag, **85**
new_infix, **160**
new_infix_definition, **162**
new_infix_list_rec_definition, 185
new_infix_prim_rec_definition, **181**, 183
new_letter, **28**, **138**
new_list_rec_definition, **185**
new_parent, **159**
new_prim_rec_definition, **181**, 195
new_recursive_definition, **196**
new_special_symbol, **28**, **139**
new_specification, **163**
new_syntax_block, **89**, 299
new_theory, **159**
new_type, **160**
new_type_abbrev, **174**
new_type_definition, **164**, 190
NIL, 184
NO_CONV, **243**
NO_TAC, **270**
Node, 188
node, 187
Node_11, 188
node_11, 188
non-printable values in ML, 9, 20, 50
not, 25, 29, **53**
NOT_CONS_NIL, 185
NOT_DEF, 169
NOT_NIL_CONS, 185
NOT_SUC, 179
notation
for specification of rules, 171
for specification of tactics, 258
null, 12, **51**
NULL, the constant in HOL logic, 184
NULL, the definition in HOL logic, 186
num, the theory in HOL logic, 179
num, the type in HOL logic, 179
num_Axiom, 180, 195

num_CONV, **179**, 223
num_EQ_CONV, **247**, 302
number theory, in HOL logic, 184
    type definition package version of, 194

ONCE_DEPTH_CONV, **242**
one, the HOL theorem, 190
one, the HOL theory, 190
one, the HOL type, 190
one-to-one predicate, in HOL logic
    abstract form of, 119, 124
    definitional axiom for, 170
    formal semantics of, 121
    in bool theory, 176
one_Axiom, 190
one_axiom, 190
ONE_ONE_DEF, 170
onto predicate, in HOL logic
    abstract form of, 119, 124
    definitional axiom for, 170
    formal semantics of, 121
    in bool theory, 176
ONTO_DEF, 170
openi, **69**
openw, **69**
operating systems functions, from HOL, 80–81
or, 25, 36
    not a dollared infix, 29
OR_DEF, 169
ordinary variables, in ML, 33, 35
ORELSE, **271**
ORELSEC, **239**, 243
outl, 21, **51**
OUTL, the constant in HOL logic, 189
OUTL, the theorem in HOL logic, 190
outr, 21, **51**
OUTR, the constant in HOL logic, 189
OUTR, the theorem in HOL logic, 190

p, the subgoal package function, 265
PAIR, 178
PAIR_EQ, 178
PAIR_EXISTS, 178
PAIRED_BETA_CONV, **247**, 301
pairing constructor, in HOL logic, 177
    associativity of, 177
    definition of, 178
pairs, in HOL logic, 177–178
    in abstractions, 150–151
pairs, in ML, 9
    evaluation of, 36
    in expression evaluation, 35
    syntax of, 24, 25
    type checking of, 44
parents, **167**
parents, of HOL theories, 158
parser library, 91
parsing, of HOL logic
    hiding constant status in, 210
    of binders, 150
    of conditionals, 149
    of constants, before definition, 162
    of function abstractions, 150
    of let-terms, 151
    of list expressions, 185
    of pairs, 177
    of quotation syntax, 142, 150
    of standard notations, 145
    of sum types, 189
partial function application, in ML, 10
partition, **63**
pattern matching, in ML evaluations, 34–35
pattern selection abstraction, *see* function selection abstraction
patterns, in ML, 19
    matching with values, 34–35
    syntax of, 24
    type checking of, 43–44
Paulson, L., 3, 176, 220, 262

Peano's axioms, 179, 180, 188
polymorphic terms, in HOL logic, 106
polymorphic types, in HOL logic, 103
    formal semantics of, 104
polymorphism, in ML, 13–14, 39
POP_ASSUM, **276**
POP_ASSUM_LIST, 282
popping, of assumptions, 276
ports, in ML input-output, 69
PPLAMB, 176
PPλ (same as PPLAMBDA), of LCF system, 117, 180
precedence, in HOL logic
    under interface maps, 205
precedence, in ML, 23
    of function application in ML, 9
    of infixes, in ML, 29
    of prefixes, in ML, 29
    of while construct, in ML, 11
precision, in ML arithmetic, 82–83
pred_sets library, 91
predeclared bindings, in ML, 51–54
prefixes, in ML, 29–30, 36, 44, 53–54
preterm, 300
    the flag, **84**
    the type, 155
preterm_handler, **155**, 301
preterm_to_term, **155**, 300
preterms, 154–155, 300
pretty printing
    flags for, in HOL system, 84, 203
    of ML, 72–75
prettyp library, 91
prim_rec, 180–183, 195
primitive constants, of HOL logic, 107, 111, 168, 176
primitive inference, in natural deduction, 156
primitive inferences, of HOL logic, *see* inference rules
primitive recursion theorem
    a degenerate case, 194
    automated use of, in HOL system, 180–183, 185, 195–196
    for binary trees, 195–198
    for lists, 185
    for numbers, 180–183
primitive recursive definitions, in HOL logic
    automated, 180–183
    justification of, 180
primitive recursive functions, 180
print_all_subgoals, **84**
print_all_thm, **73**, 157
print_begin, **73**
print_bool, **73**
print_break, **74**
print_cond, **84**, 203
print_defined_types, **82**
print_end, **73**
print_fasl, **84**
print_ibegin, **73**
print_infix, **84**, 203
print_int, **73**
print_let, 83, **84**, 203
print_lettypes, 82, **84**
print_lib, **84**, 90
print_list, **74**, 84, 203
print_load, **84**
print_newline, **74**
print_quant, **84**, 203
print_restrict, **84**
print_set, 84
print_state, **265**
print_string, **73**
print_term, **73**
print_theory, 168
print_thm, **73**
print_tok, **73**
print_top_types, **84**, 87
print_top_val, **84**

print_type, **73**
print_uncurry, **84**, 203
print_void, **73**
printing, in HOL logic
  of function abstractions, 150
  of hypotheses of theorems, 86, 157
  of let-terms, 151
  of list expressions, 185
  of proofs, 82
  of quantification, 203
  of quotation syntax, 142
  of theorems, 157
  of theories, 168
  of types, 140, 203
  pretty printing, 203
  structural depth adjustment in, 75, 212
  under interface maps, 206
printing, of ML
  of abstract-typed values, 20, 50
  of functions, 9
  of string quotes, 72
  user-modified, 86
prod, 177
product function operator, in ML, 56
product types
  in HOL logic, 177–178
  in ML, 40
prog_logic88 library, 91
prompt, **84**
prompt, in ML, 7
  changing of, 85
  printing of, 84
proof
  in natural deduction, 115, 156
  the notion of, in HOL system, 218
proof, 257
proof construction, 260
  as tree search, 255
proof functions (same as justifications, validations), 256
  THEN example of, 274
proof mode, in HOL system, 159, 167
proof steps, as ML function applications, 218, 256, 260
proofs, in HOL logic
  as generated by derived rules, 218
  as generated by tactics, 256, 260
  as ML function applications, 218, 256, 260
PROVE, **261**
prove_abs_fn_one_one, **166**
prove_abs_fn_onto, **166**
prove_cases_thm, **201**
prove_constructors_distinct, **201**
prove_constructors_one_one, **201**
prove_induction_thm, **198**, 200, 201
prove_rec_fn_exists, **196**
prove_rep_fn_one_one, **166**
prove_rep_fn_onto, **166**
prove_thm, **261**
PURE_ASM_REWRITE_TAC, 269
PURE_REWRITE_TAC, 269

quantifiers
  conversions for, 250, 305
  restricted, 146, 170
quit, **76**
quotation, in HOL logic, 142–152
  of non-primitive terms, 145–152
  of primitive terms, 144
  of types, 144
  parser for, 142, 145
  typechecker for, 154–155
quotient library, 91
quotient types library, 91

r, the subgoal package function, 265
rand, 142
RAND_CONV, **245**

rator, 142
RATOR_CONV, **245**
read, **69**
rectype, 19, 24, 48
recursive definitions, in classical logics, 180
recursive definitions, in HOL logic
  automated for lists, 184–186
  automated, for numbers, 180–183
  automated, for recursive types, 195–196
  automated, for trees, 187–188
recursive types, in HOL logic
  tools for construction of, 191–193
reduce, 60
reduction, by conversions, 239
REFL, 116, 117, **171**, 244
reflexivity, in HOL logic
  abstract form of, 116
  ML function for, 171
remove, **63**
remove_sticky_type, **204**
REP_..., 193
rep_..., 20, 49
REP_prod, 178
REPEAT, 270, **275**
REPEATC, **240**
repetition
  of conversions, 240
  of tactics, 275
replicate, **59**
representing types, in HOL logic
  abstract form of, 128–130
  pair example of, 177–178
RES_ABSTRACT, 147, 170, 177
RES_EXISTS, 147, 170, 177
RES_FORALL, 147, 170, 177
RES_SELECT, 147, 170, 177
resolution tactics, 269, 275
restrictions, of models, 122
rev, **62**
rev_assoc, **62**
rev_itlist, **60**
REWRITE_CONV, **252**
REWRITE_RULE, 220–221
REWRITE_TAC, 258–259, 267, **268**
rewriting
  as based on conversions, 252–253
  importance of, in goal directed proof, 258–259
  list of tactics for, 253
  main tactic for, 258–259, 268–269, 275
  rules for, 220–221
  use of DEPTH_CONV in, 252
rhs, 142
RIGHT_BETA, **232**
RIGHT_LIST_BETA, **233**
root directory, of HOL system, 89
rotate, **264**
rotation, in subgoal package, 262, 264
rules in HOL logic, some not properly derived, 222–223
rules of inference, *see* inference rules

S, constant in HOL logic, 186
S, the constant in ML, **55**
S_DEF, 186
S_THM, 187
satisfaction of sequents, by model, 114
save, **76**
save_thm, **161**
save_top_thm, **264**
saving theorems, 161
scope
  in ML abstract types, 49
  of bindings, 31
search paths
  for HOL system, 89
  for help system, 88
search_path, **70**
security, in goal directed proof, 257–258

SELECT_AX, 169
    abstract form of, 121
SELECT_ELIM, **231**
SELECT_INTRO, **230**
selective sequencing tactical, 274
selectors, in HOL logic, 178
sequences, in ML, 38
sequencing
    of conversions, 239
    of ML expressions, 11, 26, 38, 45
    of tactics, 260, 272–274
sequents
    in natural deduction, 114, 156
    representation of, in HOL logic, 156
sessions, with HOL system, 158
set theory (of predicates) library, 91
set theory libraries, 91, 297
set theory notation, 152, 295
set_equal, **65**
set_fail, **67**
set_fail_prefix, **67**
set_flag, **85**
set_goal, **263**
set_help, **88**, 292
set_help_search_path, **88**, 292
SET_INDUCT_TAC, 298
set_interface_map, **205**
set_lambda, **209**
set_margin, **74**
set_pretty_mode, **75**
set_prompt, **85**
set_search_path, **70**
SET_SPEC_CONV, 298
set_state, **265**
set_sticky_type, **204**
set_string_escape, **29**, 300
set_thm_count, **213**
set_turnstile, **209**
setify, **65**
sets library, 91
sets, as ML lists, 64
show_types, **84**, 203
side effects, in ML, 31, 35, 36
side effects, in subgoal package, 263
signatures, of HOL logic
    formal semantics of, 107
    of HOL theories, 118
    standard, 111
simple bindings, in ML, 33
Skolemization, 249, 304
snd, **51**
SND, the axiom in HOL logic, 178
SND, the constant in HOL logic, 177
SND_DEF, 178
solving, of goals, 256
soundness
    of HOL deductive system, 117
    of HOL logic, 210
spaces
    in ML expressions, 26
    in ML strings, 29
SPEC, 223, **227**
special symbols, in ML, 27–28
special_symbols, **28**, **139**
specialization rule, in HOL logic, 227
specialization tactic, 266
specification of constants, in HOL logic, 163–170
split, **64**, 272
stack, of assumptions, 276
stacks, in subgoal package, 262
standard models, of HOL logic, 111
standard signatures, of HOL logic, 111
standard theory, 121
sticky, **84**, 203
sticky types, in HOL logic, 84, 204–205
sticky_list, **205**
stores, 31, 36
strategies, for proof, 255
string, 40

`string` library, 91
`string_of_int`, **52**
strings, in ML, 29
  as input to HOL type definition package, 192
  changing the escape, 29
  for input-output, 69
  processing functions for, 65–66
  to identify failures, 15, 37
`STRIP_TAC`, **267**
strong validity, of tactics, 257
structural induction, in HOL logic, *see* induction rule
`SUB`, 184
`SUB_CONV`, **245**
subgoal package
  as HOL interface, 262
  flags for, 265
  functions of, 263–265
subgoal tree
  in proof construction, 262, 275
  in subgoal package, 262
`subgoals`, 257
`SUBS`, 223
`SUBS_OCCS`, 223
`SUBST`, 116, 117, 172
`SUBST1_TAC`, **277**
`SUBST_CONV`, 223
`SUBST_TAC`, **267**
substitution rule, in HOL logic
  abstract form of, 116
  formal semantics of, 109, 110
  implementation of, 117
  ML function for, 172
substitution, tactic for, 267
`subtract`, **64**
successive application
  conversion operator for, 240
  tactical for, 274
`sum`, 189
`SUM`, the constant in HOL logic, 186
`SUM`, the theorem in HOL logic, 186
`sum_Axiom`, 189
`sum_axiom`, 189
`SYM`, 223, **224**
symbols in HOL logic, changeable, 209–210
symmetry of equality rule, in HOL logic, 224
syntax blocks, 88–89, 299–300
syntax of ML
  explanation of, 23
  short form, 32
`system`, **80**

`T`, 176
  abstract form of, 119, 124
  defined in terms of primitives, 169
  definitional axiom for, 169
  formal semantics of, 120
  rules of inference for, 226–228
`T_DEF`, 169
tab
  in ML expressions, 27
  in ML strings, 29
`TAC_PROOF`, **261**
`tactic`, 257
tacticals, 260, 271–275
  for alternation, 271
  for repetition, 275
  for sequencing, 272–274
  for successive application, 274
  list of some, 271–275
  purpose of, 271, 276, 283
tactics
  alternation of, 271
  as documentation of proofs, 255, 283
  assumption transforming, 275
  compound, 260, 271
  debugging of, 261

definition of new, 260
for manipulating assumptions, 275–286
identity for, 270
indirect implementation of, 260, 286
list of some, 265–271
ML type of, 257
purpose of, 255–256
repetition of, 275
sequencing of, 260, 272–274
tacticals for, 271–275
term transforming, 275
`taut` library, 91
tautologies, in rewriting tactic, 259, 268
tautology checker library, 91
term component, of goal, 256
term constructors, in HOL logic, 141, 144–146
term destructors, in HOL logic, 141
term substitution, in HOL logic, *see* substitution rule
term-in-context, 106
termination, of HOL sessions, 76
terminator, of ML expressions, 7
terms, in HOL logic
antiquotation, 154
as logical formulas, 111, 156
conditional, 149
constructors for, 141, 144–146
function abstraction, 144, 150
input of, 142
`let-`, 151
non-primitive, 145
pair, 150–151
primitive, 144
syntax of, 144–152
`THEN`, 260, 270, **272**, 277
ML implementation of, 272–274
`THENC`, **239**, 244
`THENL`, 266, 270, 274
`theorem`, **168**
theorem continuations, 276
use of, in derivation of induction tactics, 199–201
theorem notation, in HOL logic, 157
`theorems`, **167**
theorems, in HOL logic
abstract form of, 118
theorems, in HOL logic
as inference rules, 222
as rewrite rules, 252
destructors for, 156
equational, 220, 238
rules inexpressible as, 222, 238
saving of, 161
theorems, in natural deduction, 156
theorems, printing of, 72
theories, in HOL logic
abstract form of, 113, 118
caching of, 211
creation of, 159–161
directory of built-in, 70
extension of, 161–165
functions for accessing, 167–168
hierarchies of, 158, 167, 175
naming of, 158
paging of, 211
representation of, 158–159
theory caching, 211
theory files, 158
pretty printing of, 84, 203
theory segments, 158
`theory_pp`, **84**, 203
`thm`, 156, 171, 217
`thm_count`, **213**
`thm_tactic`, 276
`timer`, **213**
`timing`, **84**, 203
timing of HOL evaluations, 213
`tl`, 12, 13, **51**

TL, the constant in HOL logic, 184
TL, the definition in HOL logic, 186
top level terminator, in ML, 7
top level, of ML, 7
  abstract type expressions at, 50
  customization of, 85
  declarations at, 31, 42
  expressions at, 42
  printing at, 84
TOP_DEPTH_CONV, **242**, 246
top_goal, **264**
top_print, 86
top_thm, **264**
TRANS, 223, **224**
transitivity of equality rule, in HOL logic, 224
tree of subgoals, in proof construction, 262, 275
tree, the HOL theory, 187
tree, the HOL type, 187
tree_Axiom, 188
tree_Induct, 188
trs library, 91
true in ML, 29, 43
truth values, in HOL logic, 140, 176
  abstract form of, 112, 119, 124
  constants for, 169
  definition of, 169
truth values, in ML, 29
truth, in HOL logic, *see* T
TRY_CONV, **244**
tryfind, **61**
tty_read, **70**
tty_write, **70**
turnstile notation, 115, 156–157
..._TY_DEF, 165, 193
TY_DEF_THM, 189
tydefs, 189
type, 17–19, 24, 48
type abbreviations
  in HOL logic, 174
  in ML, 17, 40, 47
type checker, for HOL logic *vs* ML, 144
type checking, in HOL logic
  antiquotation in, 154
  of quotation syntax, 142–145
  special forms in, 145–152
  sticky types in, 204–205
  user programmed, 155, 300
  verbose errors in, 203
type checking, in ML, 32, 41–45
  inference involved in, 14, 41–43
  reasons behind algorithm for, 45–47
  reasons for, 39
type constants, in HOL logic, 102, 140
type constraint
  in HOL logic, 100, 140, 143–144
  in ML, 7, 24, 25, 42, 44
type constructors
  in HOL logic, 140, 144
  in ML, 20–22, 40–41, 48–50
type context, 103
type definition extension, in HOL logic, 190–193
  ML function for, 164–165
type definition package, in HOL system, 190–202
  input to, 192
  output of, 192–193
  use of, in case definition, 197
  use of, in curried infix definition, 198
  use of, in partial definition, 197
  use of, in recursive function definition, 195–196
  use of, in recursive type definition, 191–193
  use of, to build induction tools, 198–201
  use of, to build number theory, 194
type definitions, in HOL logic, 164–165

abstract structure of, 129
autoloading of, 211
automated, *see* type definition package
defining bijections for, 165–166
introduction of, 164
primitive recursive, 180–183
properties of bijections for, 166–167
type destructors, in HOL logic, 140
type instantiation, in HOL logic
abstract form of, 116
in rewriting rule, 220
ML function for, 173
type operators, in HOL logic
associativity of, 102
declaration, 160
definitional axioms for, 164
for pairs, 177
type operators, in ML, *see* type constructors
type specification, 131
type structure, in HOL logic, 102
type structures, of HOL theories, 118
type substitution, in HOL logic
formal semantics of, 109
type variables, in HOL logic, 102
abstract form of, 101, 111
constructor for, 140, 144
destructors for, 140
differences from classical, 168
names of, 138
type variables, in ML, 14, 24–25, 34, 35, 43
type-in-context, 103
`type_abbrevs`, **174**
`TYPE_DEFINITION`, 165, 177
`type_error`, **84**, 203
`type_of`, **142**
`types`, **167**
types, in HOL logic, 140
dependent, 293
abbreviation of, 174
constructors for, 140, 144
default, 203–205
dependent, 146
destructors for, 140
determination of, 142
explicit printing of, 204
generic, 204
instantiation of, 104, 110, 173
polymorphic, 103, 104
printing of, 203
sticky, 204–205
tools for construction of, 184, 191, 195
types, in ML
abbreviation of, 17, 24, 40, 47
abstract, 19–20, 24, 40, 49–50
concrete, 17–19, 24, 40, 48
printing of, 84, 87
printing of currently defined, 82

`UNCURRY`, 150–151, 177
`uncurry`, **57**
`UNDISCH`, 219, **224**
`undo_autoload`, **77**, **211**
`unhide_constant`, **210**
`union`, **64**
universal quantifier, in HOL logic, 176
abbreviation for multiple, 119, 146
abstract form of, 112, 119, 124
defined in terms of primitives, 169
definitional axiom for, 169
formal semantics of, 120
in four primitive axioms, 169
inference rules for, 228, 230
syntax of, 145
tactics for, 266
universe, in semantics of HOL logic, 99
`unlink`, **81**
`unwind` library, 91

unwinding rule library, 91

validations, 256
validity, of tactics, 257–258, 261
values, of ML variables, 31
Van Tassel, J., 3
variables, in HOL logic
    abstract form of, 105, 106
    constructor for, 141, 144
    destructor for, 141
    formal semantics of, 108
    multiple bound, 146
    names of, 137–139
    syntax of, 144
    with constant names, 187, 210
    with same names, 106
variables, in ML, 31
    assignable, 8, 31, 36, 47
    evaluation of, 35
    in expression evaluation, 34
    loading of predefined, 77
    ordinary, 35
    syntax of, 24–25
    type checking of, 43
`version`, **81**
version number, of HOL system, 81
`void`, 37, 40, 41, 43

`W`, **56**
Wadsworth, C., 3, 220
well-ordering library, 91
`where`, 8, 26
`whereabsrectype`, 26
`whereabstype`, 26
`whererec`, 26
`whereref`, 26
`wheretype`, 26
while construct, in ML, 26, 37, 45
`while···do···`
    evaluation of, 37
    syntax of, 26
    type checking of, 45
wildcard, in ML, 22, 24, 26, 32, 34, 43
word$n$ library, 91
`words`, **65**
`words2`, **65**
`write`, **69**

`X_SKOLEM_CONV`, **249**, 304

`Z` notation, 123
Zermelo-Fraenkel set theory, 100
`ZFC`, 100