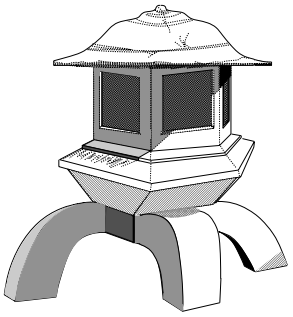
[For HOL Kananaskis-8] October 11, 2012

The HOL System DESCRIPTION



Preface

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

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

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

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

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

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

This document, *DESCRIPTION*, is 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. 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 [2]. 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 [3].) However, the logic does not restrict applications to hardware; HOL has been applied to many other areas.

This document presents the HOL logic in its ML guise, and explains the means by which meta-language functions can be used to generate proofs in the logic. Thus, it describes how the abstract system of *LOGIC* is actually implemented in the ML programming language, providing comprehensive descriptions of the system's major features.

The approach to mechanizing formal proof used in HOL is due to Robin Milner [7], 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.

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. HOL90 (released in 1990) was a port of HOL88 to SML [9] by Konrad Slind at the University of Calgary. It has been further developed through the 1990's. HOL 4 is the latest version of HOL, and is also implemented in SML; it features a number of novelties compared to its predecessors. HOL 4 is also the supported version of the system for the international HOL community.

We have retroactively decided to number HOL implementations in the following way

1. HOL88 and earlier: implementations based on a Lisp substrate, with Classic ML.
2. HOL90: implementations in Standard ML, principally using the SML/NJ implementation.
3. HOL98 (Athabasca and Taupo releases): implementations using Moscow ML, and with a new library and theory mechanism.
4. HOL (Kananaskis releases)

Therefore, with HOL 4, we do away with the habit of associating implementations with year numbers. Individual releases within HOL 4 will retain the *lake-number* naming scheme.

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'.

Acknowledgements

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

Current edition

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

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

Second edition

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

First edition

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

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

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

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

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

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

Contents

Chapter 1

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 1.3); the quotation, parsing, and printing of logical types and terms (Section 1.4); the representation of theorems (Section 1.6); the representation of theories (Section 1.9); the fundamental HOL theory `bool` (Section 3.2.1); the primitive rules of inference (Section 1.7); and the methods for extending theories (throughout Section 1.9 and also later in Section 5.3). 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 the ML types `hol_type` and `term` which implement the types and terms of the HOL logic, as defined in *LOGIC*. It also provides primitive ML functions for creating and manipulating values of these types. Upon this basis the HOL logic is implemented. 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 is 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.

1.1 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 1.1.1 below). Using 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 existing names). The name of a type variable in the HOL logic is formed by a prime (`'`) followed by an alphanumeric which itself contains no prime (see Section 1.1.1 for

examples). The name of a type constant or a term constant in the HOL logic can be any identifier, although some names are treated specially by the HOL parser and printer and should therefore be avoided.

1.1.1 Identifiers

In addition to special forms already present in the relevant grammar, a HOL identifier can be of two forms:

(i) A finite sequence of *alphanumerics* starting with a letter. The underscore character is considered a digit character, and so can occur after an identifier's first letter. Greek characters (roughly Unicode range `U+0370` to `U+03FF`) are also letters, except for λ (`U+03BB`), which is treated as a symbol. HOL is case-sensitive: upper and lower case letters are considered to be different.

Digits are the ASCII characters 0–9, the underscore character, and the Unicode subscripts and superscripts. The apostrophe character is special. It is not a letter, but can appear as part of an alphanumeric term identifier after the first letter. It must appear at the start of a type variable's name, and can also appear in the term context as a sequence of apostrophes on their own.

(ii) A *symbolic* identifier, i.e., a finite sequence formed by any combination of the ASCII symbols and the Unicode symbols. The basic ASCII symbols are

```
# ? + * / \ = < > & % @ ! : | - ^ `
```

Use of the caret and back-tick characters is complicated by the fact that these characters have special meaning in the quotation mechanism; see Section 5.1.3. The ASCII grouping symbols (braces, brackets, and parentheses), and the tilde (`~`), full-stop (`.`), comma (`,`), semi-colon (`;`) and hyphen (`-`) characters are called *non-aggregating* characters. Unless the desired token is already present in the grammar, these characters do not combine with themselves or other symbolic characters. Thus, the string `"(("` is viewed as *two* tokens, as are `"+;"` and `"-+"`.

Unicode code characters that are not letters or digits are regarded as symbolic. None of these are non-aggregating.

(iii) A *number* is a string of one or more digits. If not the initial digit, an underscore can be used within the sequence to provide spacing. In order to distinguish different kinds of numbers a single character suffix may be used: for example `3n` is a natural number while `3i` is an integer. The `0x` and `0b` prefixes may also be used to change the base of the number. If the `0x` prefix is used, hexadecimal 'digits' `a`–`f` and `A`–`F` can also be used. See also Section 3.3.3.

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)

Special identifiers The following valid identifiers are used by the grammar in the theory of booleans, and thus in all descendent theories as well. They should not be used as the name of a variable or a constant unless the user is very confident of their ability to mess with grammars.

```
let  in  and  \  .  ;  =>  |  :  :=  with  updated_by  case  of
```

Type variable names The name of a type variable in the HOL logic is a string beginning with a prime (') followed by an alphanumeric which itself contains no prime; for example all of the following are valid type variable names except for the last:

```
'a  'b  'cat  'A11  'g_a_p  'f'oo
```

User tokens In general, a HOL user has a great deal of freedom to create their own syntax, involving special tokens quite apart from variables and names for constants. For example, the if-then-else syntax for the conditional operator has special tokens (the "if", "then" and "else") that are not names for variables, nor constants (the underlying constant is actually called `COND`). In order to make sure that the operations of printing and parsing tokens are suitably inverse to each other, users should not create tokens that include whitespace, or the comment strings (`(*` and `*)`).

1.2 Types

The allowed types depend on which type constants have been declared in the current theory. See Section 1.9 for details of how such declarations are made. There are two primitive constructor functions for values of type `hol_type`:

```
mk_vartype : string -> hol_type
mk_thy_type : {Tyop:string, Thy:string, Args:hol_type list} -> hol_type
```

The function `mk_vartype` constructs a type variable with a given name; it gives a warning if the name is not an allowable type variable name (i.e. not a ' followed by an alphanumeric). The function `mk_thy_type` constructs a compound type from a record `{Tyop,Thy,Args}` where `Tyop` is a string representing the name of the type operator, `Thy` is a string representing the theory that `Tyop` was declared in, and `Args` is 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 *LOGIC*, however, function types were not regarded as compound types).

The evaluation of `mk_thy_type{Tyop` $= name$, `Thy` $= thyname$, `Args` $= [\sigma_1, \cdots, \sigma_n]\}$ fails if

(i) $name$ is not a type operator of theory $thyname$

(ii) $name$ is a type operator of theory $thyname$, but its arity is not n.

For example, `mk_thy_type{Tyop="bool", Thy="bool", Args=[]}` evaluates to an ML value of type `term` representing the type *bool*.

Type constants may be bound to ML values and need not be repeatedly constructed: e.g., the type built by `mk_thy_type{Tyop="bool", Thy="bool", Args=[]}` is abbreviated by the ML value `bool`. Similarly, function types may be constructed with the infix ML function `-->`. A few common type variables have been constructed and bound to ML identifers, e.g., `alpha` is the type variable `'a` and `beta` is the type variable `'b`. Thus the ML code `alpha --> bool` is equal to, but much more concise than

```
mk_thy_type{Tyop="fun", Thy="min",
            Args=[mk_vartype "'a",
                  mk_thy_type{Tyop="bool", Thy="bool", Args=[]}}
```

There are two primitive destructor functions for values of type `hol_type`:

```
dest_vartype : hol_type -> string
dest_thy_type : hol_type -> {Tyop:string, Thy:string, Args:hol_type list}
```

The function `dest_vartype` extracts the name of a type variable. A compound type is destructed by the function `dest_thy_type` into the name of the type operator, the name of the theory it was declared in, and a list of the argument types; `dest_vartype` and `dest_thy_type` are thus the inverses of `mk_vartype` and `mk_thy_type`, respectively. The destructors fail on arguments of the wrong form.

1.3 Terms

The four primitive kinds of terms of the logic are described in *LOGIC*. The ML functions for manipulating these are described in this section. There are also *derived* terms that are described in Section 3.2.1.2.

At any time, the terms that may be constructed depends on which constants have been declared in the current theory. See Section 1.9 for details of how such declarations are made.

There are four primitive constructor functions for values of type `term`:

```
mk_var : (string * hol_type) -> term
```

`mk_var(`x`,`σ`)` evaluates to a variable with name x and type σ; it always succeeds.

```
mk_thy_const : {Name:string, Thy:string, Ty:hol_type} -> term
```

`mk_thy_const{Name` $= c$, `Thy` $= thyname$, `Ty` $= \sigma$`}` evaluates to a term representing the constant with name c and type σ; it fails if:

(i) c is not the name of a constant in the theory $thyname$;

(ii) 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 1.9).

```
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 * hol_type)
dest_thy_const : term -> {Name:string, Thy:string, Ty:hol_type}
dest_comb      : term -> (term * term)
dest_abs       : term -> (term * term)
```

These are the inverses of `mk_var`, `mk_thy_const`, `mk_comb` and `mk_abs`, respectively. They fail when applied to terms of the wrong form. Other useful destructor functions are `rator`, `rand`, `bvar`, `body`, `lhs` and `rhs`. See *REFERENCE* for details.

The function

```
type_of : term -> hol_type
```

returns the type of a term. The function

```
aconv : term -> term -> bool
```

implements the α-convertibility test for λ-calculus terms. From the point of view of the HOL logic, α-convertible terms are identical. A variety of other functions are available for performing β-reduction (`beta_conv`), η-reduction (`eta_conv`), substitution (`subst`), type instantiation (`inst`), computation of free variables (`free_vars`) and other common term operations. See *REFERENCE* for more details.

1.4 Quotation

It would be tedious to always have to input types and terms using the constructor functions. The HOL system, adapting the approach taken in LCF, has special quotation parsers for HOL types and terms which enable types and terms to be input using a fairly standard syntax. For example, the ML expression ``:bool -> bool`` denotes exactly the same value (of ML type hol_type) as

```
mk_thy_type{Tyop = "fun",Thy = "min",
            Args = [mk_thy_type{Tyop = "bool", Thy = "bool", Args = []},
                    mk_thy_type{Tyop = "bool", Thy = "bool", Args = []}]}
```

and the expression ``\x. x + 1`` can be used instead of[1]

```
let val numty = mk_thy_type{Tyop="num",Thy="num",Args=[]}
in
 mk_abs
  (mk_var("x",numty),
   mk_comb(mk_comb
    (mk_thy_const
      {Name="+",Thy="arithmetic",Ty=numty --> numty --> numty},
       mk_var("x", numty)),
     mk_comb(mk_thy_const{Name="NUMERAL",Thy="arithmetic",Ty=numty-->numty},
      mk_comb(mk_thy_const{Name="BIT1",Thy="arithmetic",Ty=numty-->numty},
              mk_thy_const{Name="ZERO",Thy="arithmetic",Ty=numty}))))
end
```

The HOL printer, which is integrated into the ML toplevel loop, also outputs types and terms using this syntax. Types are printed in the form ``:*type*``. For example, the ML value of type hol_type representing $\alpha \rightarrow (\mathit{ind} \rightarrow \mathit{bool})$ would be printed out as ``:'a -> ind -> bool``. Similarly, terms are printed in the form ``*term*``. Thus, 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``
```

A leading colon is used to distinguish a type quotation from a term quotation: the former have the form ``: ⋯ `` and the latter have the form `` ⋯ ``.

Section 5.1 has more detailed information about the capabilities of the term and type parsing and printing facilities in the system. The remainder of this section provides a brief overview of what is possible.

[1] In order to be processed successfully, this quotation requires the theory of arithmetic to have already been loaded, which can be accomplished in the interactive system by `load "arithmeticTheory"`.

1.4.1 Type inference

Notice 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. However, 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)``;
Type inference failure: unable to infer a type for the application of

$= (x :num)

which has type

:num -> bool

to

(x :num) = (1 :num)

which has type

:bool

unification failure message: unify failed
```

The desired value can be directly constructed by the primitive constructor functions:

2
```
- mk_eq
   (mk_var("x",bool),
    mk_eq(mk_var("x",numty),
          mk_numeral (Arbnum.fromString "1")));
> val it = ``x <=> (x = 1)`` : term
```

The original 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.

At times, the user may want to control the exact type of a subterm. To support such functionality, 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. This treatment of types within quotations is inherited from LCF.

1.4.2 Viewing the grammar

The behaviour of the HOL quotation parser and printer is determined by the current grammar. Thus, a familiarity with the basic vocabulary of the standard collection of HOL theories is important if one is to use HOL effectively. One can examine the current grammar used by the parser with the functions `type_grammar` and `term_grammar`.

For example, in the following session, we see that the type grammar used in the startup context of HOL has the type operators `fun`, `sum`, `prod`, `list`, `recspace`, `num`, `option`, `one`, `label`, `ind`, and `bool`.

1
```
- type_grammar();
> val it =
    Rules:
    (50)   TY  ::=  TY -> TY [fun] (R-associative)
    (60)   TY  ::=  TY + TY [sum] (R-associative)
    (70)   TY  ::=  TY # TY [prod] (R-associative)
    (100)  TY  ::=  TY list | TY recspace | num | (TY, TY)prod | TY option |
                    one | (TY, TY)sum | label | (TY, TY)fun | ind | bool
     : grammar
```

Also, `fun`, `sum`, and `prod` have infix notation (`->`), (`+`), and (`#`), respectively, with different binding strengths: `#` (with 70) binds stronger than `+` (60), which binds stronger than `->` (50). All postfix type operators (with 100) bind more strongly than the infixes.

The next session, in Figure 1.1, shows the (abbreviated) output from invoking the `term_grammar` function in the startup HOL environment. The deleted output includes a listing of all constants known to the system, including prefix operators, along with all overloadings currrently in force. The portrayed grammar ranges from binding operators at very low (0) binding strength, through to function application (2000) and record selection (2500), which bind very tightly.

1.4.3 Namespace control

In order to provide convenience, the parser deals with overloading and ambiguity. Overloading of numeric literals is discussed in Section 3.3.3.1, although any symbol may be overloaded, not just numerals. At times such flexibility is quite useful; however, it can happen that one wishes to explicitly designate a particular constant. In that case, the notation $thy\$const$ may be used in the parser to designate the constant $const$ declared in theory thy. In the following example, the less-than operator is explicitly specified.

1
```
- ``prim_rec$< x y``
> val it = ``x < y`` : term
```

Note how the < symbol is not treated as an infix by the parser when given in "fully-qualified" form. Syntactically, such tokens are never given special treatment by the parser of HOL's concrete syntax.

```
- term_grammar();                                                              1
> val it =
    (0)    TM  ::=  "LEAST" <..binders..>  "." TM |
                    "?!" <..binders..>  "." TM | "?" <..binders..>  "." TM |
                    "!" <..binders..>  "." TM | "@" <..binders..>  "." TM |
                    "\" <..binders..>  "." TM
    (2)    TM  ::=  "let" TM "in" TM  [let]
    (4)    TM  ::=  TM "::" TM (restricted quantification operator)
    (5)    TM  ::=  TM TM  (binder argument concatenation)
    (7)    TM  ::=  "case" TM "of" TM  [case__magic]
    (8)    TM  ::=  TM "|" TM  [case_split__magic]   (R-associative)
    (9)    TM  ::=  TM "and" TM   (L-associative)
    (10)   TM  ::=  TM "=>" TM  [case_arrow__magic]   (R-associative)
    (50)   TM  ::=  TM "##" TM | TM "," TM   (R-associative)
    (70)   TM  ::=  "if" TM "then" TM "else" TM  [COND]
    (80)   TM  ::=  TM ":-" TM   (non-associative)
    (100)  TM  ::=  TM "=" TM   (non-associative)
    (200)  TM  ::=  TM "==>" TM   (R-associative)
    (300)  TM  ::=  TM "\/" TM   (R-associative)
    (400)  TM  ::=  TM "/\" TM   (R-associative)
    (425)  TM  ::=  TM "IN" TM   (non-associative)
    (440)  TM  ::=  TM "++" TM   (L-associative)
    (450)  TM  ::=  TM "::" TM  [CONS] | TM ">=;" TM | TM "<=" TM |
                    TM ">" TM | TM "<;" TM | TM ">=" TM | TM "<=" TM |
                    TM ">" TM | TM "<" TM | TM "LEX" TM | TM "RSUBSET" TM |
                    TM ":=" TM  [record field update] |
                    TM "updated_by" TM  [functional record update] |
                    TM "with" TM  [record update]
                    (R-associative)
    (500)  TM  ::=  TM "-" TM | TM "+" TM | TM "RUNION" TM   (L-associative)
    (600)  TM  ::=  TM "DIV" TM | TM "*" TM | TM "RINTER" TM
                    (L-associative)
    (650)  TM  ::=  TM "MOD" TM   (L-associative)
    (700)  TM  ::=  TM "**" TM | TM "EXP" TM   (R-associative)
    (800)  TM  ::=  TM "O" TM | TM "o" TM   (R-associative)
    (900)  TM  ::=  "~" TM
    (1000) TM  ::=  TM ":" TY  (type annotation)
    (2000) TM  ::=  TM TM  (function application) |    (L-associative)
    (2500) TM  ::=  TM "." TM  [record field selection]   (L-associative)
           TM  ::=  "[" ... "]"  (separator = ";") |
                    "<|" ... "|>"  (separator = ";")
           TM  ::=  "(" ")"  [one] |
                    "(" TM ")"  [just parentheses, no term produced]
    ... <further output omitted>
  : grammar
```

Figure 1.1: Result of a call to `term_grammar()`

1.5 Ways to Construct Types and Terms

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	`: 'alphanum`	`mk_vartype("'alphanum")`
Type constant	`: op`	`mk_type("op",[])`
	`: thy$op`	`mk_thy_type{Thy="thy", Tyop="op", Args=[]}`
Function type	`: σ1 -> σ2`	`σ1 --> σ2`
Compound type	`:(σ1, ... , σn)op`	`mk_type("op", [ σ1 , ... , σn])`
	`:(σ1, ... , σn)thy$op`	`mk_thy_type{Thy="thy", Tyop="op", Args=[ σ1,...,σn ]}`

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)`
Constant	`thy$const:σ`	`mk_thy_const{Name="const",Thy="thy",Ty=σ}`
Combination	`t1 t2`	`mk_comb(t1, t2)`
Abstraction	`\x.t`	`mk_abs(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,[t1, ... ,tn])`
$\backslash x_1 \cdots x_n . t$	$\backslash x_1 .\ \cdots\ \backslash x_n . t$	`list_mk_abs([x1, ... ,xn],t)`

1.6 Theorems

In *LOGIC*, 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 set of sequents.[2] 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 *LOGIC*) 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[3] 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 an implementation of the ML signature `HOLset` supporting operations such as `member` and `union`.

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 destructor function

```
dest_thm : thm -> (term list * term)
```

which returns a pair consisting of a list of the hypotheses and the conclusion, respectively, of a theorem. The order of assumptions in the list should not be relied on. A theorem's hypotheses are also available in the set form with the function

```
hyp_set : thm -> term HOLset.set
```

Using `dest_thm`, two further destructor functions are derived

[2] Note that these sequents form a list, not a set; that is, are ordered.

[3] Note that 'hypotheses' and 'conclusion' are also used for the components of sequents.

```
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 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 in Section 1.7.)

It was mentioned in *LOGIC* that the deductive system of HOL includes four axioms.[4] In that manual, the axioms were presented in abstract form. Concretely, axioms are just theorem values that are introduced through the use of the ML function `new_axiom` (see Section 1.9.1 below). For example, the axiom `BOOL_CASES_AX` mentioned in *LOGIC* 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) : thm
```

Note the special print format, with the approximation to the abstract $\vdash$ notation, `|-`, used to indicate ML type `thm` status; as well as the absence of HOL quotation marks in the `|-` context. The session below illustrates the use of the destructor functions:

1

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

- hyp th;
> val it = [] : term list

- concl th;
> val it = ``!t. (t = T) \/ (t = F)`` : term

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

In addition to the print conventions mentioned above, the printing of theorems prints hypotheses as periods (i.e. full stops or dots). The flag `show_assums` allows theorems to be printed with hypotheses shown in full. These points are illustrated with a theorem inferred, for example purposes, from another axiom mentioned in *LOGIC*, `SELECT_AX`.

[4] This is a simplification: in fact the various axioms are an extension of the basic logic.

```
- val th = UNDISCH (SPEC_ALL SELECT_AX);                                    2
> val th =  [.] |- P ($@ P) : thm

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

- th;
> val it =  [P x] |- P ($@ P) : thm
```

1.7 Primitive Rules of Inference of the HOL Logic

The primitive rules of inference of the logic were described abstractly in *LOGIC*. 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 *LOGIC*'s rules 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 *LOGIC*: hypotheses above a horizontal line and conclusion beneath. The machine-readable ASCII notation is used for the logical constants.

1.7.1 Assumption introduction

```
   ASSUME : term -> thm
```

$$\frac{}{t \text{ |- } t}$$

`ASSUME` t evaluates to t`|-` t. Failure if t is not of type `bool`.

1.7.2 Reflexivity

```
   REFL : term -> thm
```

$$\frac{}{\text{|- } t = t}$$

REFL t evaluates to |- t = t. A call to REFL never fails.

1.7.3 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 ($\backslash x.t_1)t_2$.

1.7.4 Substitution

```
SUBST : (thm * term)list -> term -> thm -> thm
```

$$\frac{\Gamma_1 \texttt{ |- } t_1\texttt{=}t'_1 \quad \cdots \quad \Gamma_n \texttt{ |- } t_n\texttt{=}t'_n \quad \Gamma \texttt{ |- } t[t_1,\ldots,t_n]}{\Gamma_1 \cup \cdots \cup \Gamma_n \cup \Gamma \texttt{ |- } t[t'_1,\ldots,t'_n]}$$

- 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)] $t[x_1,\ldots,x_n]$ Γ |- $t[t_1,\ldots,t_n]$

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 $1 \leq i \leq n$.

1.7.5 Abstraction

```
ABS : term -> thm -> thm
```

$$\frac{\Gamma \texttt{ |- } t_1 \texttt{ = } t_2}{\Gamma \texttt{ |- (\}x.t_1\texttt{) = (\}x.t_2\texttt{)}}$$

- where x is not free in Γ.

`ABS` x Γ `|-` t_1=t_2 returns the theorem Γ `|- (\`x`.`t_1`) = (\`x`.`t_2`)`. Failure if x is not a variable, or x occurs free in any assumption in Γ.

1.7.6 Type instantiation

```
INST_TYPE : {redex : hol_type, residue : hol_type} list -> thm -> thm
```

$$\frac{\Gamma \texttt{ |- } t}{\Gamma[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n] \texttt{ |- } t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]}$$

- where $t[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]$ denotes the result of substituting (in parallel) the types σ_1, ... , σ_n for the type variables α_1, ... , α_n in the term t. Similarly, $\Gamma[\sigma_1, \ldots, \sigma_n/\alpha_1, \ldots, \alpha_n]$ denotes the result of performing the same substitution to all of the hypotheses in the set Γ.

`INST_TYPE[`α_1 `|->` σ_1`,...,`α_n `|->` σ_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 occurs if an α_i is not a type variable.

The polymorphic ML infix function `|->` is used to construct values of the record type `redex-residue`. It is defined

```
fun ((x:'a) |-> (y:'b)) = {redex = x, residue = y}
```

1.7.7 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`.

1.7.8 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.

1.8 Oracles

HOL extends the LCF tradition by allowing the use of an *oracle* mechanism, enabling arbitrary formulas to become elements of the thm type. By use of this mechanism, HOL can utilize the results of arbitrary proof procedures. In spite of such liberalness, one can still make strong assertions about the security of ML objects of type thm.

To avoid unsoundness, a *tag* is attached to any theorem coming from an oracle. This tag is propagated through every inference that the theorem participates in (much as ordinary assumptions are propagated in the inference rule MP). If it happens that falsity becomes derived, the offending oracle can be found by examining the tags component of the theorem. A theorem proved without use of any oracle will have an empty tag, and can thus be considered to have been proved solely by deductive steps in the HOL logic.

A tagged theorem can be created via

```
mk_oracle_thm : string -> term list * term -> thm
```

which directly creates the requested theorem and attaches the given tag to it. The tag is created with a call to

```
Tag.read : string -> tag
```

As well as providing principled access to the results of external reasoners, tags are used to implement some useful 'system' operations on theorems. For example, one can directly create a theorem via the function mk_thm. The tag MK_THM gets attached to each theorem created with this call. This allows users to directly create useful theorems, e.g., to use as test data for derived rules of inference. Another tag is used to implement so-called 'validity checking' for tactics.

The tags in a theorem can be viewed by setting Globals.show_tags to true.

1

```
- Globals.show_tags := true;
> val it = () : unit

- mk_thm([], Term `F`);;
> val it = [oracles: MK_THM] [axioms: ] [] |- F : thm
```

There are three elements to the left of the turnstile in the fully printed representation of a theorem: the first two[5] comprise the tags component and the third is the standard assumption list. The tag component of a theorem can be extracted by

```
Thm.tag : thm -> tag
```

and prettyprinted by

```
Tag.pp : ppstream -> tag -> unit.
```

1.9 Theories

In *LOGIC* a theory is described as 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 $\mathcal{T}$.

In the implementation of HOL, theories are structured hierarchically to represent sequences of extensions called *segments* of an initial theory called `min`. A theory segment is not really a logical concept, but rather a means of representating theories in the HOL system. Each segment records some types, constants, axioms and theorems, together with pointers to other segments called its *parents*. The theory represented by a segment is obtained by taking the union of all the types, constants, axioms and theorems in the segment, together with the types, constants, axioms and theorems in all the segments reachable by following pointers to parents. This collection of reachable segments is called the *ancestry* of the segment.

[5]Tags are also used for tracking the use of axioms in proofs.

1.9.1 ML functions for theory operations

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 importing some existing theory segments, making a number of definitions, and perhaps proving and storing some theorems in the current segment. Then the current segment (named $name$ say) is exported. The concrete result will be an ML module $name$`Theory` whose contents is the current theory segment created during the session and whose ancestry represents the desired logical theory $\mathcal{T}$. Subsequent work sessions can access the definitions and theorems of $\mathcal{T}$ by importing $name$`Theory`; this avoids having to load the tools and replay the proofs that created $name$`Theory` in the first place.

The naming of data in theories is based on the names given to segments. Specifically an axiom, definition, specification or theorem is accessed by an ML long identifier thy`Theory`.$name$, 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. Thus each theory segment provides a separate namespace of ML bindings of HOL items.

Various additional pieces of information are stored in a theory segment, including the parsing status of the constants (e.g. whether they are infixes or binders).

Determining the context 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 segment is returned by the ML function:

```
current_theory : unit -> string
```

When an interactive HOL session begins, some theories will already be in the logical context. The exact set of theories in context will vary. If the executable used is `hol.bare`, then only `min` and `bool` will be loaded. When the `hol` executable is used, a richer context is loaded.

The exact set of theories loaded can be determined with the `ancestry` command.

```
ancestry : string -> string list
```

This function provides a general mechanism for examining the structure of the theory hierarchy. The argument is the name of a theory (or `"-"` as an abbreviation for the current theory), to which `ancestry` will respondwith a list of the argument's ancestors in the theory hierarchy.

```
- ancestry "-";
> val it =
    ["num", "prim_rec", "normalForms", "relation", "pair",
      "arithmetic", "while", "numeral", "label", "combin", "sum", "min",
      "bool", "marker", "one", "option", "ind_type", "list"] : string list
```

Creating a theory segment New theory segment are created by a call to `new_theory`.

```
new_theory : string -> unit
```

This allocates a new 'area' where subsequent theory operations take effect. If the current theory (thy_1 say) at the time of a call to `new_theory` thy_2 is non-empty, i.e., has had an axiom, definition, or theorem stored in it, then thy_1 is exported before thy_2 is allocated. Furthermore, thy_2 will obtain thy_1 as a parent. If `new_theory` thy is called when the current theory segment is already named thy, then that is interpreted as a request merely to clear the current theory segment (nothing will be exported).

A call to `new_theory "`$name$`"` fails if:

- $name$ is not an alphanumeric starting with a letter.
- there is a theory already named $name$ in the ancestry of the current segment.
- if it is necessary to export the current segment before creating the new theory and the export attempt fails.

On startup, the current theory segment of HOL is named `scratch`, which is an empty theory, having a useful collection of theories in its ancestry. Typically, a user would begin by loading whatever extra logical context is required for the work at hand.

The current theory segment acts as a kind of scratchpad. Elements stored in the current segment may be overwritten by subsequent additions, or deleted outright. Any theory elements that were built from overwritten or deleted elements would then be held to be *out-of-date*, and would not be included in the theory when it is finally exported. Out-of-date constants and types are detected by the HOL printer, which will print them surrounded by odd-looking syntax to alert the user.

In contrast to the current segment, (proper) ancestor segments may not be altered.

Loading prebuilt theories Since HOL theories are represented by ML modules, one imports an existing theory segment by simply importing the corresponding module.

```
load : string -> unit
```

Executing `load` $name$`Theory` imports the first file named $name$`Theory.uo` found along the `loadPath` into the session. Any unloaded ancestors of $name$ will be loaded before loading of $name$`Theory` continues. Note that `load` can not be used in ML files that are to be compiled; it can only be used in the interactive system.

Adding to the current theory The following ML functions add types and terms to the current theory segment. In typical usage, these functions will not be needed since higher-level definition facilities will invoke these as necessary. However, these functions can be useful for those writing proof tools and derived definition principles.

```
new_type : int -> string -> unit
```

Executing `new_type` n `"`*op*`"` makes *op* a new n-ary type operator in the current theory. If *op* is not an allowed name for a type, a warning will be issued.

```
new_constant : (string * type) -> unit
```

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. If c is not an allowed name for a constant, a warning will be issued.

```
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) the type of t is not `bool`.

(ii) t contains out-of-date constants or types, i.e., constants or types that have been re-declared after t was built.

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.

Exporting a theory Once a theory segment has been constructed, it can be written out to a file, which, after compilation, can be imported into future sessions.

```
export_theory : unit -> unit
```

When `export_theory` is called, all out-of-date entities are removed from the current segment. Also, the parenthood of the theory is computed. The current theory segment is written to file $name$`Theory.sml` in the current working directory. The file $name$`Theory.sig`, which documents the contents of $name$, is also written to the current working directory. Notice that the exported theory is not compiled by HOL. That is left to an external tool, `Holmake` (see section 6.3), which maintains dependencies among collections of HOL theory segments.

1.9.2 ML functions 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.

The current theory can be extended by adding new parents, types, constants, axioms and definitions. Theories that are in the ancestry of the current theory cannot be extended in this way; they can be thought of as *frozen*.

There are various functions for loading the contents of theory files:

```
parents     : string -> string list
types       : string -> (int * string) list
constants   : string -> term list
```

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. Individual axioms, definitions and theorems can be read from the current theory using the following ML functions:

```
axiom      : string -> thm
definition : string -> thm
theorem    : string -> thm
```

The first argument is the user supplied name of the axiom, definition or theorem in the current theory. Further, a list of all of a theory's axioms, definitions and theorems can be retreived with the ML functions:

```
axioms      : string -> (string * thm) list
definitions : string -> (string * thm) list
theorems    : string -> (string * thm) list
```

The contents of the current theory can be printed in a readable format using the function `print_theory`.

1.9.3 Functions for creating definitional extensions

There are three kinds of definitional extensions: constant definitions, constant specifications and type definitions.

1.9.3.1 Constant definitions

In *LOGIC* 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 *LOGIC* 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("`$name$`", ```$c\ v_1\ \cdots\ v_n\ =\ t$``)`, declares the sequent `({},`$c = \lambda v_1\ \cdots\ v_n.\ t$`)` to be a constant definition of the current theory. The name associated with the definition in this theory is $name$. Failure occurs if:

(i) 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);

(ii) there is a type variable in v_1, ..., v_n or t that does not occur in the type of c.

1.9.3.2 Constant specifications

In *LOGIC* a constant specification for a theory $\mathcal{T}$ is defined to be a pair:

$$\langle (c_1, \ldots, c_n),\ \lambda x_{1\sigma_1} \cdots x_{n\sigma_n}.\ t_{\mathit{bool}} \rangle$$

such that:

(i) c_1, ..., c_n are distinct names.

(ii) $\lambda x_{1\sigma_1} \cdots x_{n\sigma_n}.\ t_{\mathit{bool}} \in \mathsf{Terms}_{\mathcal{T}}$.

(iii) $tyvars(\lambda x_{1_{\sigma_1}} \cdots x_{n_{\sigma_n}}.\, t_{\mathit{bool}}) \subseteq tyvars(\sigma_i)$ 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 list * thm -> thm
```

Evaluating:

```
new_specification("name", ["c1", ..., "cn"],
                  |- ?x1 ... xn. t[x1, ..., xn])
```

simultaneously introduces new constants named c_1, ..., c_n satisfying the property:

```
|- t[c1, ... ,cn]
```

This theorem is stored, with name $name$, as a definition in the current theory segment. A call to `new_specification` fails if:

(i) the theorem argument has a non-empty assumption list;

(ii) there are free variables in the theorem argument;

(iii) c_1, ..., c_n are not distinct variables;

(iv) the type of some c_i does not contain all the type variables which occur in the term `\x1 ... xn. t[x1, ..., xn]`.

1.9.3.3 Type definitions

In *LOGIC* it is explained that defining a new type $(\alpha_1, \ldots, \alpha_n)\mathit{op}$ in a theory $\mathcal{T}$ consists of introducing *op* as a new n-ary type operator and

$$\vdash\ \exists f_{(\alpha_1,\ldots,\alpha_n)\mathit{op}\to\sigma}.\ \mathsf{Type\_Definition}\ p\ f$$

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

$$\langle \sigma,\ (\alpha_1, \ldots, \alpha_n)\mathit{op},\ p_{\sigma\to\mathit{bool}} \rangle$$

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 $\sigma \to$*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 * thm) -> thm
```

If t is a term of type σ`->bool` containing n distinct type variables, then evaluating:

`new_type_definition("`*op*`", |- ?`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`.. The constant `TYPE_DEFINITION` is defined in the theory `bool` by:

```
|- TYPE_DEFINITION (P:'a->bool) (rep:'b->'a) =
    (!x' x''. (rep x' = rep x'') ==> (x' = x'')) /\
    (!x. P x = (?x'. x = rep x'))
```

Executing `new_type_definition("`*op*`", |- ?`x`. `t x`)` fails if:

(i) t does not have a type of the form σ`->bool`.

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
   : {name:string, ABS:string, REP:string, tyax:thm} -> thm
```

This function takes a record `{ABS, REP, name, tyax}` The `tyax` argument must be a definitional axiom of the form returned by `new_type_definition`. The `name` 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 `ABS` and `REP` 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="name", ABS="abs", REP="rep",
      tyax = |- ?rep:newty->ty. TYPE_DEFINITION P rep}
```

automatically defines two new constants abs`:ty->newty` and rep`:ty->newty` such that:

```
|- (!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) th is not a theorem of the form returned by `new_type_definition`.

Properties of type bijections The following ML functions are provided for proving that the bijections introduced by `define_new_type_bijections` 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 in the current theory.

Chapter 2

Derived Inference Rules

The notion of *proof* is defined abstractly in the manual *LOGIC*: 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 *LOGIC*: 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 1.7, 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 four axioms, to form the basic logic. 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.

2.1 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

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.

2

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

- val th1 = ASSUME ``t1 ==> t2``;
> val th1 = [t1 ==> t2] |- t1 ==> t2 : thm

- val th2 = ASSUME ``t1:bool``
> val th2 = [t1] |- t1 : thm

- MP th1 th2;
> val it = [t1 ==> t2, t1] |- t2 : thm
```

More briefly, one could evaluate the following, and 'count' the invocations of functions representing primitive inferences.

3

```
#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 applicationof 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

$$\frac{\Gamma \text{ |- } t_1 \text{ ==> } t_2}{\Gamma \cup \{t_1\} \text{ |- } t_2}$$

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

```
- val UNDISCH th = MP th (ASSUME(fst(dest_imp(concl th))));;
> val UNDISCH = fn : thm -> thm
```

4

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.) Assume that the printing of theorems has been adjusted as above and th is bound as shown below:

```
#th;;
|- (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
```

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 2.3, 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*.

2.2 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 2.3.13 and 2.3.11) for instantiating terms. The definition of REWRITE_RULE in ML also relies on a large number of general and HOL-oriented ML functions.

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[1]; 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.

2

```
#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.

[1]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.

2.3 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 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 3.2.1), 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 2.3.9, 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 familyof theorems. No single theorem can be expressed in the HOL logic that is equivalent to `BETA_CONV`. (`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 *LOGIC* 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.

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` (standing for 'conversion') is used for the ML type `term -> thm`.

2.3.1 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]

2.3.2 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]

2.3.3 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]

2.3.4 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]

2.3.5 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]

2.3.6 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]

2.3.7 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]

2.3.8 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]

2.3.9 $\top$-Introduction

```
TRUTH
```

$$\vdash \top$$

1. $\vdash \top = ((\lambda x.\ x) = (\lambda x.\ x))$ [Definition of $\top$]
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]

2.3.10 Equality-with-$\top$ 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]

2.3.11 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 $\forall$]
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]

2.3.12 Equality-with-⊤ 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]

2.3.13 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]

2.3.14 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.[2]

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]

2.3.15 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]

[2] 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.

2.3.16 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 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 (x' is a fresh)]
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]

2.3.17 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]

2.3.18 ε-elimination

```
SELECT_ELIM : thm -> (term # thm) -> thm
```

$$\frac{\Gamma_1 \vdash t_1(\varepsilon\, t_1) \qquad\qquad \Gamma_2,\ t_1\ v \vdash t}{\Gamma_1 \cup \Gamma_2 \vdash t}$$

- 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]

2.3.19 $\exists$-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]

2.3.20 ∃-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]

2.3.21 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]

2.3.22 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]

2.3.23 $\wedge$-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]

2.3.24 $\wedge$-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]

2.3.25 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]

2.3.26 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]

2.3.27 $\vee$-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]

2.3.28 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 3

Core Theories

The HOL system provides a collection of theories on which to base verification tools or further theory development. In the rest of this section, these theories are briefly described. The sections that follow provide an overview of the contents of each theory. For a complete list of all the axioms, definitions and theorems in HOL, see the online resources distributed with the system. In particular, the HTML file `help/HOLindex.html` is a good place to start browsing the available theories. For a graphical picture of the theory hierarchy, see `help/theorygraph/theories.html`.

3.1 The Theory `min`

The starting theory of HOL is the theory `min`. In this theory, the type constant `bool` of booleans, the binary type operator (α, β)`fun` of functions, and the type constant `ind` of individuals are declared. Building on these types, three primitive constants are declared: equality, implication, and a choice operator:

Equality Equality (`= : 'a -> 'a -> bool`) is an infix operator.

Implication Implication (`==> : bool -> bool -> bool`) is the *material implication* and is an infix operator that is right-associative, i.e., `x ==> y ==> z` parses to the same term as `x ==> (y ==> z)`.

Choice Equality and implication are standard 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 [2], Leisenring's book on Hilbert's ε-symbol [6], or Andrews' textbook on type theory [1].

No theorems or axioms are placed in theory `min`. The primitive rules of inference of HOL depend on the presence of `min`.

3.2 Basic Theories

The most basic theories in HOL provide support for a standard collection of types. The theory `bool` defines the basis of the HOL logic, including the boolean operations and quantifiers. On this platform, quite a bit of theorem-proving infrastructure can already be built. Further basic types are developed in the theory of pairs (`prod`), disjoint sums (`sum`), the one-element type (`one`), and the (`option`) type.

3.2.1 The theory `bool`

At start-up, the initial theory for users of the HOL system is called `bool`, which is constructed when the HOL system is built. The theory `bool` is an extension of the combination of the "conceptual" theories `LOG` and `INIT`, described in *LOGIC*. Thus it contains the four axioms for higher order logic. These axioms, together with the rules of inference described in Section 1.7, constitute the core of the HOL logic. Because of the way the HOL system evolved from LCF[1], the particular axiomatization of higher order logic it uses differs from the classical axiomatization due to Church [2]. 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.

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 build on earlier ones.

```
T_DEF              |- T  = ((\x:bool. x) = (\x. x))

FORALL_DEF         |- !  = \P:'a->bool. P = (\x. T)

EXISTS_DEF         |- ?  = \P:'a->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`; the first three are the following:

[1] 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.

```
BOOL_CASES_AX   |- !t. (t = T) \/ (t = F)

ETA_AX          |- !t. (\x. t x) = t

SELECT_AX       |- !P:'a->bool x. P x ==> P($@ P)
```

The fourth and last axiom of the HOL logic is the Axiom of Infinity. Its statement is phrased in terms of the function properties `ONE_ONE` and `ONTO`. 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)
```

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.

The three other axioms of the theory `bool`, the rules of inference in Section 1.7 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.

Further definitions The theory `bool` also supplies the definitions of a number of useful constants.

```
LET_DEF  |- LET  = \f x. f x
COND_DEF |- COND = \t t1 t2. @x. ((t=T)==>(x=t1)) /\ ((t=F)==>(x=t2))
IN_DEF   |- IN   = \x (f:'a -> bool). f x
```

The constant `LET` is used in representing terms containing local variable bindings (i.e. `let`-terms). For example, the concrete syntax `let v = M in N` is translated by the parser to the term `LET (\v.N) M`. For the full description of how `let` expressions are translated, see Section 3.2.3.

The constant `COND` is used to represent conditional expressions. The concrete syntax `if` t_1 `then` t_2 `else` t_3 abbreviates the application `COND` t_1 t_2 t_3.

The constant `IN` (written as an infix) is the basis of the modelling of sets by their characteristic functions. The term x `IN` P can be read as "x is an element of the set P", or (more in line with its definition) as "the predicate P is true of x".

Finally, the polymorphic constant `ARB` : α denotes a fixed but arbitrary element. `ARB` is occasionally useful when attempting to deal with the issue of partiality.

3.2.1.1 Restricted quantifiers

The theory `bool` also defines constants that implement *restricted quantification*. This provides a means of simulating subtypes and dependent types with predicates. The most heavily used are restrictions of the existential and universal quantifiers:

```
RES_FORALL_DEF |- RES_FORALL = \P m. !x. x IN P ==> m x

RES_EXISTS_DEF |- RES_EXISTS = \P m. ?x. x IN P /\ m x

RES_ABSTRACT_DEF |- (!P m x. x IN P ==> (RES_ABSTRACT P m x = m x) /\
                    (!P m1 m2.
                        (!x. x IN P ==> (m1 x = m2 x)) ==>
                         (RES_ABSTRACT P m1 = RES_ABSTRACT P m2)
```

The definition of `RES_ABSTRACT` is a characterising formula, rather than a direct equation. There are two important properties

- if y is an element of P then $(\backslash x :: P.\ M)y = M[y/x]$
- If two restricted abstractions agree on all values over their (common) restricting set, then they are equal.

For completeness, restricted versions of unique existence and indefinite description are provided, although hardly used.

```
RES_EXISTS_UNIQUE_DEF
 |- RES_EXISTS_UNIQUE = \P m. (?x :: P. m x) /\
                              (!x y :: P. m x /\ m y ==> (x = y))

RES_SELECT_DEF
  |- RES_SELECT = \P m. @x. x IN P /\ m x
```

The definition of `RES_EXISTS_UNIQUE` uses the restricted quantification syntax with the `::` symbol, referring to the earlier definitions `RES_EXISTS` and `RES_FORALL`. The `::` syntax is used with restricted quantifiers to allow arbitrary predicates to restrict binding variables. The HOL parser allows restricted quantification of all of a sequence of binding variables by putting the restriction at the end of the sequence, thus with a universal quantification:

$$\forall x\, y\, z :: P\,.\, Q(x, y, z)$$

Here the predicate P restricts all of x, y and z.

3.2.1.2 Derived 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 1.9), as it is when `arithmeticTheory` has been loaded, 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 token (such as `if`, + or `let`) 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$`)`	`mk_select(`x`,`t`)`
Conditional	`if` t `then` t_1 `else` t_2	`COND` t t_1 t_2	`mk_cond(`t`,`t_1`,`t_2`)`
`let`-expression	`let` x`=`t_1 `in` t_2	`LET(\`$x.t_2$`)`t_1	`mk_let(mk_abs(`x`,`t_2`),`t_1`)`

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, destructors, and indicators for pairs and lists, namely `mk_pair`, `mk_cons` and `mk_list` (see *REFERENCE*). The constants `COND` and `LET` are explained in Section 3.2.1. 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 `$`c`(\`$x\,.\,t$`)` (i.e. the application of the constant c to the representation of the abstraction $\lambda x.\ t$).

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 for conjunctions, disjunctions, and implications respectively. The corresponding destructor functions are called `strip_comb`, etc.,

3.2.1.3 Theorems

A large number of theorems involving the logical constants are pre-proved in the theory `bool`. The following theorems illustrate how higher order logic allows concise expression of theorems supporting quantifier movement.

```
LEFT_AND_FORALL_THM  |- !P Q. (!x. P x) /\ Q = !x. P x /\ Q
RIGHT_AND_FORALL_THM |- !P Q. P /\ (!x. Q x) = !x. P /\ Q x

LEFT_EXISTS_AND_THM  |- !P Q. (?x. P x /\ Q) = (?x. P x) /\ Q
RIGHT_EXISTS_AND_THM |- !P Q. (?x. P /\ Q x) = P /\ ?x. Q x

LEFT_FORALL_IMP_THM  |- !P Q. (!x. P x ==> Q) = (?x. P x) ==> Q
RIGHT_FORALL_IMP_THM |- !P Q. (!x. P ==> Q x) = P ==> !x. Q x

LEFT_EXISTS_IMP_THM  |- !P Q. (?x. P x ==> Q) = (!x. P x) ==> Q
RIGHT_EXISTS_IMP_THM |- !P Q. (?x. P ==> Q x) = P ==> ?x. Q x

LEFT_FORALL_OR_THM   |- !Q P. (!x. P x \/ Q) = (!x. P x) \/ Q
RIGHT_FORALL_OR_THM  |- !P Q. (!x. P \/ Q x) = P \/ !x. Q x

LEFT_OR_EXISTS_THM   |- !P Q. (?x. P x) \/ Q = ?x. P x \/ Q
RIGHT_OR_EXISTS_THM  |- !P Q. P \/ (?x. Q x) = ?x. P \/ Q x

EXISTS_OR_THM        |- !P Q. (?x. P x \/ Q x) = (?x. P x) \/ ?x. Q x
FORALL_AND_THM       |- !P Q. (!x. P x /\ Q x) = (!x. P x) /\ !x. Q x

NOT_EXISTS_THM       |- !P. ~(?x. P x) = !x. ~P x
NOT_FORALL_THM       |- !P. ~(!x. P x) = ?x. ~P x

SKOLEM_THM           |- !P. (!x. ?y. P x y) = ?f. !x. P x (f x)
```

Also, a theorem justifying Skolemization (`SKOLEM_THM`) is proved. Many other theorems may be found in `bool` theory.

3.2.2 Combinators

The theory `combin` contains the definitions of function composition (infixed `o`), a reversed function application operator, function override (infixed =+), and the combinators `S`, `K`, `I`, `W`, and `C`,

```
    o_DEF |- f o g = (\x. f(g x))
  APP_DEF |- x :> f = f x
UPDATE_DEF |- (k =+ v) = (\f c. if k = c then v else f c)
    K_DEF |- K = (\x y. x)
    S_DEF |- S = (\f g x. f x(g x))
    I_DEF |- I = S K K
    W_DEF |- W = (\f x. f x x)
    C_DEF |- C = (\f x y. f y x)
```

The following elementary properties are 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

UPDATE_EQ
        |- !f a b c. (a =+ c) ((a =+ b) f) = (a =+ c) f
UPDATE_COMMUTES
        |- !f a b c d. a <> b ==>
                       ((a =+ c) ((b =+ d) f) = (b =+ d) ((a =+ c) f))

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
W_THM   |- !f x. W f x = f x x
C_THM   |- !f x y. C f x y = f y x
```

There are no theorems about `:>`; its use is as a convenient syntax for function applications. For example, chains of updates can lose some parentheses if written

```
f :> (k1 += v1) :> (k2 += v2) :> (k3 += v3)
```

This presentation also makes the order in which functions are applied read from left-to-right.

Having the symbols `o`, `S`, `K`, `I`, `W`, and `C` 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).[2] Variables with these names can be used in the current system if `o`, `S`, `K`, `I`, `W`, and `C` are first hidden (see Section 5.1.2.9). In fact, this happens so often with the constant `C` that it is "hidden" by default. While hidden, it must be written in fully-qualified form, as `combin$C`.

[2]Constants declared in new theories can freely re-use these names, with ambiguous inputs resolved by type inference.

3.2.3 Pairs

The Cartesian product type operator `prod` is defined in the theory `pair`. 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 type parser converts type expressions of the form $:\sigma_1\#\sigma_2$ into (σ_1,σ_2)`prod`, and the printer inverts this transformation. Pairs are constructed with an infixed comma symbol

```
$, : 'a -> 'b -> 'a # 'b
```

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 $\sigma_1\#\sigma_2$. Usually, pairs are written within brackets: (t_1,t_2). The comma symbol associates to the right, so that $(t_1,t_2,\ldots,t_n)$ means $(t_1,(t_2,\ldots,t_n))$.

Defining the product type The type of Cartesian products is defined by representing a pair (t_1,t_2) by the function

```
\a b. (a=t1) /\ (b=t2)
```

The representing type of $\sigma_1\#\sigma_2$ is thus σ_1`->`σ_2`->bool`. It is easy to prove the following theorem.[3]

```
|- ?p:'a->'b->bool. (\p. ?x y. p = \a b. (a = x) /\ (b = y)) p
```

The type operator `prod` is defined by invoking `new_type_definition` with this theorem which results in the definitional axiom `prod_TY_DEF` shown below being asserted in the theory `pair`.

```
prod_TY_DEF
  |- ?rep. TYPE_DEFINITION (\p. ?x y. p = (\a b. (a = x) /\ (b = y))) rep
```

Next, the representation and abstraction functions `REP_prod` and `ABS_prod` for the new type are introduced, along with the following characterizing theorem, by use of the function `define_new_type_bijections`.

```
|- (!a. ABS_prod (REP_prod a) = a) /\
   (!r. (\p. ?x y. p = (\a b. (a=x) /\ (b=y)) r = (REP_prod(ABS_prod r) = r)
```

[3] This theorem has an un-reduced β-redex in order to meet the interface required by the type definition principle.

Pairs and projections The infix constructor ',' is then defined to be an application of the abstraction function. Subsequently, two crucial theorems are proved: PAIR_EQ asserts that equal pairs have equal components and ABS_PAIR_THM shows that every term having a product type can be decomposed into a pair of terms.

```
COMMA_DEF    |- !x y. $, x y = ABS_prod (\a b. (a = x) /\(b = y))

PAIR_EQ      |- ((x,y) = (a,b)) = (x=a) /\ (y=b)

ABS_PAIR_THM |- !x. ?q r. x = (q,r)
```

By Skolemizing ABS_PAIR_THM and making constant specifications for FST and SND, the following theorems are proved.

```
PAIR     |- !x. (FST x,SND x) = x
FST      |- !x y. FST(x,y) = x
SND      |- !x y. SND(x,y) = y
```

Pairs and functions In HOL, a function of type $\alpha\#\beta \to \gamma$ always has a counterpart of type $\alpha \to \beta \to \gamma$, and *vice versa*. This conversion is accomplished by the functions CURRY and UNCURRY. These functions are inverses.

```
CURRY_DEF    |- !f x y. CURRY f x y = f (x,y)
UNCURRY_DEF  |- !f x y. UNCURRY f (x,y) = f x y

CURRY_UNCURRY_THM |- !f. CURRY (UNCURRY f) = f
UNCURRY_CURRY_THM |- !f. UNCURRY (CURRY f) = f
```

Mapping functions over a pair Functions $f : \alpha \to \gamma_1$ and $g : \beta \to \gamma_2$ can be applied component-wise (##, infix) over a pair of type $\alpha\#\beta$ to obtain a pair of type $\gamma_1\#\gamma_2$.

```
PAIR_MAP_THM  |- !f g x y. (f ## g) (x,y) = (f x,g y)
```

Binders and pairs When doing proofs, statements involving tuples may take the form of a binding (quantification or λ-abstraction) of a variable with a product type. It may be convenient in subsequent reasoning steps to replace the variables with tuples of variables. The following theorems support this.

```
FORALL_PROD  |- (!p. P p) = !p_1 p_2. P (p_1,p_2)
EXISTS_PROD  |- (?p. P p) = ?p_1 p_2. P (p_1,p_2)
LAMBDA_PROD  |- !P. (\p. P p) = \(p1,p2). P (p1,p2)
```

The theorem LAMBDA_PROD involves a *paired abstraction*, discussed in Section 3.2.3.1.

Wellfounded relations on pairs Wellfoundedness, defined in Section 3.3.1.4, is a useful notion, especially for proving termination of recursive functions. For pairs, the lexicographic combination of relations (LEX, infix) may be defined by using paired abstractions. Then the theorem that lexicographic combination of wellfounded relations delivers a wellfounded relation is easy to prove.

```
LEX_DEF =
   |- !R1 R2. R1 LEX R2 = (\(s,t) (u,v). R1 s u \/ (s = u) /\ R2 t v)
WF_LEX
   |- !R Q. WF R /\ WF Q ==> WF (R LEX Q)
```

3.2.3.1 Paired abstractions

It is notationally convenient to include pairing in the lambda notation, as a simple pattern-matching mechanism. The quotation parser will convert the term `\(`x_1`,`x_2`).`t to `UNCURRY(\`x_1 x_2`.`t`)`. 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 $\quad\rightsquigarrow\quad$ `UNCURRY(\`v_1`.\`v_2`.`t`)`

until no more variable structures remain. For example:

`\(`x`,`y`).`t	$\rightsquigarrow$ `UNCURRY(\`$x\,y$`.`t`)`
`\(`x_1`,`x_2`,...,`x_n`).`t	$\rightsquigarrow$ `UNCURRY(\`x_1`.\(`x_2`,...,`x_n`).`t`)`
`\((`x_1`,...,`x_n`),`y_1`,...,`y_m`).`t	$\rightsquigarrow$ `UNCURRY(\(`x_1`,...,`x_n`).\(`y_1`,...,`y_m`).`t`)`

As a result of this parser translation, a variable structure, such as `(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, antiquoting a pair into the bound variable position of a lambda abstraction fails:

1
```
- ‘‘\(x,y).x+y‘‘;
> val it = ‘\(x,y). x + y‘ : term

- val p = Term ‘(x:num,y:num)‘;
> val p = ‘(x,y)‘ : term

- Lib.try Term ‘\^p.x+y‘;

Exception raised at Term.dest_var:
not a var
! Uncaught exception:
```

If b is a binder, then $b(x_1,x_2).t$ is parsed as $b(\backslash(x_1,x_2).t)$, and hence transformed as above. For example, `!(x,y). x > y` parses to `$!(UNCURRY(\x.\y. x > y))`.

3.2.3.2 `let`-terms

The quotation parser accepts `let`-terms 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)
```

`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` $f\,v_1 \ldots v_n$ `=` t_1 `in` t_2	$\rightsquigarrow$	`LET(\`$f.t_2$`)(\`$v_1 \ldots v_n.t_1$`)`
`let` $(v_1,\ldots,v_n)$ `=` t_1 `in` t_2	$\rightsquigarrow$	`LET(\`$(v_1,\ldots,v_n).t_2$`)`t_1
`let` v_1`=`t_1 `and` ... `and` v_n`=`t_n `in` t	$\rightsquigarrow$	`LET(`$\ldots$`(LET(LET(\`$v_1 \ldots v_n.t$`)`t_1`)`t_2`)`$\ldots$`)`t_n

The underlying structure of the term can be seen by applying destructor operations. For example:

2

```
- Term ‘let x = 1 and y = 2 in x+y‘;
> val it = ‘let x = 1 and y = 2 in x + y‘ : term

- dest_comb it;
> val it = (‘LET (LET (\x y. x + y) 1)‘, ‘2‘) : term * term

- Term ‘let (x,y) = (1,2) in x+y‘;
> val it = ‘let (x,y) = (1,2) in x + y‘ : Term.term

- dest_comb it;
> val it = (‘LET (\(x,y). x + y)‘, ‘(1,2)‘) : Term.term * Term.term
```

Readers are encouraged to convince themselves that the translations of `let`-terms represent the intuitive meaning suggested by the surface syntax.

3.2.4 Disjoint sums

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.[4] The HOL parser converts ``:$\sigma_1$+$\sigma_2$`` into ``:$(\sigma_1,\sigma_2)$sum``, and the printer inverts this.

The standard operations on sums are:

```
INL  : 'a -> 'a + 'b
INR  : 'b -> 'a + 'b
ISL  : 'a + 'b -> bool
ISR  : 'a + 'b -> bool
OUTL : 'a + 'b -> 'a
OUTR : 'a + 'b -> 'b
```

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 theorem is proved in the theory sum. It provides a complete and abstract characterization of the disjoint sum type, and is used to justify the definition of functions over sums.

```
sum_Axiom  |- !f g. ?! h. (!x. h(INL x) = f x) /\ (!x. h(INR x) = g x)
```

Also provided 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
```

The sum theory also provides the following theorems relating the projection functions and the discriminators.

```
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)
```

[4] The definition of disjoint unions in the HOL system is due to Tom Melham. The technical details of this definition can be found in [8].

3.2.5 The one-element type

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:'a->one) (g:'a -> one). f = g
one         |- !(v:one). v = one
one_Axiom   |- !(e:'a). ?!(fn:one->'a). fn one = e
```

These three theorems are equivalent characterizations of the type with only one value. The theory `one` is typically used in constructing more elaborate types. The one value of the type `one`, can also be written as `()` by analogy with the unit value in ML. This is also the default way in which this value is printed by the system pretty-printer.

3.2.5.1 The itself type

The unary `itself` type operator provides a family of singleton types akin to `one`. Thus, for every type α, α `itself` is a type containing just one value. This value's name is `the_value`, but the parser and pretty-printer are set up so that for the type α `itself`, `the_value` can be written as `(:`α`)` (the syntax includes the parentheses). For example, `(:num)` is the single value inhabiting the type `num itself`.

The point of the itself type is that if one defines a function with α `itself` as the domain, the function picks out just one value in its range, and so one can think of the function as being one from the type to a value for the whole type.

For example, one could define

```
finite_univ (:'a) = FINITE (UNIV :'a set)
```

It would then be straightforward to prove the following theorems

```
⊢ finite_univ(:bool)
⊢ ¬finite_univ(:num)
⊢ finite_univ(:'a) ∧ finite_univ(:'b) ⇒ finite_univ(:'a # 'b)
```

The itself type is used in the Finite Cartesian Product construction that underlies the fixed-width word type (see Section 3.3.8 below).

3.2.6 The option type

The theory `option` defines a type operator `option` that 'lifts' its argument type, creating a type with all of the values of the argument and one other, specially distinguished value. The constructors of this type are

```
NONE : 'a option
SOME : 'a -> 'a option
```

Options can be used to model partial functions. If a function of type $\alpha \to \beta$ does not have useful β values for all α inputs, then this distinction can be marked by making the range of the function β option, and mapping the undefined α values to NONE.

An inductive type, options have a recursion theorem supporting the definition of primitive recursive functions over option values.

```
option_Axiom
  |- !e f.
      ?h:'a option -> 'b.
        (!x. h (SOME x) = f x) /\
        (h NONE = e)
```

The option theory also defines a case constant that allows one to inspect option values in a "pattern-matching" style.

```
case e of
  NONE => u
| SOME x => f x
```

The constant underlying this syntactic sugar is option_case with definition

```
option_case_def |- (option_case u f NONE = u) /\
                   (option_case u f (SOME x) = f x)
```

Another useful function maps a function over an option:

```
OPTION_MAP_DEF  |- (OPTION_MAP f NONE = NONE) /\
                   (OPTION_MAP f (SOME x) = SOME (f x))
```

Finally, the THE function takes a SOME value to that constructor's argument, and is unspecified on NONE:

```
THE_DEF   |- THE (SOME x) = x
```

3.3 Numbers

The natural numbers, integers, and real numbers are provided in a series of theories. Also available are theories of n-bit words (numbers modulo 2^n), floating point and fixed point numbers.

3.3.1 Natural numbers

The natural numbers are developed in a series of theories: num, prim_rec, arithmetic, and numeral. In num, the type of numbers is defined from the Axiom of Infinity, and Peano's axioms are derived. In prim_rec the Primitive Recursion theorem is proved. Based on that, a large theory treating the standard arithmetic operations is developed in arithmetic. Lastly, a theory of numerals is developed.

3.3.1.1 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.

3.3.1.2 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.[5] 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, named num_Axiom in HOL, follows from Peano's axioms.

```
num_Axiom  |- !x f. ?fn. (fn 0 = x) /\ (!n. fn(SUC n) = f n (fn 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.

[5] 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.

The less-than relation The less-than relation '<' is most naturally defined by primitive recursion. However, in our development 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[6] and contains `m` but not `n`.

3.3.1.3 Mechanizing 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

```
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 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 1.9.3.2) 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:

[6] A set of numbers is *downward closed* if whenever it contains the successor of a number, it also contains the number.

```
|- 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 a function which automatically proves the existence of primitive recursive functions and then makes a constant specification to introduce the constant that denotes such a function:

```
new_recursive_definition :
   {def : term, name : string, rec_axiom : thm} -> thm
```

In fact, `new_recursive_definition` handles primitive recursive definitions over a range of types, not just the natural numbers. For details, see the *REFERENCE* documentation.

More conveniently still, the `Define` function (see Section 5.3.1) supports primitive recursion, along with other styles of recursion, and does not require the user to quote the primitive recursion axiom. It may, however, require termination proofs to be performed; fortunately, these need not be done for primitive recursions.

3.3.1.4 Dependent choice and wellfoundedness

The primitive recursion theorem is useful beyond its main purpose of justifying recursive definitions. For example, the theory `prim_rec` proves the Axiom of Dependent Choice (DC).

```
DC  |- !P R a.
         P a /\ (!x. P x ==> ?y. P y /\ R x y)
          ==>
        ?f. (f 0 = a) /\ !n. P (f n) /\ R (f n) (f (SUC n))
```

The proof uses `SELECT_AX`. The theorem DC is useful when one wishes to build a function having a certain property from a relation. For example, one way to define the wellfoundedness of a relation R is to say that it has no infinite decreasing R chains.

```
wellfounded_def
  |- wellfounded (R:'a->'a->bool) = ~?f. !n. R (f (SUC n)) (f n)

WF_IFF_WELLFOUNDED
  |- !R. WF R = wellfounded R
```

By use of DC, this statement can be proved to be equal to the notion of wellfoundedness WF (namely, that every set has an R-minimal element) defined in the theory relation.

Theorems asserting the wellfoundedness of the predecessor relation and the less-than relation, as well as the wellfoundedness of measure functions are also proved in prim_rec.

```
WF_PRED     |- WF (\x y. y = SUC x)
WF_LESS     |- WF $<

measure_def |- measure = inv_image $<
measure_thm |- !f x y. measure f x y = f x < f y
WF_measure  |- !m. WF (measure m)
```

3.3.2 Arithmetic

The HOL theory arithmetic contains primitive recursive definitions of the 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 = if m < n then 0 else 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))
```

Note that EXP is an infix. The infix notation ** may be used in place of EXP. Thus (x EXP y) means x^y, and so does (x ** y).

Comparison operators A full set of comparison operators is defined in terms of <.

```
GREATER_DEF    |- !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 and modulus A constant specification is used to introduce division (DIV, infix) and modulus (MOD, infix) operators, together with their characterizing property.

```
DIVISION
  |- !n. 0 < n ==> !k. (k = ((k DIV n) * n) + (k MOD n)) /\ (k MOD n) < n
```

Even and odd The properties of a number being even or odd are defined recursively.

```
EVEN |- (EVEN 0 = T) /\ !n. EVEN (SUC n) = ~EVEN n

ODD  |- (ODD 0 = F) /\ !n. ODD (SUC n) = ~ODD n
```

Maximum and minimum The minimum and maximum of two numbers are defined in the usual way.

```
MAX_DEF |- !m n. MAX m n = (if m < n then n else m)
MIN_DEF |- !m n. MIN m n = (if m < n then m else n)
```

Factorial The factorial of a number is a primitive recursive definition.

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

Function iteration The iterated application $f^n x$ of a function $f : \alpha \to \alpha$ is defined by primitive recursion. The definition (FUNPOW) is tail-recursive, which can be awkward to reason about. An alternative characterization (FUNPOW_SUC) may be easier to apply when doing proofs.

```
FUNPOW
  |- (!f x. FUNPOW f 0 x = x) /\
     (!f n x. FUNPOW f (SUC n) x = FUNPOW f n (f x))
FUNPOW_SUC
  |- !f n x. FUNPOW f (SUC n) x = f (FUNPOW f n x)
```

On this basis, an *ad hoc* but useful collection of over two hundred and fifty elementary theorems of arithmetic are proved when HOL is built and stored in the theory arithmetic. For a complete list of the available theorems, see *REFERENCE*. See also Section 3.6 for discussion of the LEAST operator, which returns the least number satisfying a predicate.

3.3.2.1 Grammar information

The following table gives the parsing status of the arithmetic constants.

Operator	Strength	Associativity
>=	450	non
<=	450	non
>	450	non
<	450	non
+	500	left
-	500	left
*	600	left
DIV	600	left
MOD	650	left
EXP	700	right

3.3.3 Numerals

The type num is usually thought of as being supplied with an infinite collection of numerals: 1, 2, 3, etc.. However, the HOL logic has no way to define such infinite families of constants; instead, all numerals other than 0 are actually built up from the constants introduced by the following definitions:

```
NUMERAL_DEF |- !x. NUMERAL x = x

BIT1        |- !n. BIT1 n = n + (n + SUC 0)
BIT2        |- !n. BIT2 n = n + (n + SUC(SUC 0))

ALT_ZERO    |- ZERO = 0
```

For example, the numeral 5 is represented by the term

NUMERAL(BIT1(BIT2 ZERO))

and the HOL parser and pretty-printer make such terms appear as numerals. This binary representation for numerals allows for asymptotically efficient calculation. Theorems supporting arithmetic calculations on numerals can be found in the numeral theory; these are mechanized by the reduce library. Thus, arithmetic calculations are performed by deductive steps in HOL. For example the following calculation of $2^{(1023+14)/9}$ takes approximately 4,200 primitive inference steps and returns in 30 milli-seconds.

1
```
- reduceLib.REDUCE_CONV ``2 EXP ((1023 + 14) DIV 9)``;

> val it = |- 2 ** ((1023 + 14) DIV 9) = 4153837486827862102824397063376076 8
```

Construction of numerals Numerals may of course be built using mk_comb, and taken apart with dest_comb; however, a more convenient interface to this functionality is provided by the functions mk_numeral, dest_numeral, and is_numeral (found in the structure numSyntax). These entry-points make use of an ML structure Arbnum which implements arbitrary precision numbers num. The following session shows how HOL numerals are constructed from elements of type num and how numerals are destructed. The structure Arbnum provides a full collection of arithmetic operations, using the usual names for the operations, e.g.+, *, -, etc..

2
```
- numSyntax.mk_numeral
    (Arbnum.fromString "3432432423423423234");
> val it = ``3432432423423423234`` : term

- numSyntax.dest_numeral it;
> val it = 3432432423423423234 : num

- Arbnum.+(it,it);
> val it = 6864864846846846468 : num
```

Numerals and the parser Simple digit sequences are parsed as decimal numbers, but the parser also supports the input of numbers in binary, octal and hexadecimal notation. Numbers may be written in binary and hexadecimal form by prefixing them with the strings `0b` and `0x` respectively. The 'digits' A–F in hexadecimal numbers may be written in upper or lower case. Binary numbers have their most significant digits left-most. In the interests of backwards compatibility, octal numbers are not enabled by default, but if the reference `base_tokens.allow_octal_input` is set to `true`, then octal numbers are those that appear with leading zeroes.

Finally, all numbers may be padded with underscore characters (`_`). These can be used to groups digits for added legibility and have no semantic effect.

Thus

3
```
- ``0xAA``;
> val it = ``170`` : term

- ``0b1010_1011``;
> val it = ``171`` : term

- base_tokens.allow_octal_input := true;
> val it = () : unit

- ``067``;
> val it = ``55`` : term
```

Numerals and Peano numbers Numerals are related to numbers built from `0` and `SUC` via the derived inference rule `num_CONV`, found in the `numLib` library.

```
   num_CONV : term -> thm
```

`num_CONV` can be used to generate the 'SUC' equation for any non-zero numeral. For example:

4
```
- load "numLib"; open numLib;

- num_CONV ``2``;
> val it = |- 2 = SUC 1 : thm

- num_CONV ``3141592653``;
> val it = |- 3141592653 = SUC 3141592652 : thm
```

The `num_CONV` function works purely by inference.

3.3.3.1 Overloading of arithmetic operators

When other numeric theories are loaded (such as those for the reals or integers), numerals are overloaded so that the numeral `1` can actually stand for a natural number, an integer or a real value. The parser has a pass of overloading resolution in which it attempts to determine the actual type to give to a numeral. For example, in the following session, the theory of integers is loaded, whereupon the numeral `2` is taken to be an integer.

5

```
- load "integerTheory";
> val it = () : unit

- ``2``;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = `2` : term

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

In order to precisely specify the desired type, the user can use single character suffixes (‘`n`’ for the natural numbers, and ‘`i`’ for the integers):

6

```
- type_of ``2n``;
> val it = `:num` : hol_type

- type_of ``42i``;
> val it = `:int` : hol_type
```

A numeric literal for a HOL type other than `num`, such as `42i`, is represented by the application of an *injection* function of type `num -> ty` to a numeral. The injection function is different for each type `ty`. See Section 3.3.4 for further discussion.

The functions `mk_numeral`, `dest_numeral`, and `is_numeral` only work for numerals, and not for numeric literals with character suffixes other than `n`. For information on how to install new character suffixes, consult the `add_numeral_form` entry in *REFERENCE*.

3.3.4 Integers

There is an extensive theory of integers in HOL. The type of integers is constructed as a quotient on pairs of natural numbers. A standard collection of operators are defined. These are overloaded with similar operations on the natural numbers, and on the real numbers. The constants defined in the integer theory include those found in the following table.

Constant	Overloaded symbol	Strength	Associativity
int_ge	>=	450	non
int_le	<=	450	non
int_gt	>	450	non
int_lt	<	450	non
int_add	+	500	left
int_sub	-	500	left
int_neg	~	900	trueprefix
int_mul	*	600	left
/		600	left
%		650	left
int_exp	**	700	right
int_of_num	&		prefix

The overloaded symbol `& : num -> int` denotes the injection function from natural numbers to integers. The following session illustrates how overloading and integers literals are treated.

1

```
Term ‘1i = &(1n + 0n)‘;
> val it = ‘1 = & (1 + 0)‘ : term

- show_numeral_types := true;
> val it = () : unit

- Term ‘&1 = &(1n + 0n)‘;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = ‘1i = & (1n + 0n)‘ : Term.term
```

3.3.5 Rational numbers

The type of rationals is constructed as a quotient on ordered pairs of integers (the numerator and the denominator of a fraction) whose second component must not be zero. To make things easier in the HOL theory, the sign of a rational number is always moved to the numerator. So, the denominator is always positive.

A standard collection of operators, which are overloaded with similar operations on the integers, are defined. These include those found in the following table. Injection from natural numbers is supported by the overloaded symbol `& : num -> rat` and the suffix `q`.

Constant	Overloaded symbol	Strength	Associativity
rat_geq	>=	450	non
rat_leq	<=	450	non
rat_gre	>	450	non
rat_les	<	450	non
rat_add	+	500	left
rat_sub	-	500	left
rat_ainv	~	900	trueprefix
rat_minv			
rat_mul	*	600	left
rat_div	/	600	left
rat_of_num	&		

The theorems in the theory of rational numbers include field properties, arithmetic rules, manipulation of (in)equations and their reduction to (in)equations between integers, properties of less-than relations and the density of rational numbers. For details, consult *REFERENCE* and the source files.

3.3.6 Real numbers

There is an extensive collection of theories that make up the development of real numbers and analysis in HOL, due to John Harrison [4]. We will only give a sketchy overview of the development; the interested reader should consult *REFERENCE* and Harrison's thesis.

The axioms for the real numbers are derived from the 'half reals' which are constructed from the 'half rationals'. This part of the development is recorded in `hratTheory` and `hrealTheory`, but is not used once the reals have been constructed. The real axioms are derived in the theory `realaxTheory`. A standard collection of operators on the reals, and theorems about them, is found in `realaxTheory` and `realTheory`. The operators and their parse status are listed in the following table.

Constant	Overloaded symbol	Strength	Associativity
real_ge	>=	450	non
real_lte	<=	450	non
real_gt	>	450	non
real_lt	<	450	non
real_add	+	500	left
real_sub	-	500	left
real_neg	~	900	trueprefix
real_mul	*	600	left
real_div	/	600	left
pow		700	right
real_of_num	&		prefix

On the basis of `realTheory`, the following sequence of theories is constructed:

topology Topologies and metric spaces, including metric on the real line.

nets Moore-Smith convergence nets, and special cases like sequences.

seq Sequences and series of real numbers.

lim Limits, continuity and differentiation.

powser Power series.

transc Transcendental functions, *e.g.*, exp, sin, cos, ln, root, sqrt, pi, tan, asn, acs, atn. Also the Kurzweil-Henstock gauge integral the fundamental theorem of calculus, and McLaurin's theorem.

HOL also includes a basic theory of the complex numbers (`complexTheory`), where the type `complex` is a type abbreviation for a pair of real numbers. The $\sqrt{-1}$ value is the HOL constant `i`. Numerals are supported (with the suffix `c` available to force numerals to be parsed as complex numbers). The standard arithmetic operations are defined, with the appropriate theorems proved about them.

3.3.7 Probability theory

A foundational construction of probability theory developed by Joe Hurd [5]. First a type of boolean sequences is defined to model an infinite sequence of coin flips. Next a probability function is formalized which takes as input a set of boolean sequences, and returns a real number between 0 and 1. Unfortunately not all sets can be assigned a probability (the Banach-Tarski paradox), rather the sets that can be assigned a probability are called *measurable sets*, and this is also formalized in the HOL theory.

Building on this foundation, the probability theory is used to define a sampling function that takes an infinite sequence of coin flips and a positive integer N, and returns an integer n in the range $0 \leq n < N$, picked uniformly at random from the available choices. This sampling function for the uniform distribution is later used to verify the Miller-Rabin primality test.

3.3.8 Bit vectors

HOL provides a theory of bit vectors, or n-bit words. For example, in computer architectures one finds: bytes/octets ($n = 8$), half-words ($n = 16$), words ($n = 32$) and long-words ($n = 64$). In the theory `words`, bit vectors are represented as *finite Cartesian products*: an n-bit word is given type *bool*$[\alpha]$ where the *size* of the type α determines

the word length n. This approach comes from an idea of John Harrison, which was presented at TPHOLs 2005.[7]

3.3.8.1 Finite Cartesian products

The HOL theory `fcp` introduces an infix type operator `**`, which is used to represent finite Cartesian products.[8] The type `'a ** 'b`, or equivalently *'a*[*'b*], is conceptually equivalent to:

$$\underbrace{\texttt{'a \# 'a \# } \cdots \texttt{ \# 'a}}_{\texttt{dimindex('b)}}$$

where `dimindex('b)` is the cardinality of `univ(:'b)` when *'b* is finite and is one when it is infinite. Thus, *'a*[*num*] is similar to *'a*, and *'a*[*bool*] is similar to *'a* # *'a*. Numeral type names are supported, so one can freely work with indexing sets of any size, e.g. the type *32* has thirty-two elements and *bool*[*32*] represents 32-bit words.

The *components* of a finite Cartesian product are accessed with an indexing function

```
fcp_index : 'a['b]→num→'a
```

which is typically written with an infixed apostrophe: `x ' i` denotes the value of vector `x` at position `i`. Typically, indices are constrained to be less than the size of *'b*.

The following theorem shows that two Cartesian products `x` and `y` are equal if, and only if, all of their components `x ' i` and `y ' i` are equal:

```
CART_EQ: |- !x y. (x = y) = !i. i < dimindex (:'a) ==> (x ' i = y ' i)
```

In order to construct Cartesian products, the theory `fcp` introduces a binder `FCP`, which is characterised by the following theorems:

```
FCP_BETA: |- !i. i < dimindex (:'a) ==> ($FCP g ' i = g i)
FCP_ETA:  |- !x. (FCP i. x ' i) = x
```

The theorem `FCP_BETA` shows that the components of `$FCP g` are determined by the function `g:`*num*`→'a`. The theorem `FCP_ETA` shows that a binding can be eliminated when all of the components are identical to that of `x`. These two theorems, together with `CART_EQ`, can be found in the *simpset* fragment `fcpLib.FCP_ss`.

Finite Cartesian products provide a good means to model n-bit words. That is to say, the type *bool*[*'a*] can represent a binary word whose length n corresponds with the size of the type *'a*. The binder `FCP` provides a flexible means for defining words – one can supply a function `f:`*num*`→`*bool* that gives the word's bit values, each of which can be accessed using the indexing map `fcp_index`.

[7] The current theory subsumes previous word theories – it evolved from a development based on an equivalence class construction. Wai Wong's word theory, which was based on Paul Curzon's `rich_list` theory, is no longer distributed with HOL. The principle advantages of the current theory are that there is just one theory for all word sizes and that word length side conditions are not required.

[8] The theory of finite Cartesian products was ported from HOL Light.

3.3.8.2 Bit theory

The theory `bit` defines some bit operations over the natural numbers, e.g. `BITS`, `SLICE`, `BIT`, `BITWISE` and `BIT_MODIFY`. In this context, natural numbers are treated as binary words of unbounded length. The operations in `bit` are primarily defined using `DIV`, `MOD` and `EXP`. For example, from the definition of `BIT`, the following theorem holds:

```
|- !b n. BIT b n = ((n DIV 2 ** b) MOD 2 = 1)
```

This theory is used in the development of the word theory and it also provides a mechanism for the efficient evaluation of some word operations via the theory `numeral_bit`.

3.3.8.3 Words theory

The theory `words` introduces a selection of polymorphic constants and operations, which can be type instantiated to any word size. For example, word addition has type:

$+$:*bool*[α]$\rightarrow$*bool*[α]$\rightarrow$*bool*[α]

If *'a* is instantiated to *32* then this operation corresponds with 32-bit addition. All theorems about word operations apply for any word length.[9]

Some basic operations The function `w2n`:*bool* [α]$\rightarrow$*num* gives the natural number value of a word. If $x \in \mathsf{T}^{\{0,1,\ldots,n-1\}}$ is a finite Cartesian product representing an n-bit word then its natural number value is:

$$\mathrm{w2n}(x) = \sum_{i=0}^{n-1} \textbf{if } x_i \textbf{ then } 2^i \textbf{ else } 0\ .$$

The length of a word (the number n) is given by the function `word_len`:*bool* [α]$\rightarrow$*num*. The function `n2w`:*num*$\rightarrow$*bool* [α] maps from a number to a word and is defined in HOL by:

```
|- !n. n2w n = FCP i. BIT i n
```

The suffix `w` is used to denote word literals, e.g. `255w` is the same as `n2w 255`.

The function `w2w`:*bool* [α]$\rightarrow$*bool* [β] provides word-to-word conversion (casting):

```
|- !w. w2w w = n2w (w2n w)
```

[9] Note that it is impossible to introduce words of length zero because all types must be inhabited, and hence their size will always be greater than or equal to one.

If β is smaller than α then the higher bits of `w` will be lost (it performs bit extraction), otherwise the longer word will have the same value as the original (in effect providing zero padding). However, if one were treating `w` as a two's complement number then the word needs to be sign extended, i.e.

$$\begin{array}{ll} (-\text{ve}) & 1b_{n-2}\cdots b_0 \;\mapsto\; 1\cdots 11b_{n-2}\cdots b_0 \\ (+\text{ve}) & 0b_{n-2}\cdots b_0 \;\mapsto\; 0\cdots 00b_{n-2}\cdots b_0 \end{array}$$

The function `sw2sw:`*`bool`*`[`α`]`$\rightarrow$*`bool`*`[`β`]` provides this sign extending version of `w2w`.

A collection of operations are provided for mapping to and from strings and number (digit) lists, e.g.

```
|- word_to_dec_string 876w = "876"
```

and

```
|- word_to_hex_list 876w = [12; 6; 3]
```

These function are specialised versions of `w2s` and `w2l` respectively.

Concatenation The operation `word_concat:`*`bool`*`[`α`]`$\rightarrow$*`bool`*`[`β`]`$\rightarrow$*`bool`*`[`γ`]` concatenates words. Note that the return type is not constrained. This means that two sixteen bit words can be concatenated to give a word of any length – which may be smaller or larger than the expect value of 32. The related function `word_join` does return a word of the expected length, i.e. of type *bool*[$\alpha + \beta$]; however, the concatenation operation is more useful because we often want *bool*[*32*] and not the logically distinct *bool*[*16+16*].

Signed and unsigned words Words can be *viewed* as being either signed (using the two's complement representation) or as being unsigned. However, this is not made explicit within the theory[10] and all of the arithmetic operations are defined using the natural numbers, i.e. via `w2n` and `n2w`. In particular, addition and multiplication work naturally (have the same definition) under the two's complement representation. This is not the case however with word-to-word conversion, orderings, division and right shifting, where signed and unsigned variants are needed. When operating over the natural numbers, some of the two's complement versions have slightly unnatural looking presentations. For example, with the signed (two's complement) version of "less than" we have `255w < (0w:word8)` because the word `255w` is actually taken to be representing the integer -1, whereas the unsigned version is more natural: `0w <+ (255w:word8)`.

[10]Words are not tagged as being signed/unsigned. Mappings to/from the integers (`w2i` and `i2w`) are provided in the theory `integer_word`.

Bit field operations The standard Boolean bit field operations are provided, i.e. bitwise negation (one's complement), conjunction, disjunction and exclusive-or. These functions are defined quite naturally using the Cartesian product binder; for example, bitwise conjunction is defined by:

```
|- !v w. v && w = FCP i. v ' i /\ w ' i .
```

There is also a collection of word *reduction* operations, which reduce bit vectors to 1-bit words, e.g.

$$\text{reduce\_and}(x)\ '\ 0 = \bigwedge_{i=0}^{n-1} x_i\ .$$

The functions `word_lsb`, `word_msb` and `word_bit(i)` give the bit value of a word at positions 0, $n-1$ and i respectively. Four operations are provided for selecting bit fields, or sub-words: `word_bits` (`--`), `word_signed_bits` (`---`), `word_slice` (`''`) and `word_extract` (`><`). For example, `word_bits 4 1` will select four bits starting from bit position 1. The slice function is an in-place variant (it zeroes bits outside of the bit range) and the extract function combines `word_bits` with a word cast (`w2w`). The operation `word_signed_bits` is similar to `word_bits`, except that it sign-extends the bit field.

The `bit_field_insert` operation inserts a bit field. For example,

```
bit_field_insert 5 2 a b
```

is word `b` with bits 5–2 replaced by bits 3–0 of `a`.

A word's bit ordering can be flipped over with `word_reverse`, i.e. bit zero is swapped with bit $n-1$ and so forth.

The function `word_modify`$:(\mathit{num} \to \mathit{bool} \to \mathit{bool}) \to \mathit{bool}\ [\alpha] \to \mathit{bool}\ [\alpha]$ changes a word by applying a map at each bit position. This operation provides a very flexible and convenient mechanism for manipulating words, e.g.

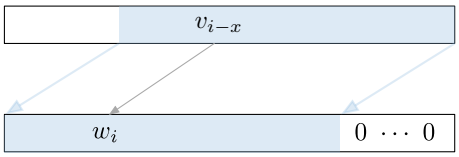
```
word_modify (λi b. if EVEN i then ~b else b) w
```

negates the bits of `w` that are in even positions. Of course, the binder `FCP` also provides a very general means to represent words using a predicate e.g. `$FCP ODD` represents a word where all the odd bits are set.

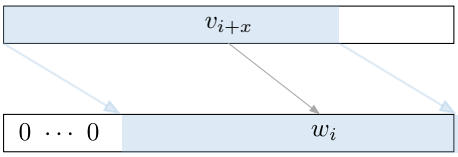
Shifts Six types of shifts are provided: logical shift left/right (`<<` and `>>>`), arithmetic shift right (`>>`), rotate left/right (`#<<` and `#>>`) and rotate right extended by 1 place (`word_rrx`). These shifts are illustrated in Figure 3.1 and are defined in a similar manner to the other bit field operations. For example, rotating right is defined by:

```
|- !w n. w #>> x = FCP i. w ' (i + x) MOD dimindex (:'a) .
```

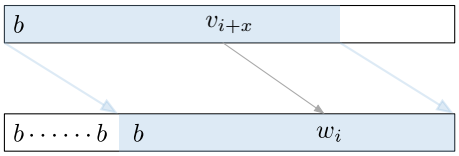
Rotating left by x places is defined as rotating right by $n - x \bmod n$ places.



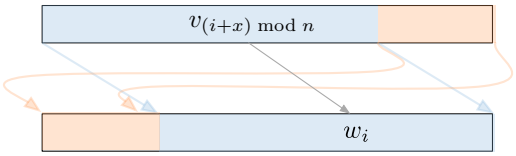
(a) Logical shift left: `w = v << x`.



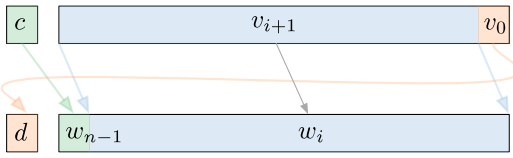
(b) Logical shift right: `w = v >>> x`.



(c) Arithmetic shift right: `w = v >> x`.



(d) Rotate right: `w = v #>> x`.



(e) Rotate right extended by 1 place: `(d,w) = word_rrx (c,v)`.

Figure 3.1: Shift operations.

Arithmetic and orderings The arithmetic operations are: addition, subtraction, unary minus (two's complement), logarithm (base-2), multiplication, modulus and division (signed and unsigned). These operations are defined with respect to the natural numbers. For example, word addition is defined by:

```
|- !v w. v + w = n2w (w2n v + w2n w)
```

The `+` on the left-hand side is word addition and on the right it is natural number addition.

All of the standard word orderings are provided, with signed and unsigned versions of $<$, $\leq$, $>$ and $\geq$. The unsigned versions are suffixed with a plus; for example, `<+` is unsigned "less than".

Constants The word theory also defines a few word constants:

Constant	Value	Binary
`word_T` or `UINT_MAXw`	$2^l - 1$	$11 \cdots 11$
`word_L` or `INT_MINw`	2^{l-1}	$10 \cdots 00$
`word_H` or `INT_MAXw`	$2^{l-1} - 1$	$01 \cdots 11$

List of bit vector operations A list of operations is provided in the table below.

Operation	Symbol	Type	Description
n2w		*num*→*bool*[α]	Map from a natural number
w2n		*bool*[α]→*num*	Map to a natural number
w2w		*bool*[α]→*bool*[β]	Map word-to-word (unsigned)
sw2sw		*bool*[α]→*bool*[β]	Map word-to-word (signed)
w2l		*num*→*bool*[α]→*num list*	Map word to digit list
l2w		*num*→*num list*→*bool*[α]	Map digit list to word
w2s		*num*→(*num*→*char*)→*bool*[α]→*string*	Map word to string
s2w		*num*→(*char*→*num*)→*string*→*bool*[α]	Map string to word
word_len		*bool*[α]→*num*	The word length
word_lsb		*bool*[α]→*bool*	The least significant bit
word_msb		*bool*[α]→*bool*	The most significant bit
word_bit		*num*→*bool*[α]→*bool*	Test bit position
word_bits	--	*num*→*num*→*bool*[α]→*bool*[α]	Select a bit field
word_signed_bits	---	*num*→*num*→*bool*[α]→*bool*[α]	Sign-extend selected bit field
word_slice	''	*num*→*num*→*bool*[α]→*bool*[α]	Set bits outside field to zero
word_extract	><	*num*→*num*→*bool*[α]→*bool*[β]	Extract (cast) a bit field
word_reverse		*bool*[α]→*bool*[α]	Reverse the bit order
bit_field_insert		*num*→*num*→*bool*[α]→ *bool*[β]→*bool*[β]	Insert a bit field
word_modify		(*num*→*bool*→*bool*)→ *bool*[α]→*bool*[α]	Apply a function to each bit
word_join		*bool*[α]→*bool*[β]→*bool*[$\alpha+\beta$]	Join words
word_concat	@@	*bool*[α]→*bool*[β]→*bool*[γ]	Concatenate words
concat_word_list		*bool*[α] *list*→*bool*[β]	Concatenate list of words
word_replicate		*num*→*bool*[α]→*bool*[β]	Replicate word
word_or	||	*bool*[α]→*bool*[α]→*bool*[α]	Bitwise disjunction
word_xor	??	*bool*[α]→*bool*[α]→*bool*[α]	Bitwise exclusive-or
word_and	&&	*bool*[α]→*bool*[α]→*bool*[α]	Bitwise conjunction
word_nor	~||	*bool*[α]→*bool*[α]→*bool*[α]	Bitwise NOR
word_xnor	~??	*bool*[α]→*bool*[α]→*bool*[α]	Bitwise XNOR
word_nand	~&&	*bool*[α]→*bool*[α]→*bool*[α]	Bitwise NAND
word_reduce		(*bool*→*bool*→*bool*)→ *bool*[α]→*bool*[*1*]	Word reduction
reduce_or		*bool*[α]→*bool*[*1*]	Disjunction reduction
reduce_xor		*bool*[α]→*bool*[*1*]	Exclusive-or reduction
reduce_and		*bool*[α]→*bool*[*1*]	Conjunction reduction
reduce_nor		*bool*[α]→*bool*[*1*]	NOR reduction
reduce_xnor		*bool*[α]→*bool*[*1*]	XNOR reduction
reduce_nand		*bool*[α]→*bool*[*1*]	NAND reduction
word_1comp	~	*bool*[α]→*bool*[α]	One's complement
word_2comp	-	*bool*[α]→*bool*[α]	Two's complement
word_add	+	*bool*[α]→*bool*[α]→*bool*[α]	Addition
word_sub	-	*bool*[α]→*bool*[α]→*bool*[α]	Subtraction

continued on next page

continued from previous page

Operation	Symbol	Type	Description
word_mul	*	*bool*[α]→*bool*[α]→*bool*[α]	Multiplication
word_div	//	*bool*[α]→*bool*[α]→*bool*[α]	Division (unsigned)
word_sdiv	/	*bool*[α]→*bool*[α]→*bool*[α]	Division (signed)
word_mod		*bool*[α]→*bool*[α]→*bool*[α]	Modulus
word_log2		*bool*[α]→*bool*[α]	Logarithm base-2
word_lsl	<<	*bool*[α]→*num*→*bool*[α]	Logical shift left
word_lsr	>>>	*bool*[α]→*num*→*bool*[α]	Logical shift right
word_asr	>>	*bool*[α]→*num*→*bool*[α]	Arithmetic shift right
word_ror	#>>	*bool*[α]→*num*→*bool*[α]	Rotate right
word_rol	#<<	*bool*[α]→*num*→*bool*[α]	Rotate left
word_rrx		*bool*#*bool*[α]→*bool*#*bool*[α]	Rotate right extended by 1 place
word_lt	<	*bool*[α]→*bool*[α]→*bool*	Signed "less than"
word_le	<=	*bool*[α]→*bool*[α]→*bool*	Signed "less than or equal"
word_gt	>	*bool*[α]→*bool*[α]→*bool*	Signed "greater than"
word_ge	>=	*bool*[α]→*bool*[α]→*bool*	Signed "greater than or equal"
word_lo	<+	*bool*[α]→*bool*[α]→*bool*	Unsigned "less than"
word_ls	<=+	*bool*[α]→*bool*[α]→*bool*	Unsigned "less than or equal"
word_hi	>+	*bool*[α]→*bool*[α]→*bool*	Unsigned "greater than"
word_hs	>=+	*bool*[α]→*bool*[α]→*bool*	Unsigned "greater than or equal"

3.4 Sequences

HOL provides theories for various kinds of sequences: finite lists, lazy lists, paths, and finite strings.

3.4.1 Lists

HOL lists are inductively defined finite sequences where each element in a list has the same type. The theory `list` introduces the unary type operator α list by a type definition and a standard collection of list processing functions are defined. The primitive constructors `NIL` and `CONS`

```
NIL  : 'a list
CONS : 'a -> 'a list -> 'a list
```

are used to build lists and have been defined from the representing type for lists. The HOL parser has been specially modified to parse the expression `[]` into `NIL`, to parse the expression `h::t` into `CONS h t`, 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.

Based on the inductive characterization of the type, the following fundamental theorems about lists are proved and stored in the theory `list`.

```
list_Axiom
  |- !x f. ?fn. (fn [] = x) /\ (!h t. fn (h::t) = f(fn t)h t)
list_INDUCT
  |- !P. P [] /\ (!t. P t ==> (!h. P(h::t))) ==> (!l. P l)
list_CASES
  |- !l. (l = []) \/ (?t h. l = h::t)
CONS_11
  |- !h t h' t'. (h::t = h'::t') = (h = h') /\ (t = t')
NOT_NIL_CONS
  |- !h t. ~([] = h::t)
NOT_CONS_NIL
  |- !h t. ~(h::t = [])
```

The theorem `list_Axiom` shown above is analogous to the primitive recursion theorem on the natural numbers discussed above in Section 3.3.1.3. It states the validity of primitive recursive definitions on lists, and can be used to justify any such definition. The ML function `new_recursive_definition` uses 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.

The induction theorem for lists, `list_INDUCT`, provides the main proof tool used to reason about operations that manipulate lists. The theorem `list_CASES` is used to perform case analysis on whether a list is empty or not.

The theorem `CONS_11` shows that `CONS` is injective; the theorems `NOT_NIL_CONS` and `NOT_CONS_NIL` show that `NIL` and `CONS` are distinct, i.e., cannot give rise to the same structure. Together, these three theorems are used for equational reasoning about lists.

The predicate `NULL` and the selectors `HD` and `TL` are defined in the theory `list` by

```
NULL |- NULL [] /\ (!h t. ~NULL(h::t))
HD   |- !h t. HD(h::t) = h
TL   |- !h t. TL(h::t) = t
```

The following functions on lists are also defined in the theory `list`.

Case expressions Compound HOL expressions that branch based on whether a term is an empty or non-empty list have the surface syntax (roughly borrowed from ML)

```
case e1
 of [] => e2
  | (h::t) => e3
```

Such an expression is translated to list_case e_2 $(\lambda h\ t.\ e_3)$ e_1 where the constant `list_case` is defined as follows:

```
list_case_def
  |- (!v f. list_case v f [] = v) /\
     (!v f a0 a1. list_case v f (a0::a1) = f a0 a1)
```

List membership Membership in a list, MEM, is defined as follows:

```
MEM |- (!x. MEM x [] = F) /\
       (!x h t. MEM x (h::t) = (x = h) \/ MEM x t)
```

Concatenation of lists Binary list concatenation (APPEND) may also be denoted by the infix operator ++; thus the expression L1 ++ L2 is translated into APPEND L1 L2. The concatenation of a list of lists into a list is achieved by FLAT.

```
APPEND
  |- (!l. APPEND [] l = l) /\
     (!l1 l2 h. APPEND (h::l1) l2 = h::APPEND l1 l2)
FLAT
  |- (FLAT [] = []) /\ (!h t. FLAT(h::t) = h ++ FLAT t)
```

Numbers and lists The length (LENGTH) and size (list_size) of a list are related notions. The size of a list takes account of the size of each element of the list (given by parameter $f : \alpha \to \mathsf{num}$), while the length of the list ignores the size of each list element. The alternate length definition (LEN) is tail-recursive. Numbers can also be used to index into lists, extracting the element at the specified position.

```
LENGTH
  |- (LENGTH [] = 0) /\ (!h t. LENGTH (h::t) = SUC(LENGTH t))
LEN_DEF
  |- (!n. LEN [] n = n) /\ !h t n. LEN (h::t) n = LEN t (n + 1)
list_size_def
  |- (!f. list_size f [] = 0) /\
     !f a0 a1. list_size f (a0::a1) = 1 + (f a0 + list_size f a1))
EL
  |- (!l. EL 0 l = HD l) /\ (!l n. EL (SUC n) l = EL n (TL l))
```

Note that the extraction of the nth element (EL) of a list starts its indexing from 0. If the length of the list ℓ is less than or equal to n, the result of EL n ℓ is unspecified.

Mapping functions over lists There are functions for mapping a function $f : \alpha \to \beta$ over a single list (MAP) or a function $f : \alpha \to \beta \to \gamma$ over two lists (MAP2).

```
MAP
  |- (!f. MAP f [] = []) /\
     (!f h t. MAP f (h::t) = f h::MAP f t)
MAP2
  |- (!f. MAP2 f [] [] = []) /\
     !f h1 t1 h2 t2. MAP2 f (h1::t1) (h2::t2) = f h1 h2::MAP2 f t1 t2
```

The behaviour of MAP2 in the cases when it is given lists of unequal lengths is unspecified.

Predicates over lists Predicates can be applied to lists in a universal sense (the predicate must hold of every element in the list) or an existential sense (the predicate must hold of some element in the list). This functionality is supported by EVERY and EXISTS, respectively. The elimination of all elements in list not satisfying a given predicate is performed by FILTER.

```
EVERY_DEF
  |- (!P. EVERY P [] = T) /\
     (!P h t. EVERY P (h::t) = P h /\ EVERY P t)
EXISTS_DEF
  |- (!P. EXISTS P [] = F) /\
     (!P h t. EXISTS P (h::t) = P h \/ EXISTS P t)
FILTER
  |- (!P. FILTER P [] = []) /\
     (!P h t. FILTER P (h::t) = if P h then h::FILTER P t else FILTER P t)
ALL_DISTINCT
  |- (ALL_DISTINCT [] = T) /\
     (!h t. ALL_DISTINCT (h::t) = ~MEM h t /\ ALL_DISTINCT t)
```

The predicate ALL_DISTINCT holds on a list just in case no element in the list is equal to any other.

Folding Applying a binary function $f : \alpha \to \beta \to \beta$ pairwise through a list and accumulating the result is known as *folding*. At times, it is necessary to do this operation from left-to-right (FOLDL), and at others the right-to-left direction (FOLDR) is required.

```
FOLDL
  |- (!f e. FOLDL f e [] = e) /\
     (!f e x l. FOLDL f e (x::l) = FOLDL f (f e x) l)
FOLDR
  |- (!f e. FOLDR f e [] = e) /\
     (!f e x l. FOLDR f e (x::l) = f x (FOLDR f e l))
```

List reversal The reversal of a list (REVERSE) and its tail recursive counterpart REV are defined in list.

```
REVERSE_DEF
  |- (REVERSE [] = []) /\
     (!h t. REVERSE (h::t) = REVERSE t ++ [h])
REV_DEF
  |- (!acc. REV [] acc = acc) /\
     (!h t acc. REV (h::t) acc = REV t (h::acc))
```

Conversion to sets Lists can be converted to sets (LIST_TO_SET) by partial application of MEM. The somewhat terse definition is used to derive the theorem IN_LIST_TO_SET.

```
LIST_TO_SET
  |- LIST_TO_SET = combin$C MEM
IN_LIST_TO_SET
  |- x IN LIST_TO_SET l = MEM x l
```

Further support for translating between different kinds of collections may be found in the `container` theory.

Pairs and lists Two lists of equal length may be component-wise paired by the `ZIP` operation. The result is unspecified when the lists are not the same length. The inverse operation, `UNZIP`, translates a list of pairs into a pair of lists.

```
ZIP
  |- (ZIP ([],[]) = []) /\
     (!x1 l1 x2 l2. ZIP (x1::l1,x2::l2) = (x1,x2)::ZIP (l1,l2))
UNZIP_THM
  |- (UNZIP [] = ([],[])) /\
     (UNZIP ((x,y)::t) = let (L1,L2) = UNZIP t in (x::L1,y::L2))
```

Alternate access Lists are essentially treated in a stack-like manner. However, at times it is convenient to access the last element (`LAST`) of a non-empty list directly. The last element of a non-empty list is dropped by `FRONT`.

```
LAST_DEF
  |- !h t. LAST (h::t) = if t = [] then h else LAST t
FRONT_DEF
  |- !h t. FRONT (h::t) = if t = [] then [] else h::FRONT t
APPEND_FRONT_LAST
  |- !l. ~(l = []) ==> (FRONT l ++ [LAST l] = l)
```

Joining the front part and the last element of a non-empty list yields the original list. Both `LAST` and `FRONT` are unspecified on empty lists.

Prefix checking The relation capturing whether a list ℓ_1 is a prefix of ℓ_2 (`isPREFIX`) can be defined by recursion. The infix `<<=` can also be used as notation for this partial order.

```
isPREFIX_THM
  |- ([] <<= l <=> T) /\
     (h::t <<= [] <=> F) /\
     (h1::t1 <<= h2::t2 <=> (h1 = h2) /\ t1 <<= t2)
```

The above theorem states that: the empty list is a prefix of any other list (clause 1); that no non-empty list is a prefix of the empty list (clause 2); and that a non-empty list is a prefix of another non-empty list if the first elements of the lists are the same, and if the tail of the first is a prefix of the tail of the second.

For a complete list of available theorems in `list`, see *REFERENCE*. Further development of list theory can be found in `rich_list`.

3.4.1.1 List permutations and sorting

The sorting theory defines a notion of two lists being permutations of each other, then defines a general notion of sorting, then shows that Quicksort is a sorting function.

List permutation Two lists are in permutation if they have exactly the same members, and each member has the same number of occurrences in both lists. One definition (PERM) that captures this relationship is the following:

```
PERM_DEF
  |- !L1 L2. PERM L1 L2 = !x. FILTER ($= x) L1 = FILTER ($= x) L2
PERM_IND =
  |- !P.
       P [] [] /\
       (!x l1 l2. P l1 l2 ==> P (x::l1) (x::l2)) /\
       (!x y l1 l2. P l1 l2 ==> P (x::y::l1) (y::x::l2)) /\
       (!l1 l2 l3. P l1 l2 /\ P l2 l3 ==> P l1 l3)
      ==>
      !l1 l2. PERM l1 l2 ==> P l1 l2
```

A derived induction theorem (PERM_IND) is very useful in proofs about permutations.

Sorting A list is R-sorted if R holds pairwise through the list. This notion (SORTED) is captured by a recursive definition. Then a function of type

```
('a -> 'a -> bool) -> 'a list -> 'a list
```

is a sorting function (SORTS) with respect to R if it delivers a permutation of its input, and the result is R-sorted.

```
SORTED_DEF
  |- (SORTED R [] = T) /\
     (SORTED R [x] = T) /\
     (SORTED R (x::y::rst) = R x y /\ SORTED R (y::rst))
SORTS_DEF
  |- !f R. SORTS f R = !l. PERM l (f R l) /\ SORTED R (f R l)
```

Quicksort is defined in the usual functional programming style, and it is indeed a sorting function, provided R is a transitive and total relation.

```
QSORT_DEF =
  |- (QSORT ord [] = []) /\
     (QSORT ord (h::t) =
        let (l1,l2) = PARTITION (\y. ord y h) t
        in
          QSORT ord l1 ++ [h] ++ QSORT ord l2)
QSORT_SORTS
  |- !R. transitive R /\ total R ==> SORTS QSORT R
```

3.4.2 Possibly infinite sequences (`llist`)

The theory `llist` contains the definition of a type of possibly infinite sequences. This type is similar to the "lazy lists" of programming languages like Haskell, hence the name of the theory. The `llist` theory has a number of constants that are analogous to constants in the theory of finite lists. The `llist` versions of these constants have the same names, but with a capital 'L' prepended. Thus, some of the core constants in this theory are:

```
LNIL  : 'a llist
LCONS : 'a -> 'a llist -> 'a llist
LHD   : 'a llist -> 'a option
LTL   : 'a llist -> 'a llist option
```

The `LHD` and `LTL` constants return `NONE` when applied to the empty sequence, `LNIL`. This use of an option type is another way of modelling the essential partiality of these constants. (In the theory of lists, the analogous `HD` and `TL` functions simply have unspecified values when applied to empty lists.)

The type `llist` is not inductive, and there is no primitive recursion theorem supporting the definition of functions that have domains of type `llist`. Rather, `llist` is a coinductive type, and has an axiom that justifies the definition of (co-)recursive functions that map *into* the `llist` type:

```
llist_Axiom
   |- !f : 'a -> ('a # 'b) option.
        ?g : 'a -> 'b llist.
          (!x. LHD (g x) = OPTION_MAP SND (f x)) /\
          (!x. LTL (g x) = OPTION_MAP (g o FST) (f x))
```

An equivalent form of the above is

```
llist_Axiom_1
   |- !f. ?g.
        !x. g x =
            case f x
             of NONE => LNIL
              | SOME (x',y) => LCONS y (g x')
```

Other constants in the theory `llist` include `LMAP`, `LFINITE`, `LNTH`, `LTAKE`, `LDROP`, and `LFILTER`. Their types are

```
LMAP    : ('a -> 'b) -> 'a llist -> 'b llist
LFINITE : 'a llist -> bool
LNTH    : num -> 'a llist -> 'a option
LTAKE   : num -> 'a llist -> 'a list option
LDROP   : num -> 'a llist -> 'a llist option
LFILTER : ('a -> bool) -> 'a llist -> 'a llist
```

They are characterised by the following theorems

```
LMAP
   |- (LMAP f LNIL = LNIL) /\
      (LMAP f (LCONS h t) = LCONS (f h) (LMAP f t))
LFINITE_THM
   |- (LFINITE LNIL = T) /\
      (LFINITE (LCONS h t) = LFINITE t)
LNTH_THM
   |- (!n. LNTH n LNIL = NONE) /\
      (!h t. LNTH 0 (LCONS h t) = SOME h) /\
      (!n h t. LNTH (SUC n) (LCONS h t) = LNTH n t)
LTAKE_THM
   |- (LTAKE 0 l = SOME []) /\
      (LTAKE (SUC n) LNIL = NONE) /\
      (LTAKE (SUC n) (LCONS h t) = OPTION_MAP (CONS h) (LTAKE n t)
LDROP_THM
   |- (LDROP 0 ll = SOME ll) /\
      (LDROP (SUC n) ll = NONE) /\
      (LDROP (SUC n) (LCONS h t) = LDROP n t)
LFILTER_THM
   |- (LFILTER P LNIL = LNIL) /\
      (LFILTER P (LCONS h t) = if P h then LCONS h (LFILTER P t)
                                      else LFILTER P t)
```

Concatenation Two lazy lists may be concatenated by `LAPPEND`. If the first lazy list is infinite, elements of the second are inaccessible in the result. A lazy list of lazy lists can be flattened to a lazy list by `LFLATTEN`.

```
LAPPEND
   |- (!x. LAPPEND LNIL x = x) /\
      (!h t x. LAPPEND (LCONS h t) x = LCONS h (LAPPEND t x))
LFLATTEN_THM
   |- (LFLATTEN LNIL = LNIL) /\
      (!tl. LFLATTEN (LCONS LNIL t) = LFLATTEN t) /\
      (!h t tl. LFLATTEN (LCONS (LCONS h t) tl) =
                  LCONS h (LFLATTEN (LCONS t tl)))
```

Lists and lazy lists Mapping back and forth from lists to lazy lists is accomplished by `fromList` and `toList`:

```
fromList
   |- (fromList [] = LNIL) /\
      (!h t. fromList (h::t) = LCONS h (fromList t))
toList_THM
   |- (toList LNIL = SOME []) /\
      (!h t. toList (LCONS h t) = OPTION_MAP (CONS h) (toList t))
```

Proof principles Finally, there are two very important proof principles for proving that two `llist` values are equal. The first states that two sequences are equal if they return the same prefixes of length n for all possible values of n:

```
LTAKE_EQ |- (ll1 = ll2) = (!n. LTAKE n ll1 = LTAKE n ll2)
```

This theorem is subsequently used to derive the bisimulation principle:

```
LLIST_BISIMULATION
         |- (ll1 = ll2) =
            ?R. R ll1 ll2 /\
                !ll3 ll4. R ll3 ll4 ==>
                          (ll3 = LNIL) /\ (ll4 = LNIL) \/
                          (LHD ll3 = LHD ll4) /\
                          R (THE (LTL ll3)) (THE (LTL ll4))
```

The principle of bisimulation states that two `llist` values l_1 and l_2 are equal if (and only if) it is possible to find a relation R such that

- R relates the two values, i.e., $R\ l_1\ l_2$; and
- if R holds of any two values l_3 and l_4, then either
  - both l_3 and l_4 are empty; or
  - the head elements of l_3 and l_4 are the same, and the tails of those two values are again related by R

Of course, a possible R would be equality itself, but the strength of this theorem is that other, more convenient relations can also be used.

3.4.3 Labelled paths (`path`)

The theory `path` defines a binary type operator (α, β)`path`, which stands for possibly infinite paths of the following form

$$\alpha_1 \xrightarrow{\beta_1} \alpha_2 \xrightarrow{\beta_2} \alpha_3 \xrightarrow{\beta_3} \cdots \alpha_n \xrightarrow{\beta_n} \alpha_{n+1} \xrightarrow{\beta_{n+1}} \cdots$$

The `path` type is thus an appropriate model for reduction sequences, where the α parameter corresponds to "states", and the β parameter corresponds to the labels on the arrows.

The model of (α, β)`path` is $\alpha \times ((\alpha \times \beta)$`llist`$)$. The type of paths has two constructors:

```
stopped_at : 'a -> ('a,'b) path
pcons      : 'a -> 'b -> ('a,'b) path -> ('a,'b) path
```

The `stopped_at` constructor returns a path containing just one state, and no transitions. (Thus, the reduction sequence has "stopped at" this state.) The `pcons` constructor takes a state, a label, and a path, and returns a path which is now headed by the state argument, and which moves from that state via the label argument to the path. Graphically, `pcons` $x\ l\ p$ is equal to

$$x \xrightarrow{l} \underbrace{p_1 \xrightarrow{l_1} p_2 \xrightarrow{l_2} \cdots}_{p}$$

Other constants defined in theory `path` include

```
finite  : ('a,'b) path -> bool
first   : ('a,'b) path -> 'a
labels  : ('a,'b) path -> 'b llist
last    : ('a,'b) path -> 'a
length  : ('a,'b) path -> num option
okpath  : ('a -> 'b -> 'a -> bool) -> ('a,'b) path -> bool
pconcat : ('a,'b) path -> 'b -> ('a,'b) path -> ('a,'b) path
pmap    : ('a -> 'c) -> ('b -> 'd) -> ('a,'b)path -> ('c,'d)path
```

The `first` function returns the first element of a path. There always is such an element, and the defining equations are

```
first_thm  |- (first (stopped_at x) = x) /\
              (first (pcons x l p) = x)
```

On the other hand, the `last` function does not always have a well-specified value, though it still has nice characterising equations:

```
last_thm   |- (last (stopped_at x) = x) /\
              (last (pcons x l p) = last p)
```

The theorem for `finite` has a similar feel, but has a definite value (`F`, or *false*) on infinite paths), whereas the value of `last` on such paths is unspecified:

```
finite_thm |- (finite (stopped_at x) = T) /\
              (finite (pcons x l p) = finite p)
```

The function `pconcat` concatenates two paths, linking them with a provided label. If the first path is infinite, then the result is equal to that first path. The defining equation is

```
pconcat_thm |- (pconcat (stopped_at x) lab p2 = pcons x lab p2) /\
               (pconcat (pcons x r p) lab p2 =
                    pcons x r (pconcat p lab p2)
```

These equations are true even when the first argument to `pconcat` is an infinite path.

The `okpath` predicate tests whether or not a path is a valid transition given a ternary transition relation. Its characterising theorem is

```
okpath_thm |-
   (okpath R (stopped_at x)) /\
   (okpath R (pcons x r p) = R x r (first p) /\ okpath R p)
```

There is also an induction principle that simplifies reasoning about finite R-paths:

```
finite_okpath_ind |-
    (!x. P (stopped_at x)) /\
    (!x r p. okpath R p /\ finite p /\ R x r (first p) /\ P p ==>
             P (pcons x r p)) ==>
    !p. okpath R p /\ finite p ==> P p
```

One can show that a set P of paths are all R-paths with the co-induction principle:

```
okpath_co_ind |-
   !P.
      (!x r p. P (pcons x r p) ==> R x r (first p) /\ P p) ==>
      !p. P p ==> okpath R p
```

3.4.4 Character strings (string)

The theory string defines a type of characters and a type of finite strings built from those characters, along with a useful suite of definitions for operating on strings.

Characters The type char is represented by the numbers less than 256. Two constants are defined: CHR : num → char and ORD : char → num. The following theorems hold:

```
CHR_ORD  |- !a. CHR (ORD a) = a
ORD_CHR  |- !r. r < 256 = (ORD (CHR r) = r)
```

Character literals can also be entered using ML syntax, with a hash character immediately followed by a string literal of length one. Thus:

1
```
- val t = ``f #"c" #"\n"``;
<<HOL message: inventing new type variable names: 'a>>
> val t = ``f #"c" #"\n"`` : term

- dest_term ``#"\t"``;
> val it = COMB(``CHR``, ``9``) : lambda
```

Strings The type string is an alias for the type char list. All functions and predicates over lists are thus available for use over strings. Some of these constants are overloaded so that they are printed (and can be parsed) with names that are more appropriate for the particular case of lists of characters.

For example, NIL and CONS over strings have alternative names EMPTYSTRING and STRING respectively:

```
EMPTYSTRING : string
STRING      : char -> string -> string
```

The HOL parser maps the syntax "" to EMPTYSTRING, and the HOL printer inverts this. The parser expands string literals of the form "$c_1c_2\ldots c_n$" to the compound term

STRING c_1 (STRING c_2 ... (STRING c_{n-1} (STRING c_n EMPTYSTRING)) ...)

Of course, one could also write

```
- ``[#"a"; #"b"]``;                                                    2
> val it = ``"ab"`` : term
```

String literals can be constructed using the various special escape sequences that are used in ML. For example, \n for the newline character, and a backslash followed by three decimal digits for characters of the given number.

```
- val t = ``"foo bar\n\001"``;                                         3
> val t = ``"foo bar\n\^A"`` : term
```

Note that if one wants to use the control-character syntax with the caret that the pretty-printer has chosen to use in printing the given string, and this occurs inside a quotation, then the caret will need to be doubled. (See Section 5.1.3.)

There is also a destructor function DEST_STRING for strings which returns an option type.

```
DEST_STRING
  |- (DEST_STRING "" = NONE) /\
     (DEST_STRING (STRING c rst) = SOME(c,rst))
```

Case expressions Compound HOL expressions that branch based on whether a term is an empty or non-empty string have the surface syntax

```
case s
 of "" => e1
  | STRING c rst => e2
```

Such an expression is translated to string_case e_1 $(\lambda c\, rst.\ e_2)$ s where the constant string_case is defined as follows:

```
STRING_CASE_DEF
  |- (string_case b f ""  = b) /\
     (string_case b f (STRING c s) = f c s)
```

Length and concatenation A standard function LENGTH can be written STRLEN when applied to a string, and APPEND can be written as STRCAT. There are also theorems characterising these constants in stringTheory, though they are simply instantiations of results from listTheory:

```
STRLEN_THM
  |- (STRLEN "" = 0) /\
     (STRLEN (STRING c s) = 1 + STRLEN s)

STRCAT_EQNS =
  |- (STRCAT "" s = s) /\
     (STRCAT s "" = s) /\
     (STRCAT (STRING c s1) s2 = STRING c (STRCAT s1 s2))
```

3.5 Collections

Several different notions of a collection of elements are available in HOL: sets, multisets, relations, and finite maps.

3.5.1 Sets (pred_set)

An extensive development of set theory is available in the theory pred_set. Sets are represented by functions of the type $\alpha \rightarrow$ bool, i.e., they are so-called characteristic functions. One can use the type abbreviation α set instead of $\alpha \rightarrow$ bool. Sets may be finite or infinite. All of the elements in a set must have the same type.

Set membership is the basic notion that formalized set theory is based on. In HOL, membership is represented by a the infix constant IN, defined in theory bool for convenience.

```
IN_DEF   |- IN = \x f. f x
```

The IN operator is merely a way of applying the characteristic function to an item, as the following trivial consequence of the definition shows:

```
SPECIFICATION   |- !P x. x IN P = P x
```

Two sets are equal if they have the same elements.

```
EXTENSION   |- !s t. (s = t) = (!x. (x IN s) = (x IN t))
```

Empty and universal sets The empty set is the characteristic function that is constantly false. The constant `EMPTY` denotes the empty set; it may be written as $\{\}$ and $\emptyset$ (U+2205). The universal set, `UNIV`, on a type is the characteristic function that is always true for elements of that type.

```
EMPTY_DEF   |- {} = (\x. F)
UNIV_DEF    |- UNIV = (\x. T)
```

In addition to `UNIV` (perhaps with a type annotation `:'a set`), one may also write `univ(:'a)` to represent the universal set over type `:'a`. The Unicode syntax $\mathbb{U}$`(:'a)` means the same. The Unicode symbol for $\mathbb{U}$ is U+1D54C, and may not exist in many fonts.

One of these forms will be used to print `UNIV` by default. The user trace (see Section 6.2) `"Univ pretty-printing"` can be set to zero to cancel this behaviour. Additionally, the trace `"Unicode Univ printing"` can be used to stop the U+1D54C syntax from being used, even if the Unicode trace is set.

The symbols `univ` and $\mathbb{U}$ are high-priority prefixes (see Section 5.1.2.7), and overloaded patterns (see Section 5.1.2.3) mapping a value of the itself type to the corresponding `UNIV` constant. One effect is that one can write things like

```
FINITE univ(:'a)
```

without the need for parentheses around `FINITE`'s argument.

Insertion, union, and intersection The insertion (`INSERT`, written infix) of an element into a set is defined with a set comprehension. Set comprehension is discussed in the next subsection. Set union (`UNION`, written infix) and intersection (`INTER`, also infix) are given their usual definitions by set comprehension.

```
INSERT_DEF  |- !x s. x INSERT s = {y | (y = x) \/ y IN s}
UNION_DEF   |- !s t. s UNION t = {x | x IN s \/ x IN t}
INTER_DEF   |- !s t. s INTER t = {x | x IN s /\ x IN t}
```

`UNION` and `INTER` are binary operations. Indexed union and intersection operations, i.e., $\bigcup_{i \in P}$ and $\bigcap_{i \in P}$ are provided by the definitions of `BIGUNION` and `BIGINTER`.

```
BIGUNION    |- !P. BIGUNION P = {x | ?s. s IN P /\ x IN s}
BIGINTER    |- !P. BIGINTER P = {x | !s. s IN P ==> x IN s}
```

Both `BIGUNION` and `BIGINTER` reduce a set of sets to a set and thus have the type $((\alpha \to \mathsf{bool}) \to \mathsf{bool}) \to (\alpha \to \mathsf{bool})$.

Subsets Set inclusion (SUBSET, infix), proper set inclusion (PSUBSET, infix), and power set (POW) are defined as follows:

```
SUBSET_DEF  |- !s t. s SUBSET t = !x. x IN s ==> x IN t
PSUBSET_DEF |- !s t. s PSUBSET t = s SUBSET t /\ ~(s = t)
POW_DEF     |- !set. POW set = {s | s SUBSET set}
```

Set difference and complement The difference between two sets (DIFF, infix) is defined by a set comprehension. Based on that, the deletion of a single element (DELETE, infix) from a set is straightforward. Since the universe of a type is always available via UNIV, the complement (COMPL) of a set may be taken.

```
DIFF_DEF    |- !s t. s DIFF t = {x | x IN s /\ ~(x IN t)}
DELETE_DEF  |- !s x. s DELETE x = s DIFF {x}
COMPL_DEF   |- !P. COMPL P = UNIV DIFF P
```

Functions on sets The image of a function $f : \alpha \to \beta$ on a set (IMAGE) is defined with a set comprehension.

```
IMAGE_DEF   |- !f s. IMAGE f s = {f x | x IN s}
```

Injections, surjections, and bijections between sets are defined as follows:

```
INJ_DEF
     |- !f s t.
          INJ f s t =
          (!x. x IN s ==> f x IN t) /\
          !x y. x IN s /\ y IN s ==> (f x = f y) ==> (x = y)
SURJ_DEF
     |- !f s t.
          SURJ f s t =
          (!x. x IN s ==> f x IN t) /\
          !x. x IN t ==> ?y. y IN s /\ (f y = x)

BIJ_DEF |- !f s t. BIJ f s t = INJ f s t /\ SURJ f s t
```

Finite sets The finite sets (FINITE) are defined inductively as those built from the empty set by a finite number of insertions.

```
FINITE_DEF
  |- !s. FINITE s = !P. P {} /\ (!s. P s ==> !e. P (e INSERT s)) ==> P s
```

A set is infinite iff it is not finite, and there is an abbreviation in the system that parses ‘‘INFINITE s‘‘ into ‘‘~FINITE s‘‘. The pretty-printer reverses this transformation.

The finite sets have an induction theorem:

```
FINITE_INDUCT
  |- !P. P {} /\
        (!s. FINITE s /\ P s ==> !e. ~(e IN s) ==> P (e INSERT s))
        ==>  !s. FINITE s ==> P s
```

As mentioned, set operations apply to both finite and infinite sets. However, some operations, such as cardinality (CARD), are only defined for finite sets. The cardinality of an infinite set is not specified.

```
CARD_DEF
  |- (CARD {} = 0) /\
     !s. FINITE s ==>
         !x. CARD (x INSERT s) = if x IN s then CARD s else SUC (CARD s)
```

Since the finite and infinite sets are dealt with uniformly in pred_set, properties of operations on finite sets must explicitly include constraints about finiteness. For example the following theorem relating cardinality and subsets is only true for finite sets.

```
CARD_PSUBSET
  |- !s. FINITE s ==> !t. t PSUBSET s ==> CARD t < CARD s
```

An extensive suite of theorems dealing with finiteness and cardinality is available in pred_set.

Cross product The product of two sets (CROSS, infix) is defined with a set comprehension.

```
CROSS_DEF   |- !P Q. P CROSS Q = {p | FST p IN P /\ SND p IN Q}
```

Cardinality and cross product are related by the following theorem:

```
CARD_CROSS
  |- !P Q. FINITE P /\ FINITE Q ==> (CARD (P CROSS Q) = CARD P * CARD Q)
```

Recursive functions on sets Recursive functions on sets may be defined by well-founded recursion. Usually, the totality of such a function is established by measuring the cardinality of the (finite) set. However, another theorem may be used to justify a fold (ITSET) for finite sets. Provided a function $f : \alpha \to \beta \to \beta$ obeys a condition known as *left-commutativity*, namely, $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$, then f can be applied by folding it on the set in a tail-recursive fashion.

```
ITSET_EMPTY
  |- !f b. ITSET f {} b = b
COMMUTING_ITSET_INSERT
  |- !f s. (!x y z. f x (f y z) = f y (f x z)) /\ FINITE s ==>
           !x b. ITSET f (x INSERT s) b = ITSET f (s DELETE x) (f x b)
```

A recursive version is also available:

```
COMMUTING_ITSET_RECURSES
  |- !f e s b.
       (!x y z. f x (f y z) = f y (f x z)) /\ FINITE s ==>
       (ITSET f (e INSERT s) b = f e (ITSET f (s DELETE e) b))
```

For the full derivation, see the sources of `pred_set`. The definition of `ITSET` allows, for example, the definition of summing the results of a function on a finite set of elements, from which a recursive characterization and other useful theorems are derived.

```
SUM_IMAGE_DEF
  |- !f s. SIGMA f s = ITSET (\e acc. f e + acc) s 0
SUM_IMAGE_THM
  |- !f. (SIGMA f {} = 0) /\
         !e s. FINITE s ==>
               (SIGMA f (e INSERT s) = f e + SIGMA f (s DELETE e))
```

Other definitions and theorems There are more definitions in `pred_set`, but they are not as heavily used as the ones presented here. Similarly, most theorems in `pred_set` relate the various common set operations to each other, but do not express any deep theorems of set theory.

However, one notable theorem is Koenig's Lemma, which states that every finitely branching infinite tree has an infinite path. There are many ways to formulate this theorem, depending on how the notion of tree is formalized. In `pred_set`, finite branching is defined as a predicate on a relation.

```
finite_branching_def
  |- !R. finitely_branching R = !x. FINITE {y | R x y}
```

From this, the following version of Koenig's Lemma is stated and proved:

```
KoenigsLemma
  |- finitely_branching R ==>
       !x. ~FINITE {y | RTC R x y} ==>
           ?f. (f 0 = x) /\ !n. R (f n) (f (SUC n))
```

3.5.1.1 Syntax for sets

The special purpose set-theoretic notations $\{t_1; t_2; \ldots; t_n\}$ and $\{t \mid p\}$ are recognized by the HOL parser and printer when the theory `pred_set` is loaded.

The normal interpretation of $\{t_1; t_2; \ldots; t_n\}$ is the finite set containing just $t_1, t_2, \ldots, t_n$. This can be modelled by starting with the empty set and performing a sequence of insertions. For example, `{1;2;3;4}` parses to

```
1 INSERT (2 INSERT (3 INSERT (4 INSERT EMPTY)))
```

Set comprehensions The normal interpretation of {t | p} is the set of all ts such that p. In HOL, such syntax parses to: GSPEC(\(x_1,...,x_n).(t,p)) where $x_1, \ldots, x_n$ are those free variables that occur in both t and p if both have at least one free variable. If t or p has no free variables, then $x_1, \ldots, x_n$ are taken to be the free variables of the other term. If both terms have free variables, but there is no overlap, 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). For example,

```
{p+q | p < q /\ q < r}
```

parses to:

```
GSPEC(\(p,q). ((p+q), (p < q /\ q < r)))
```

where GSPEC is characterized by:

```
GSPECIFICATION  |- !f v. (v IN GSPEC f) = (?x. (v,T) = f x)
```

This somewhat cryptic specification can be understood by exercising an example. The syntax

```
a IN {p+q | p < q /\ q < r}
```

is mapped by the HOL parser to

```
a IN GSPEC(\(p,q). ((p+q), (p < q /\ q < r)))
```

which, by GSPECIFICATION, is equal to

```
?x. (a,T) = (\(p,q). ((p+q), (p < q /\ q < r))) x
```

The existentially quantified variable x has a pair type, so it can be replaced by a pair (p,q) and a paired-β-reduction can be performed, yielding

```
?(p,q). (a,T) = ((p+q), (p < q /\ q < r))
```

which is equal to the intended meaning of the original syntax:

```
?(p,q). (a = p+q) /\ (p < q /\ q < r)
```

Unambiguous set comprehensions There is also an unambiguous set comprehension syntax, which allows the user to specify which variables are to be quantified over in the abstraction that is the argument of `GSPEC`. Terms of the form

```
{ t | vs | P }
```

generate sets containing values of the form given by `t`, where the variables mentioned in `vs` must satisfy the constraint `P`. For example, the set

```
{ x + y | x | x < y }
```

is the set of numbers from `y` up to but not including `2 * y`. The set can be "read" computationally: draw out all those `x` that are less than `y`, and to each such `x` add `y`, thereby generating a set of numbers.

In the example above, the underlying `GSPEC` term will be

```
GSPEC (\x. (x + y, x < y))
```

The `vs` component of the unambiguous notation must be a single "variable structure" that might appear underneath a possibly paired abstraction as in section 3.2.3.1. In other words, this

```
{ x + y | (x,y) | x < y }
```

is fine, but this

```
{ x + y | x y | x < y }
```

will raise an error. (Additionally, the outermost parentheses around pairs in the `vs` position can be omitted.)

The unambiguous notation is printed by the pretty-printer whenever the set to be printed can not be expressed with the default notation, or if the trace variable with name `pp_unambiguous_comprehensions` is set to `true`.

3.5.2 Multisets (`bag`)

Multisets, also known as *bags*, are similar to sets, except that they allow repeat occurrences of an element. Whereas sets are represented by functions of type $\alpha \to$ bool, which signal the presence, or absence, of an element, multisets are represented by functions of type $\alpha \to$ num, which give the multiplicity of each element in the multiset. Multisets may be finite or infinite.

The type abbreviations α multiset and α bag can be used instead of $\alpha \to$ num.

Empty multiset The empty bag has no elements. Thus, the function implementing it returns 0 for every input.

```
EMPTY_BAG  |- EMPTY_BAG = K 0
```

The special syntax `{||}` can be used to represent the empty bag.

Membership Much of the theory can be based on the notion of membership in a bag. There are two notions: does an element occur at least n times in a bag (`BAG_INN`); and does an element occur in a bag at all (`BAG_IN`).

```
BAG_INN  |- BAG_INN e n b = (b e >= n)
BAG_IN   |- BAG_IN e b = BAG_INN e 1 b
```

Two bags are equal if all elements have the same tally.

```
BAG_EXTENSION
  |- !b1 b2. (b1 = b2) = (!n e. BAG_INN e n b1 = BAG_INN e n b2)
```

Sub-multiset A sub-bag relationship (`SUB_BAG`) holds between b_1 and b_2 provided that every element in b_1 occurs at least as often in b_2. The notion of a proper sub-bag (`PSUB_BAG`) is easily defined.

```
SUB_BAG
  |- SUB_BAG b1 b2 = !x n. BAG_INN x n b1 ==> BAG_INN x n b2
PSUB_BAG
  |- PSUB_BAG b1 b2 = SUB_BAG b1 b2 /\ ~(b1 = b2)
```

Insertion Inserting an element into a bag (`BAG_INSERT`) updates the tally for that element and leaves the others unchanged.

```
BAG_INSERT
  |- BAG_INSERT e b = (\x. if (x = e) then b e + 1 else b x)
```

Explicitly-given multisets are supported by the syntax `{|`t_1`;`t_2`;`$\ldots$`;`t_n`|}`, where there may, of course, be repetitions. This is modelled by starting with the empty multiset and performing a sequence of insertions. For example, `{|1; 2; 3; 2; 1|}` parses to

```
BAG_INSERT 1 (BAG_INSERT 2 (BAG_INSERT 3
                              (BAG_INSERT 2 (BAG_INSERT 1 {||}))))
```

Union and difference The union (BAG_UNION) and difference (BAG_DIFF) operations on bags both reduce to an arithmetic calculation on their elements. Deleting a single element from a bag may be expressed by taking the multiset difference with a single-element multiset; however, there is also a relational presentation (BAG_DELETE) which relates its first and last arguments only if the first contains exactly one more occurrence of the middle argument than the last. This is not the same as using BAG_DIFF to remove a one-element bag because it insists that the element being removed actually appear in the larger bag.

```
BAG_UNION
  |- BAG_UNION b c = \x. b x + c x
BAG_DIFF
  |- BAG_DIFF b1 b2 = \x. b1 x - b2 x
BAG_DELETE
  |- BAG_DELETE b0 e b = (b0 = BAG_INSERT e b)
```

Intersection, merge, and filter The intersection of two bags (BAG_INTER) takes the pointwise minimum. The dual operation, merging (BAG_MERGE), takes the pointwise maximum. A bag can be 'filtered' by a set to return the bag where all the elements not in the set have been dropped (BAG_FILTER).

```
BAG_INTER
  |- BAG_INTER b1 b2 = (\x. if (b1 x < b2 x) then b1 x else b2 x)
BAG_MERGE
  |- BAG_MERGE b1 b2 = (\x. if (b1 x < b2 x) then b2 x else b1 x)
BAG_FILTER_DEF
  |- BAG_FILTER P b = (\e. if P e then b e else 0)
```

Sets and Multisets Moving between bags and sets is accomplished by the following two definitions.

```
SET_OF_BAG
  |- SET_OF_BAG b = \x. BAG_IN x b
BAG_OF_SET
  |- BAG_OF_SET P = \x. if x IN P then 1 else 0
```

Image Taking the image of a function on a multiset to get a new multiset seems to be simply a matter of applying the function to each element of the multiset. However, there is a problem if f is non-injective and the multiset is infinite. For example, take the multiset consisting of all the natural numbers and apply $\lambda x.\ 1$ to each element. The resulting multiset would hold an infinite number of 1s. To avoid this requires some constraints: for example, stipulating that the function be only finitely non-injective, or that the input multiset be finite. Such conditions would be onerous in proof; the compromise is to map the multipicity of problematic elements to 0.

```
BAG_IMAGE_DEF
  |- BAG_IMAGE f b =
       \e. let sb = BAG_FILTER (\e0. f e0 = e) b
           in
             if FINITE_BAG sb then BAG_CARD sb else 0
```

Finite multisets The finite multisets (FINITE_BAG) are defined inductively as those built from the empty bag by a finite number of insertions.

```
FINITE_BAG
  |- FINITE_BAG b =
       !P. P EMPTY_BAG /\
           (!b. P b ==> (!e. P (BAG_INSERT e b))) ==> P b
```

The finite multisets have an induction theorem, and also a strong induction theorem.

```
FINITE_BAG_INDUCT
  |- !P. P {||} /\
         (!b. P b ==> (!e. P (BAG_INSERT e b)))
         ==> (!b. FINITE_BAG b ==> P b)

STRONG_FINITE_BAG_INDUCT
  |- !P. P {||} /\
         (!b. FINITE_BAG b /\ P b ==> !e. P (BAG_INSERT e b))
         ==> (!b. FINITE_BAG b ==> P b)
```

The cardinality (BAG_CARD) of a multiset counts the total number of occurrences. It is only specified for finite multisets.

```
BAG_CARD_THM
  |- (BAG_CARD {||} = 0) /\
     (!b. FINITE_BAG b ==>
            !e. BAG_CARD (BAG_INSERT e b) = BAG_CARD b + 1)
```

Recursive functions on multisets Recursive functions on multiset may be defined by wellfounded recursion. Usually, the totality of such a function is established by measuring the cardinality of the (finite) multiset. However, a fold (ITBAG) for finite sets is provided. Provided a function $f : \alpha \to \beta \to \beta$ obeys a condition known as *left-commutativity*, namely, $f\ x\ (f\ y\ z) = f\ y\ (f\ x\ z)$, then f can be applied by folding it on the multiset in a tail-recursive fashion.

```
ITBAG_EMPTY
  |- !f acc. ITSET f {||} acc = acc
COMMUTING_ITBAG_INSERT
  |- !f b. (!x y z. f x (f y z) = f y (f x z)) /\ FINITE_BAG b ==>
           !x a. ITBAG f (BAG_INSERT x b) a = ITBAG f b (f x a)
```

A recursive version is also available:

```
COMMUTING_ITBAG_RECURSES
  |- !f e b a. (!x y z. f x (f y z) = f y (f x z)) /\ FINITE_BAG b ==>
              (ITBAG f (BAG_INSERT e b) a = f e (ITBAG f b a))
```

3.5.3 Relations (`relation`)

Mathematical relations can be represented in HOL by the type $\alpha \to \beta \to$ bool. (In most applications, the type of a relation is an instance of $\alpha \to \alpha \to$ bool, but the extra generality doesn't hurt.) The theory `relation` provides definitions of basic properties and operations on relations, defines various kinds of orders and closures, defines wellfoundedness and proves the wellfounded recursion theorem, and develops some basic results used in Term Rewriting.

Basic properties The following basic properties of relations are defined.

```
transitive_def
  |- transitive R = !x y z. R x y /\ R y z ==> R x z
reflexive_def
  |- reflexive R = (!x. R x x)
irreflexive_def
  |- irreflexive R = (!x. ~R x x)
symmetric_def
  |- symmetric R = (!x y. R x y = R y x)
antisymmetric_def
  |- antisymmetric R = (!x y. R x y /\ R y x ==> (x = y))
equivalence_def
  |- equivalence R = reflexive R /\ symmetric R /\ transitive R
trichotomous
  |- trichotomous R = !a b. R a b \/ R b a \/ (a = b)
total_def
  |- total R = (!x y. R x y \/ R y x)
```

Basic operations The following basic operations on relations are defined: the empty relation (`EMPTY_REL`), relation composition (`O`, infix), inversion (`inv`), domain (`RDOM`), and range (`RRANGE`).

```
EMPTY_REL_DEF
  |- !x y. EMPTY_REL x y = F
O_DEF
  |- $O R1 R2 x z = ?y. R1 x y /\ R2 y z
inv_DEF
  |- inv R x y = R y x
RDOM_DEF
  |- RDOM R x = ?y. R x y
RRANGE
  |- RRANGE R y = ?x. R x y
```

Set operations lifted to work on relations include subset (RSUBSET, infix), union (RUNION, infix), intersection (RINTER, infix), complement (RCOMPL), and universe (RUNIV).

```
RSUBSET
  |- $RSUBSET R1 R2 = !x y. R1 x y ==> R2 x y
RUNION
  |- $RUNION R1 R2 x y = R1 x y \/ R2 x y
RINTER
  |- $RINTER R1 R2 x y = R1 x y /\ R2 x y
RCOMPL
  |- RCOMPL R x y = ~R x y
RUNIV
  |- RUNIV x y = T
```

Orders A sequence of definitions capturing various notions of order are made in relation.

```
PreOrder
  |- PreOrder R = reflexive R /\ transitive R
Order
  |- Order Z = antisymmetric Z /\ transitive Z
WeakOrder
  |- WeakOrder Z = reflexive Z /\ antisymmetric Z /\ transitive Z
StrongOrder
  |- StrongOrder Z = irreflexive Z /\ antisymmetric Z /\ transitive Z
LinearOrder
  |- LinearOrder R = Order R /\ trichotomous R
WeakLinearOrder
  |- WeakLinearOrder R = WeakOrder R /\ trichotomous R
StrongLinearOrder
  |- StrongLinearOrder R = StrongOrder R /\ trichotomous R
```

Closures The transitive closure (TC) of a relation $R : \alpha \to \alpha \to \mathsf{bool}$ is defined inductively, as the least relation including R and closed under transitivity. Similarly, the reflexive-transitive closure (RTC) is defined to be the least relation closed under transitivity and reflexivity.

```
TC_DEF
  |- TC R a b =
       !P. (!x y. R x y ==> P x y) /\
           (!x y z. P x y /\ P y z ==> P x z) ==> P a b
RTC_DEF
  |- RTC R a b =
       !P. (!x. P x x) /\
           (!x y z. R x y /\ P y z ==> P x z) ==> P a b
```

From these definitions, one can recover the initial rules.

```
TC_RULES
  |- !R. (!x y. R x y ==> TC R x y) /\
         (!x y z. TC R x y /\ TC R y z ==> TC R x z)
RTC_RULES
  |- !R. (!x. RTC R x x) /\
         (!x y z. R x y /\ RTC R y z ==> RTC R x z)
RTC_RULES_RIGHT1
  |- !R. (!x. RTC R x x) /\
         (!x y z. RTC R x y /\ R y z ==> RTC R x z)
```

Notice that RTC_RULES, in keeping with the definition of RTC, extends an R-step from x to y with a a sequence of R-steps from y to z to construct RTC x z. The theorem RTC_RULES_RIGHT1 first makes a sequence of R steps and then a single R step to form RTC x z. Similar alternative theorems are proved for case analysis and induction.

For example, TC_CASES1 and TC_CASES2 in the following decompose RTC R x z to either R x y followed by RTC R y z (TC_CASES1) or RTC R x y followed by R y z (TC_CASES2).

```
TC_CASES1
  |- !R x z. TC R x z ==> R x z \/ ?y. R x y /\ TC R y z
TC_CASES2
  |- !R x z. TC R x z ==> R x z \/ ?y. TC R x y /\ R y z

RTC_CASES1
  |- !R x y. RTC R x y = (x = y) \/ ?u. R x u /\ RTC R u y
RTC_CASES2
  |- !R x y. RTC R x y = (x = y) \/ ?u. RTC R x u /\ R u y
RTC_CASES_RTC_TWICE
  |- !R x y. RTC R x y = ?u. RTC R x u /\ RTC R u y
```

As well as the basic induction theorems for TC and RTC, there are so-called *strong* induction theorems, which have stronger induction hypotheses.

```
TC_INDUCT
  |- !R P. (!x y. R x y ==> P x y) /\
           (!x y z. P x y /\ P y z ==> P x z)
           ==> !u v. TC R u v ==> P u v
RTC_INDUCT
  |- ! R P. (!x. P x x) /\
            (!x y z. R x y /\ P y z ==> P x z) ==>
            (!x y. RTC R x y ==> P x y)
TC_STRONG_INDUCT
  |- !R P. (!x y. R x y ==> P x y) /\
           (!x y z. P x y /\ P y z /\ TC R x y /\ TC R y z ==> P x z) ==>
           (!u v. TC R u v ==> P u v)
RTC_STRONG_INDUCT
  |- !R P. (!x. P x x) /\
           (!x y z. R x y /\ RTC R y z /\ P y z ==> P x z) ==>
           (!x y. RTC R x y ==> P x y)
```

Variants of these induction theorems are also available which break apart the closure from the left or right, as for the case analysis theorems.

The reflexive (RC) and symmetric closures (SC) are straightforward to define. The equivalence closure (EQC) is the symmetric then transitive then reflexive closure of R.

```
RC_DEF   |- RC R x y = (x = y) \/ R x y
SC_DEF   |- SC R x y = R x y \/ R y x
EQC_DEF  |- EQC R = RC (TC (SC R))
```

Wellfounded relations A relation R is wellfounded (WF) if every non-empty set has an R-minimal element. Wellfoundedness is used to justify the principle of wellfounded induction (WF_INDUCTION_THM).

```
WF_DEF
  |- !R. WF R = !B. (?w. B w) ==> ?min. B min /\ !b. R b min ==> ~B b
WF_INDUCTION_THM
  |- !R WF R ==> !P. (!x. (!y. R y x ==> P y) ==> P x) ==> !x. P x
```

The *wellfounded part* (WFP) of a relation can be inductively defined, from which its rules, case-analysis theorem and induction theorems may be derived.

```
WFP_DEF
  |- WFP R a = !P. (!x. (!y. R y x ==> P y) ==> P x) ==> P a
WFP_RULES
  |- !R x. (!y. R y x ==> WFP R y) ==> WFP R x
WFP_CASES
  |- !R x. WFP R x = !y. R y x ==> WFP R y
WFP_INDUCT
  |- !R P. (!x. (!y. R y x ==> P y) ==> P x)
           ==> !x. WFP R x ==> P x
WFP_STRONG_INDUCT
  |- !R. (!x. WFP R x /\ (!y. R y x ==> P y) ==> P x)
         ==> !x. WFP R x ==> P x
```

Wellfoundedness can also be used to justify a general recursion theorem. Intuitively, a collection of recursion equations can be admitted into the HOL logic with no loss of consistency provided that every possible sequence of recursive calls is finite. Wellfounded relations are used to capture this notion: if there is a wellfounded relation R on the domain of the desired function such that every sequence of recursive calls is R-decreasing, then the recursion equations specify a unique total function and the equations can be admitted into the logic.

The recursion theorems WFREC_COROLLARY and WF_RECURSION_THM use the notion of a function restriction (RESTRICT) in order to force the recursive function to be applied to R-smaller arguments in recursive calls..

```
RESTRICT_DEF
  |- !f R x. RESTRICT f R x = \y. if R y x then f y else ARB

WFREC_COROLLARY
  |- !M R f. (f = WFREC R M) ==> WF R ==> !x. f x = M (RESTRICT f R x) x

WF_RECURSION_THM
  |- !R. WF R ==> !M. ?!f. !x. f x = M (RESTRICT f R x) x
```

The theorems `WF_INDUCTION_THM` and `WFREC_COROLLARY` are used to automate recursive definitions; see Section 4.5. A few basic operators for wellfounded relations are also defined, along with theorems stating that they propagate wellfoundedness.

```
inv_image_def  |- !R f. inv_image R f = \x y. R (f x) (f y)

WF_inv_image   |- !R f. WF R ==> WF (inv_image R f)
WF_SUBSET      |- !R P. WF R /\ (!x y. P x y ==> R x y) ==> WF P
WF_TC          |- !R. WF R ==> WF (TC R)
WF_Empty       |- WF EMPTY_REL
```

Term Rewriting A few basic definitions from Term Rewriting theory (the diamond property (`diamond`), the Church-Rosser property (`CR` and `WCR`), and Strong Normalization (`SN`)) appear in `relation`.

```
diamond_def
  |- diamond R = !x y z. R x y /\ R x z ==> ?u. R y u /\ R z u
CR_def
  |- CR R = diamond (RTC R)
WCR_def
  |- WCR R = !x y z. R x y /\ R x z ==> ?u. RTC R y u /\ RTC R z u
SN_def
  |- SN R = WF (inv R)
```

From those, Newman's Lemma is proved.

```
Newmans_lemma  |- !R. WCR R /\ SN R ==> CR R
```

3.5.4 Finite maps (`finite_map`)

The theory `finite_map` formalizes a type (α, β) `fmap` of finite functions. These notionally have type $\alpha \to \beta$, but additionally have only finitely many elements in their domain. Finite maps are useful for formalizing substitutions and arrays. The representing type is $\alpha \to \beta + \mathsf{one}$, where only a finite number of the α map to a β and the rest map to `one`. The syntax α `|->` β is recognized by the parser as an alternative to (α, β) `fmap`.

Basic notions The empty map (FEMPTY), the updating of a map (FUPDATE), the application of a map to an argument (FAPPLY), and the domain of a map (FDOM) are the main notions in the theory.

```
FEMPTY  : 'a |-> 'b
FUPDATE : ('a |-> 'b) -> 'a # 'b -> ('a |-> 'b)
FAPPLY  : ('a |-> 'b) -> 'a -> 'b
FDOM    : ('a |-> 'b) -> 'a set
```

The HOL parser and printer will treat the syntax `f ' x` as the application of finite map `f` to argument `x`, ie, as `FAPPLY f x`. The notation `f |+ (x,y)` represents `FUPDATE f (x,y)`, i.e., the updating of finite map `f` by the pair `(x,y)`.

The basic constants have obscure definitions, from which more useful properties are then derived. FAPPLY_FUPDATE_THM relates map update with map application. fmap_EXT is an extensionality result: two maps are equal if they have the same domain and agree when applied to arguments in that domain. One can prove properties of finite maps by induction on the construction of the map (fmap_INDUCT). The cardinality of a finite map is just the cardinality of its domain (FCARD_DEF); from this a recursive characterization (FCARD_FUPDATE) is derived.

```
FAPPLY_FUPDATE_THM
  |- !f a b x. (f |+ (a,b)) ' x = (if x = a then b else f ' x)
fmap_EXT
  |- !f g. (f = g) =
           (FDOM f = FDOM g) /\ (!x. x IN FDOM f ==> (f ' x = g ' x))
fmap_INDUCT
  |- !P. P FEMPTY /\
         (!f. P f ==> !x y. ~(x IN FDOM f) ==> P (f |+ (x,y))) ==> !f. P f
FCARD_DEF  |- FCARD fm = CARD (FDOM fm)
FCARD_FUPDATE
  |- !fm a b. FCARD(fm |+ (a,b)) =
                if a IN FDOM fm then FCARD fm else 1 + FCARD fm
```

Iterated updates (FUPDATE_LIST) to a map are useful. The infix notation |++ may also be used. For example, `fm |++ [(k1,v1); (k2,v2)]` is equal to `(fm |+ (k1,v1)) |+ (k2,v2)`.

```
FUPDATE_LIST  |- FUPDATE_LIST = FOLDL FUPDATE
FUPDATE_LIST_THM
  |- !f. (f |++ [] = f) /\
         (!h t. f |++ (h::t) = (f |+ h) |++ t)
```

Domain and range The domain of a finite map is the set of elements that it applies to; this can be characterized recursively (FDOM_FUPDATE). The range of a map is defined in the usual way.

```
FDOM_FUPDATE
  |- !f a b. FDOM (f |+ (a,b)) = a INSERT (FDOM f)
FRANGE_DEF
  |- FRANGE f = {y | ?x. x IN FDOM f /\ (f ' x = y)}
```

A finite map may have its domain (DRESTRICT) or range (RRESTRICT) restricted by intersection with a set. These notions have recursive versions as well (DRESTRICT_FUPDATE and RRESTRICT_FUPDATE).

```
DRESTRICT_DEF
  |- !f r. (FDOM (DRESTRICT f r) = (FDOM f) INTER r) /\
           (!x. DRESTRICT f r ' x =
                  (if x IN ((FDOM f) INTER r) then f ' x else FEMPTY'x))
RRESTRICT_DEF
  |- !f r. (FDOM (RRESTRICT f r) = {x | x IN FDOM f /\ f ' x IN r}) /\
           (!x. RRESTRICT f r ' x =
                  (if x IN (FDOM f) /\ f ' x IN r then f ' x
                   else FEMPTY ' x))
DRESTRICT_FUPDATE
  |- !f r x y.
       DRESTRICT (f |+ (x,y)) r =
         if x IN r then (DRESTRICT f r) |+ (x,y) else DRESTRICT f r
RRESTRICT_FUPDATE
  |- !f r x y.
       RRESTRICT (f |+ (x,y)) r =
         if y IN r then (RRESTRICT f r) |+ (x,y)
                   else RRESTRICT (DRESTRICT f (COMPL {x})) r)
```

The removal of a single element from the domain of a map (\\, infix) is a simple application of (DRESTRICT), but sufficiently useful to deserve its own definition. Again, this concept has a alternate recursive presentation (DOMSUB_FUPDATE_THM).

```
fmap_domsub
  |- (fm \\ k) = DRESTRICT fm (COMPL {k})
DOMSUB_FUPDATE_THM
  |- !fm k1 k2 v. (fm |+ (k1,v)) \\ k2 =
                   if (k1 = k2) then (fm \\ k2) else (fm \\ k2) |+ (k1, v)
```

Union and sub-maps Unlike set union, the union of two finite maps (FUNION_DEF) is not symmetric: the domain of the first map takes precedence. The notion of a finite map being a submap of another (SUBMAP, infix) is an extension of how subsets are formalized.

```
FUNION_DEF
  |- !f g.
       (FDOM (FUNION f g) = FDOM f UNION FDOM g) /\
       !x. FUNION f g ' x = (if x IN FDOM f then f ' x else g ' x)
SUBMAP_DEF
  |- !f g. (f SUBMAP g) = (!x. x IN FDOM f ==> x IN FDOM g /\
                           (f ' x = g ' x))
```

Finite maps and functions As much as possible, finite maps should be like ordinary functions. Thus, if `f` is a finite map, then `FAPPLY f` is an ordinary function. Similarly, there is an operation for *totalizing* a finite map (`lookup`) so that an application of it returns an ordinary function, the range of which is the option type. An ordinary function can be turned into a finite map by restricting the function to a finite set of arguments (`FUN_FMAP_DEF`).

```
lookup_DEF
  |- FLOOKUP f x = (if x IN FDOM f then SOME (f ' x) else NONE)
FUN_FMAP_DEF
  |- !f P. FINITE P ==>
           (FDOM (FUN_FMAP f P) = P) /\
           (!x. x IN P ==> (FUN_FMAP f P ' x = f x))
```

Composition of maps There are three new definitions of composition, determined by whether the composed functions are finite maps or not. The composition of two finite maps (`f_o_f`, infix) has domain constraints attached. Composition of a finite map with an ordinary function (`o_f`, infix) applies the finite map first, then the ordinary function. Composition of an ordinary function with a finite map (`f_o`, infix) applies the ordinary function and then the finite map; the application of the ordinary function is achieved by turning it into a finite map.

```
f_o_f_DEF
  |- !f g.
       (FDOM (f f_o_f g) = (FDOM g) INTER {x | g ' x IN FDOM f}) /\
       !x. x IN FDOM (f f_o_f g) ==> ((f f_o_f g) ' x = f ' (g ' x))
o_f_DEF
  |- !f g.
       (FDOM (f o_f g) = FDOM g) /\
       !x. x IN FDOM (f o_f g) ==> ((f o_f g) ' x = f (g ' x))
f_o_DEF
  |- (f f_o g) = f f_o_f (FUN_FMAP g {x | g x IN FDOM f})
```

3.6 While Loops

It is a curious fact that higher order logic, although a logic of total functions, allows the definition of functions that don't seem total, at least from a computational perspective. An example is `WHILE`-loops. The following equation is derived in theory `while`:

```
WHILE  |- !P g x. WHILE P g x = if P x then WHILE P g (g x) else x
```

Clearly, if `P` in this theorem was instantiated to $\lambda x.$ `T`, the resulting instance of `WHILE` would 'run forever' if executed. Why is such an "obviously" partial function definable in

HOL? The answer lies in a subtle definition of WHILE, [11] which uses the expressive power of HOL to surprising effect. Consider the following total and non-recursive function:

```
\x. if (?n. P (FUNPOW g n x))
     then FUNPOW g (@n. P (FUNPOW g n x) /\
                         !m.  m < n ==> ~P (FUNPOW g m x)) x
     else ARB
```

This function does a case analysis on the iterations of function g: the finite ones return the first value in the iteration at which P holds (i.e., when the iteration stops); the infinite ones are mapped to ARB. This function is used as the witness for f in the proof of the following theorem:

```
ITERATION
  |- !P g. ?f. !x. f x = if P x then x else f (g x)
```

From this, it is a simple application of Skolemization and new_specification to obtain the equation for WHILE.

Reasoning about WHILE loops The induction theorem for WHILE loops is proved by wellfounded induction, and carries wellfoundedness constraints limiting its application. In order to apply WHILE_INDUCTION, the instantiations for B and C must be known before a wellfounded relation for R is found and used to eliminate the constraints.

```
WHILE_INDUCTION
  |- !B C R.
       WF R /\ (!s. B s ==> R (C s) s) ==>
       !P. (!s. (B s ==> P (C s)) ==> P s) ==> !v. P v
```

A more refined level of support is provided by the standard Hoare Logic WHILE rule, phrased in terms of Hoare triples (HOARE_SPEC).

```
HOARE_SPEC_DEF
  |- !P C Q. HOARE_SPEC P C Q = !s. P s ==> Q (C s)
WHILE_RULE
  |- !R B C.
       WF R /\ (!s. B s ==> R (C s) s) ==>
       HOARE_SPEC (\s. P s /\ B s) C P ==>
       HOARE_SPEC P (WHILE B C) (\s. P s /\ ~B s)
```

As a follow-on, an operator for finding the least number with property P is defined.

```
LEAST_DEF  |- !P. $LEAST P = WHILE ($~ o P) SUC 0
```

A few theorems for reasoning about LEAST may be found in theory while.

3.7 Further Theories

Other theories of interest in HOL are listed and briefly described in Figure 3.2.

[11] The original idea is due to J Moore, who suggested it for use in ACL2.

`poset`	Partial Orders, Knaster-Tarski theorem
`divides`, `gcd`	Divisibility and the greatest common divisor.
`poly`	A theory of polynomials over $\mathbb{R}$, providing a collection of operations on polynomials, and theorems about them.
`Temporal_Logic`, `Omega_Automata`	Klaus Schneider's development of temporal logic and ω-automata.
`ctl`, `mu`	Computation Tree Logic and the μ-calculus. See Hasan Amjad's thesis.
`lbtree`	Possibly infinitely deep (i.e., co-algebraic) binary trees.
`inftree`	Possibly infinitely branching, algebraic trees

Figure 3.2: A selection of HOL theories

Chapter 4

Advanced Definition Principles

4.1 Datatypes

Although the HOL logic provides primitive definition principles allowing new types to be introduced, the level of detail is very fine-grained. The style of datatype definitions in functional programming languages provides motivation for a high level interface for defining algebraic datatypes.

The `Hol_datatype` function supports the definition of such data types; the specifications of the types may be recursive, mutually recursive, nested recursive, and involve records. The syntax of declarations that `Hol_datatype` accepts is found in Table 4.1.

```
Hol_datatype ‘[binding ;]* binding‘

        binding  ::=  ident = constructor-spec
                  |   ident = record-spec

constructor-spec ::=  [clause |]* clause

         clause  ::=  ident
                  |   ident of [hol_type =>]* hol_type

    record-spec  ::=  <| [ident : hol_type ;]* ident : hol_type |>
```

Table 4.1: Datatype Declaration

HOL maintains an underlying database of datatype facts called the `TypeBase`. This database is used to support various high-level proof tools (see Section 5.3), and is augmented whenever a `Hol_datatype` declaration is made. When a datatype is defined by `Hol_datatype`, the following information is derived and stored in the database.

- initiality theorem for the type
- injectivity of the constructors
- distinctness of the constructors

- structural induction theorem
- case analysis theorem
- definition of the 'case' constant for the type
- congruence theorem for the case constant
- definition of the 'size' of the type

When the HOL system starts up, the `TypeBase` already contains the relevant entries for the types `bool`, `prod`, `num`, `option`, and `list`.

Example: Binary trees The following ML declaration of a data type of binary trees

```
datatype ('a,'b) btree = Leaf of 'a
                       | Node of ('a,'b) btree * 'b * ('a,'b) btree
```

would be declared in HOL as

```
Hol_datatype `btree = Leaf of 'a
                    | Node of btree => 'b => btree`
```

The => notation in a HOL datatype description is intended to replace $*$ in an ML datatype description, and highlights the fact that, in HOL, constructors are by default curried. Note also that any type parameters for the new type are not mentioned: the type variables are always ordered alphabetically.

This subtle point bears repeating: the format of datatype definitions does not have enough information to always determine the order of arguments to the introduced type operators. Thus, when defining a type that is polymorphic in more than one argument, there is a question of what the order of the new operator's arguments will be. For another example, if one defines

```
Hol_datatype `sum = Left of 'left | Right of 'right`;
```

and then writes `('a,'b)sum`, will the `'a` value be under the `Left` or `Right` constructor? The system chooses to make the arguments corresponding to variables appear in the order given by the dictionary ordering of the variables' names. Thus, in the example given, the `'a` of `('a,'b)sum` will be the `Left` argument because `left` comes before `right` in the standard (ASCII) dictionary ordering.

4.1.1 Further examples

In the following, we shall give an overview of the kinds of types that may be defined by `Hol_datatype`.

To start, enumerated types can be defined as in the following example:

```
Hol_datatype
  ‘enum = A1  | A2  | A3  | A4  | A5
        | A6  | A7  | A8  | A9  | A10
        | A11 | A12 | A13 | A14 | A15
        | A16 | A17 | A18 | A19 | A20
        | A21 | A22 | A23 | A24 | A25
        | A26 | A27 | A28 | A29 | A30‘
```

Other non-recursive types may be defined as well:

```
Hol_datatype
  ‘foo = N of num
       | B of bool
       | Fn of ’a -> ’b
       | Pr of ’a # ’b‘
```

Turning to recursive types, we have already seen a type of binary trees having polymorphic values at internal nodes. This time, we will declare it in “paired” format.

```
Hol_datatype
  ‘tree = Leaf of ’a
        | Node of tree # ’b # tree‘
```

This specification seems closer to the declaration that one might make in ML, but can be more difficult to deal with in proof than the curried format used above.

The basic syntax of the named lambda calculus is easy to describe:

```
Hol_datatype
  ‘lambda = Var of string
          | Const of ’a
          | Comb of lambda => lambda
          | Abs of lambda => lambda‘
```

The syntax for ‘de Bruijn’ terms is roughly similar:

```
Hol_datatype
  ‘dB = Var of string
      | Const of ’a
      | Bound of num
      | Comb  of dB => dB
      | Abs   of dB‘
```

Arbitrarily branching trees may be defined by allowing a node to hold the list of its subtrees. In such a case, leaf nodes do not need to be explicitly declared.

```
Hol_datatype
  `ntree = Node of 'a => ntree list`
```

A type of 'first order terms' can be declared as follows:

```
Hol_datatype
  `term = Var of string
        | Fnapp of string # term list`
```

Mutally recursive types may also be defined. The following, extracted by Elsa Gunter from the Definition of Standard ML, captures a subset of Core ML.

```
Hol_datatype
  `atexp = var_exp of string
         | let_exp of dec => exp ;

     exp = aexp    of atexp
         | app_exp of exp => atexp
         | fn_exp  of match ;

   match = match  of rule
         | matchl of rule => match ;

    rule = rule of pat => exp ;

     dec = val_dec   of valbind
         | local_dec of dec => dec
         | seq_dec   of dec => dec ;

 valbind = bind  of pat => exp
         | bindl of pat => exp => valbind
         | rec_bind of valbind ;

     pat = wild_pat
         | var_pat of string`
```

Simple record types may be introduced using the <| ... |> notation.

```
Hol_datatype
  `state = <| Reg1 : num; Reg2 : num; Waiting : bool |>`
```

The use of record types may be recursive. For example, the following declaration could be used to formalize a simple file system.

```
Hol_datatype
  `file = Text of string | Dir of directory
     ;
   directory = <| owner : string ;
                  files : (string # file) list |>`
```

4.1.2 Type definitions that fail

Now we address some types that cannot be declared with `Hol_datatype`. In some cases they cannot exist in HOL at all; in others, the type can be built in the HOL logic, but `Hol_datatype` is not able to make the definition.

First, an empty type is not allowed in HOL, so the following attempt is doomed to fail.

```
Hol_datatype
  ‘foo = A of foo‘
```

So called ‘nested types’, which are occasionally quite useful, cannot at present be built with `Hol_datatype`:

```
Hol_datatype
  ‘btree = Leaf of ’a
         | Node of  (’a # ’a) btree‘
```

Types may not recurse on either side of function arrows. Recursion on the right is consistent (see the theory `inftree`), but `Hol_datatype` is not capable of defining algebraic types that require it. Thus, examples such as the following will fail:

```
Hol_datatype
  ‘flist = Nil
         | Cons of ’a => (’b -> flist)‘
```

Recursion on the left must fail for for cardinality reasons. For example, HOL does not allow the following attempt to model the untyped lambda calculus as a set (note the `->` in the clause for the `Abs` constructor):

```
Hol_datatype
  ‘lambda = Var of string
          | Const of ’a
          | Comb of lambda => lambda
          | Abs of lambda -> lambda‘
```

4.1.3 Theorems arising from a datatype definition

The consequences of an invocation of `Hol_datatype` are stored in the current theory segment and in `TypeBase`. The principal consequences of a datatype definition are the primitive recursion and induction theorems. These provide the ability to define simple functions over the type, and an induction principle for the type. Thus, for a type named `ty`, the primitive recursion theorem is stored under `ty_Axiom` and the induction theorem is put under `ty_induction`. Other consequences include the distinctness of constructors (`ty_distinct`), and the injectivity of constructors (`ty_11`). A ‘degenerate’ version of `ty_induction` is also stored under `ty_nchotomy`: it provides for reasoning by

cases on the construction of elements of `ty`. Finally, some special-purpose theorems are stored: for example, `ty_case_cong` holds a congruence theorem for "case" statements on elements of `ty`. These case statements are defined by `ty_case_def`. Also, a definition of the "size" of the type is added to the current theory, under the name `ty_size_def`.

For example, invoking

```
Hol_datatype
  `tree = Leaf of num
        | Node of tree => tree`
```

results in the definitions

```
tree_case_def =
  |- (!f f1 a. case f f1 (Leaf a) = f a) /\
     !f f1 a0 a1. case f f1 (Node a0 a1) = f1 a0 a1

tree_size_def
  |- (!a. tree_size (Leaf a) = 1 + a) /\
     !a0 a1. tree_size (Node a0 a1) = 1 + (tree_size a0 + tree_size a1)
```

being added to the current theory. The following theorems about the datatype are also proved and stored in the current theory.

```
tree_Axiom
  |- !f0 f1.
     ?fn. (!a. fn (Leaf a) = f0 a) /\
          !a0 a1. fn (Node a0 a1) = f1 a0 a1 (fn a0) (fn a1)
tree_induction
  |- !P. (!n. P (Leaf n)) /\
         (!t t0. P t /\ P t0 ==> P (Node t t0)) ==> !t. P t
tree_nchotomy
  |- !t. (?n. t = Leaf n) \/ ?t' t0. t = Node t' t0
tree_11
  |- (!a a'. (Leaf a = Leaf a') = (a = a')) /\
     !a0 a1 a0' a1'. (Node a0 a1 = Node a0' a1') = (a0=a0') /\ (a1=a1')
tree_distinct
  |- !a1 a0 a. ~(Leaf a = Node a0 a1)
tree_case_cong
  |- !M M' f f1.
      (M = M') /\
      (!a. (M' = Leaf a) ==> (f a = f' a)) /\
      (!a0 a1. (M' = Node a0 a1) ==> (f1 a0 a1 = f1' a0 a1))
        ==>
      (case f f1 M = case f' f1' M')
```

When a type involving records is defined, many more definitions are made and added to the current theory.

A mutually recursive type definition results in the above theorems and definitions being added for each of the defined types.

4.2 Record Types

Record types are convenient ways of bundling together a number of component types, and giving those components names so as to facilitate access to them. Record types are semantically equivalent to big pair (product) types, but the ability to label the fields with names of one's own choosing is a great convenience. Record types as implemented in HOL are similar to C's `struct` types and to Pascal's records.

Done correctly, record types provide useful maintainability features. If one can always access the `fieldn` field of a record type by simply writing `record.fieldn`, then changes to the type that result in the addition or deletion of other fields will not invalidate this reference. One failing in SML's record types is that they do not allow the same maintainability as far as (functional) updates of records are concerned. The HOL implementation allows one to write

```
rec with fieldn := new_value
```

which replaces the old value of `fieldn` in the record `rec` with `new_value`. This expression will not need to be changed if another field is added, modified or deleted from the record's original definition.

Defining a record type Record types are defined with the function `Hol_datatype`, as previously discussed. For example, to create a record type called `person` with boolean, string and number fields called `employed`, `name` and `age`, one would enter:

```
Hol_datatype
  `person = <| employed : bool ;
               age : num ;
               name : string |>`
```

The order in which the fields are entered is not significant. As well as defining the type (called `person`), the datatype definition function also defines two other sets of constants. These are the field access functions and functional update functions. The field access functions have names of the form ⟨*record-type*⟩_⟨*field*⟩. These functions can be used directly, or one can use standard field selection notation to access the values of a record's field. Thus, one would write the expression: `bob.employed` in order to return the value of `bob`'s `employed` field. The alternative, `person_employed bob`, works, but would be printed using the first syntax, with the full-stop.

The functional update functions are given the names "⟨*record-type*⟩_⟨*field*⟩_`fupd`" for each field in the type. They take two arguments, a function and a record to be updated. The function parameter is an endomorphism on the field type, so that the resulting record is the same as the original, except that the specified field has had the given function applied to it to generate the new value for that field. They can be written with the keyword `with` and the `updated_by` operator. Thus

```
bob with employed updated_by $~
```

is a record value identical to the `bob` except that the boolean value in the `employed` field has been inverted.

Additionally, there is syntactic sugar available to let one write a record with one of its fields replaced by a specific value. This is done by using the `:=` operator instead of `updated_by`:

```
bob with employed := T
```

This form is translated at parse-time to be a use of the corresponding functional update, along with a use of the K-combinator from the `combin` theory. Thus, the above example is really

```
bob with employed updated_by (K T)
```

which is in turn a pretty form of

```
person_employed_fupd (K T) bob
```

If a chain of updates is desired, then multiple updates can be specified inside `<|-|>` pairs, separated by semi-colons, thus:

```
bob with <| age := 10; name := "Child labourer" |>
```

Both update forms (using `updated_by` and `:=`) can be used in a chain of updates.

Specifying record literals The parser accepts lists of field specifications between `<|-|>` pairs without the `with` keyword. These translate to sequences of updates of an arbitrary value (literally, the HOL value `ARB`), and are treated as literals. Thus,

```
<| age := 21; employed := F; name := "Layabout" |>
```

Using the theorems produced by record definition As well as defining the type and the functions described above, record type definition also proves a suite of useful theorems. These are all are saved (using `save_thm`) in the current segment. Some are also added to the `TypeBase`'s simplifications for the type, so they will be automatically applied when simplifying with the `srw_ss()` simpset, or with the tactics `RW_TAC` and `SRW_TAC` (see Section 5.5).

All of the theorems are saved under names that begin with the name of the type. The list below is a sample of the theorems proved. The identifying strings are suffixes appended to the name of the type in order to generate the final name of the theorem.

`_accessors` The definitions of the accessor functions. This theorem is installed in the `TypeBase`.

_fn_updates The definitions of the functional update functions.

_accfupds A theorem that states simpler forms for expressions that are of the form $\mathit{field}_i\ (\mathit{field}_j\texttt{\_fupd}\ f\ r)$. If $i = j$, then the RHS is $f(\mathit{field}_i(r))$, if not, it is $(\mathit{field}_i\ r)$. This theorem is installed in the TypeBase.

_component_equality A theorem stating that $(r_1 = r_2) \equiv \bigwedge_i(\mathit{field}_i(r_1) = \mathit{field}_i(r_2))$.

_fupdfupds A thereom stating that $\mathit{field}_i\texttt{\_fupd}\ f\ (\mathit{field}_i\texttt{\_fupd}\ g\ r) = \mathit{field}_i\texttt{\_fupd}\ (f \circ g)\ r$. This theorem is installed in the TypeBase.

_fupdcanon A theorem that states commutativity results for all possible pairs of field updates. They are constructed in such a way that if used as rewrites, they will canonicalise sequences of updates. Thus, for all $i < j$,

$$\mathit{field}_j\texttt{\_fupd}\ f\ (\mathit{field}_i\texttt{\_fupd}\ g\ r) = \mathit{field}_i\texttt{\_fupd}\ g\ (\mathit{field}_j\texttt{\_fupd}\ f\ r)$$

is generated. This theorem is installed in the TypeBase.

Big records The size of certain theorems proved in the record type package increases as the square of the number of fields in the record. (In particular, the update canonicalisation and acc_fupd theorems have this property.) To avoid inefficiency with big records, the implementation of record types uses a more efficient underlying representation when the number of fields grows too large. The exact point at which this optimisation is applied is controlled by the reference variable Datatype.big_record_size. This value is initialised to 20, but users can change it as they choose.

Unfortunately, the big record representation has the drawback that every update and accessor function has two forms: different terms that are printed the same. One form is a simple constant, and is the form produced when a term is parsed. The other is more complicated, but allows for the use of smaller theorems when record values are simplified. Therefore, it is recommended that new, user-proved theorems that mention big records' fields or field updates be passed through a phase of simplification (SIMP_RULE), applying the TypeBase's rewrites, before they are saved.

The pretty-printing of big records can be controlled with the pp_bigrecs trace-flag.

4.3 Quotient Types

HOL provides a library for defining new types which are quotients of existing types, with respect to partial equivalence relations. This library is described in *"Higher Order Quotients in Higher Order Logic"* [HOQ], from which the following description is taken.

The quotient library is accessed by opening `quotientLib`, which makes all its tools and theorems accessable.

The definition of new types corresponding to the quotients of existing types by equivalence relations is called "lifting" the types from a lower, more representational level to a higher, more abstract level. Both levels describe similar objects, but some details which are apparent at the lower level are no longer visible at the higher level. The logic is simplified.

Simply forming a new type does not complete the quotient operation. Rather, one wishes to recreate the pre-existing logical environment at the new, higher, and more abstract level. This includes not only the new types, but also new versions of the constants that form and manipulate values of those types, and also new versions of the theorems that describe properties of those constants. All of these form a logical layer, above which all the lower representational details may be safely and forever forgotten.

This can be done in a single call of the main tool of this package.

```
define_quotient_types :
        {types: {name: string,
                 equiv: thm} list,
         defs: {def_name: string,
                fname: string,
                func: Term.term,
                fixity: Parse.fixity} list,
         tyop_equivs : thm list,
         tyop_quotients : thm list,
         tyop_simps : thm list,
         respects : thm list,
         poly_preserves : thm list,
         poly_respects : thm list,
         old_thms : thm list} ->
        thm list
```

`define_quotient_types` takes a single argument which is a record with the following fields.

types is a list of records, each of which contains two fields: *name*, which is the name of a new quotient type to be created, and *equiv*, which is either 1) a theorem that a binary relation *R* is an equivalence relation (see [HOQ] §4) of the form

$$\vdash \forall x\ y.\ R\ x\ y\ \Leftrightarrow\ (R\ x = R\ y),$$

or 2) a theorem that *R* is a nonempty partial equivalence relation, (see [HOQ] §5) of the form

$$\vdash (\exists x.\ R\ x\ x)\ \wedge\ (\forall x\ y.\ R\ x\ y\ \Leftrightarrow\ R\ x\ x \wedge R\ y\ y \wedge (R\ x = R\ y)).$$

The process of forming the new quotient types is described in [HOQ] §8.

defs is a list of records specifying the constants to be lifted. Each record contains the following four fields: *func* is an HOL term, which must be a single constant, which is the constant to be lifted. *fname* is the name of the new constant being defined as the lifted version of *func*. *fixity* is the HOL fixity of the new constant being created, as specified in the HOL structure `Parse`. *def_name* is the name under which the new constant definition is to be stored in the current theory. The process of defining lifted constants is described in [HOQ] §9.

tyop_equivs is a list of conditional equivalence theorems for type operators (see [HOQ] §4.1). These are used for bringing into regular form theorems on new type operators, so that they can be lifted (see [HOQ] §11 and §12).

tyop_quotients is a list of conditional quotient theorems for type operators (see [HOQ] §5.2). These are used for lifting both constants and theorems.

tyop_simps is a list of theorems used to simplify type operator relations and map functions, e.g., for pairs, `|- ($= ### $=) = $=` and `|- (I ## I) = I`.

The rest of the arguments refer to the general process of lifting theorems over the quotients being defined, as described in [HOQ] §10.

respects is a list of theorems about the respectfulness of the constants being lifted. These theorems are described in [HOQ] §10.1.

poly_preserves is a list of theorems about the preservation of polymorphic constants in the HOL logic across a quotient operation. In other words, they state that any quotient operation preserves these constants as a homomorphism. These theorems are described in [HOQ] §10.2.

poly_respects is a list of theorems showing the respectfulness of the polymorphic constants mentioned in *poly_preserves*. These are described in [HOQ] §10.3.

old_thms is a list of theorems concerning the lower, representative types and contants, which are to be automatically lifted and proved at the higher, more abstract quotient level. These theorems are described in [HOQ] §10.4.

`define_quotient_types` returns a list of theorems, which are the lifted versions of the *old_thms*.

A similar function, `define_quotient_types_rule`, takes a single argument which is a record with the same fields as above except for *old_thms*, and returns an SML function of type `thm -> thm`. This result, typically called `LIFT_RULE`, is then used to lift the old theorems individually, one at a time.

For backwards compatibility with the excellent quotients package `EquivType` created by John Harrison (which provided much inspiration), the following function is also provided:

```
define_equivalence_type :
        {name: string,
         equiv: thm,
         defs: {def_name: string,
                fname: string,
                func: Term.term,
                fixity: Parse.fixity} list,
         welldefs : thm list,
         old_thms : thm list} ->
        thm list
```

This function is limited to a single quotient type, but may be more convenient when the generality of `define_quotient_types` is not needed. This function is defined in terms of `define_quotient_types` as

```
fun define_equivalence_type {name,equiv,defs,welldefs,old_thms} =
    define_quotient_types
     {types=[{name=name, equiv=equiv}], defs=defs, tyop_equivs=[],
      tyop_quotients=[FUN_QUOTIENT],
      tyop_simps=[FUN_REL_EQ,FUN_MAP_I], respects=welldefs,
      poly_preserves=[FORALL_PRS,EXISTS_PRS],
      poly_respects=[RES_FORALL_RSP,RES_EXISTS_RSP],
      old_thms=old_thms};
```

4.4 Case Expressions

Within the HOL logic, case expressions provide a very compact and convenient notation for multi-way selection among the values of several expressions. This is modeled on the case constructs in functional programming languages such as Standard ML. Such case expressions can simplify the expression of complicated branches between different cases or combinations of cases. The basic syntax (where the non-terminal $term$ stands for any HOL term) is

$$
\begin{array}{rcl}
term & ::= & \texttt{case}\ term\ \texttt{of}\ cases \\
cases & ::= & case_1\ morecases \\
case_1 & ::= & \texttt{|}\ case \ \mid\ case \\
morecases & ::= & \varepsilon \ \mid\ \texttt{|}\ case\ morecases \\
case & ::= & term\ \texttt{=>}\ term
\end{array}
$$

The choice in the rule for the first case ($case_1$) allows the use of more uniform syntax, where every case is preceded by a vertical bar. Omitting the bar, which is what the pretty-printer does when the syntax is printed, conforms with the syntax used by SML.

Based on the value of a test expression, a list of pattern expressions are considered in sequence to see if they match the test expression. The first pattern which successfully matches causes its associated result expression to be evaluated and its value yielded as the value of the entire case expression. For example,

```
case n of
   0 => "none"
 | 1 => "one"
 | 2 => "two"
 | _ => "many"
```

This could have been expressed using several "if–then–else" constructs, but the case expression is much more compact and clean, with the selection between various choices made clearly evident.

In addition to literals as patterns, as above, patterns may be constructor expressions. Many standard HOL types have constructors, including `num`, `list`, and `option`.

```
case spouse(employee) of
 | NONE   => "single"
 | SOME s => "married to " ++ name_of s
```

(This example uses the optional bar in front of the first case.)

HOL supports a rich structure of case expressions using a single notation. The format is related to that of definitions of recursive functions, as described in Section 4.5. In addition, case expressions may contain literals as patterns, either singly or as elements of deeply nested patterns.

Case expressions may test values of any type. If the test expression is a type with constructors, then the patterns may be expressed using the constructors applied to arguments, as for example `SOME s` in the example above. A free variable within the constructor pattern, for example `s` in the pattern `SOME s`, becomes bound to the corresponding value within the value of the test expression, and can be used within the associated result expression for that pattern.

In addition to the constructors of standard types in HOL, constructor patterns may also be used for types created by use of the datatype definition facility described in Section 4.1, including user-defined types.

Whether or not the test expression is a type with constructors, the patterns may be expressed using the appropriate literals of that type, if any such literals exist. A complex pattern may contain either or both of literals and constructor patterns nested within it. However, literals and constructors may not be mixed as alternatives of each other within the same case expression, except insofar as a particular pattern may be both a literal and also a (0-ary) constructor of its type, as for example `0` (zero) is both a literal and a constructor of the type `num`. Here is an example of this kind of improper mixture.

```
case n of
   0 => "none"
 | 1 => "one"
 | 2 => "two"
 | SUC m => "many"
```

In this pattern, the constructor pattern `SUC m` is given as an alternative to the literal patterns `1` and `2`. This makes this attempted case expression invalid. Deleting either group of rows would resolve the conflict, and make the expression valid. Note that the pattern `0` is acceptable to either group.

Patterns can be nested as well, as shown in the next example, where the function `parents` returns a pair containing the person's father and/or mother, where each is represented by `NONE` if deceased.

```
case parents(john) of
   (NONE,NONE) => "orphan"
 | _ => "not an orphan"
```

This shows the nesting of option patterns within a pair pattern, and also the use of a wildcard _ to match the cases not given.

If the set of patterns is sparse, there may be several new rows generated automatically to fill it out, and possibly some new variables or the `ARB` constant to properly represent the case expression.

```
- ``case a of
     (1, y, z) => y + z
   | (x, 2, z) => x - z
   | (x, y, 3) => x * y``;
> val it =
   ``case a of
       (1,2,3) => 2 + 3
     | (1,2,z) => 2 + z
     | (1,y,3) => y + 3
     | (1,y,z) => y + z
     | (x,2,3) => x - 3
     | (x,2,z') => x - z'
     | (x,y',3) => x * y'
     | (x,y',z') => ARB`` : term
```

This is just a brief description of some of the expressive capabilities of the case expression with patterns. Many more examples of patterns are provided in Section 4.5 on the definition of recursive functions.

4.5 Recursive Functions

HOL provides a function definition mechanism based on the wellfounded recursion theorem proved in `relationTheory`, discussed in Section 3.5.3. `Define` takes a high-level,

possibly recursive, specification of a function, and attempts to define the function in the logic. `Define` may be used to define abbreviations, recursive functions, and mutually recursive functions. An induction theorem may be generated as a by-product of `Define`'s activity. This induction theorem follows the recursion structure of the function, and may be useful when proving properties of the function. `Define` is not always successful in attempting to make the specified definition, usually because an automatic termination proof fails; in that case, another entrypoint, `Hol_defn`, which defers the termination proof to the user, can be used. The technology underlying `Define` and `Hol_defn` is explained in detail in Slind [10].

In particular, `Define` takes as input a quotation representing a conjunction of equations. The specified function(s) may be phrased using ML-style pattern-matching. A call `Define` ‘*spec*‘ should conform with the grammar in Table 4.2.

$$
\begin{array}{rcl}
spec & ::= & eqn \\
 & | & (eqn) \wedge spec \\
 & & \\
eqn & ::= & alphanumeric\ pat \ldots pat = term \\
 & & \\
 & & \\
pat & ::= & variable \\
 & | & wildcard \\
 & | & cname \\
 & | & (cname_n\ pat_1 \ldots pat_n) \\
 & & \\
cname & ::= & alphanumeric \mid symbolic \\
 & & \\
wildcard & ::= & \_ \\
 & | & \_wildcard
\end{array}
$$

Table 4.2: Syntax of Function Declaration

Pattern Expansion In general, `Define` attempts to derive exactly the specified conjunction of equations. However, the rich syntax of patterns allows some ambiguity. For example, the input

```
Define ‘(f 0 _ = 1)
  /\    (f _ 0 = 2)‘
```

is ambiguous at `f 0 0`: should the result be `1` or `2`? This ambiguity is dealt with in the usual way for compilers and interpreters for functional languages: namely, the conjunction of equations is treated as being applied left-conjunct first, followed by processing

the right conjunct. Therefore, in the example above, the value of `f 0 0` is `1`. In the implementation, ambiguities arising from such overlapping patterns are systematically translated away in a pre-processing step.

Another case of ambiguity in patterns is shown above: the specification is incomplete since it does not tell how `f` should behave when applied to two non-zero arguments: e.g., `f (SUC m) (SUC n)`. In the implementation, such missing clauses are filled in, and have the value `ARB`. This 'pattern-completion' step is a way of turning descriptions of partial functions into total functions suitable for HOL. However, since the user has not completely specified the function, the system takes that as a hint that the user is not interested in using the function at the missing-but-filled-in clauses, and so such clauses are dropped from the final theorem.

In summary, `Define` will derive the unambiguous and complete equations

```
|- (f 0 (SUC v4) = 1) /\
   (f 0 0 = 1) /\
   (f (SUC v2) 0 = 2)
   (f (SUC v2) (SUC v4) = ARB)
```

from the above ambiguous and incomplete equations. The odd-looking variable names are due to the pre-processing steps described above. The above result is only an intermediate value: in the final result returned by `Define`, the last equation is droppped since it was not specified by the original input.

```
|- (f 0 (SUC v4) = 1) /\
   (f 0 0 = 1) /\
   (f (SUC v2) 0 = 2)
```

Termination When processing the specification of a recursive function, `Define` must perform a termination proof. It automatically constructs termination conditions for the function, and invokes a termination prover in an attempt to prove the termination conditions. If the function is primitive recursive, in the sense that it exactly follows the recursion pattern of a previously declared HOL datatype, then this proof always succeeds, and `Define` stores the derived equations in the current theory segment. Otherwise, the function is not an instance of primitive recursion, and the termination prover may succeed or fail. If the termination proof fails, then `Define` fails. If it succeeds, then `Define` stores the specified equations in the current theory segment. An induction theorem customized for the defined function is also stored in the current segment. Note, however, that an induction theorem is not stored for primitive recursive functions, since that theorem would be identical to the induction theorem resulting from the declaration of the datatype.

Storing definitions in the theory segment `Define` automatically generates names with which to store the definition and, (if it exists) the associated induction theorem,

in the current theory. The name for storing the definition is built by concatenating the name of the function with the value of the reference variable `Defn.def_suffix`. The name for storing the induction theorem is built by concatenating the name of the function with the value of the reference variable `Defn.ind_suffix`. For mutually recursive functions, where there is a choice of names, the name of the function in the first clause is taken.

Since the names used to store elements in the current theory segment are transformed into ML bindings after the theory is exported, it is required that every invocation of `Define` generate names that are valid ML identifiers. For this reason, `Define` requires alphanumeric function names. If one wishes to define symbolic identifiers, the ML function `xDefine` should be used.

```
xDefine : string -> term quotation -> thm
```

The `xDefine` function is identical to `Define` except that it takes an explicit name to use when storing the definition in the current theory.

4.5.1 Function definition examples

We will give a number of examples that display the range of functions that may be defined with `Define`. First, we have a recursive function that uses "destructors" in the recursive call.

```
Define
  ‘fact x = if x = 0 then 1 else x * fact(x-1)‘;

Equations stored under "fact_def".
Induction stored under "fact_ind".
> val it = |- fact x = (if x = 0 then 1 else x * fact (x - 1)) : thm
```

Since `fact` is not primitive recursive, an induction theorem for `fact` is generated and stored in the current theory.

```
- DB.fetch "-" "fact_ind";

> val it =
   |- !P. (!x. (~(x = 0) ==> P (x - 1)) ==> P x) ==> !v. P v : thm
```

Next we have a recursive function with relatively complex pattern-matching. We omit to examine the generated induction theorem.

```
Define `(flatten  []           = [])
   /\   (flatten ([]::rst)     = flatten rst)
   /\   (flatten ((h::t)::rst) = h::flatten(t::rst))`;

Equations stored under "flatten_def".
Induction stored under "flatten_ind".

> val it =
    |- (flatten [] = []) /\
       (flatten ([]::rst) = flatten rst) /\
       (flatten ((h::t)::rst) = h::flatten (t::rst)) : thm
```

Next we define a curried recursive function, which uses wildcard expansion and pattern-matching pre-processing.

```
Define `(min (SUC x) (SUC y) = min x y + 1)
   /\   (min  ____    ____   = 0)`;

Equations stored under "min_def".
Induction stored under "min_ind".

> val it =
    |- (min (SUC x) (SUC y) = min x y + 1) /\
       (min (SUC v2) 0 = 0) /\
       (min 0 v1 = 0) : thm
```

Next we make a primitive recursive definition. Note that no induction theorem is generated in this case.

```
Define `(filter P [] = [])
  /\    (filter P (h::t) = if P h then h::filter P t else filter P t)`;

Definition has been stored under "filter_def".

> val it =
   |- (!P. filter P [] = []) /\
      !P h t. filter P (h::t) =
                (if P h then h::filter P t else filter P t) : thm
```

`Define` may also be used to define mutually recursive functions. For example, we can define a datatype of propositions and a function for putting a proposition into negation normal form as follows. First we define a datatype, named `prop`, of boolean formulas:

```
Hol_datatype
  `prop = VAR of 'a
        | NOT of prop
        | AND of prop => prop
        | OR  of prop => prop`;
```

Then two mutually recursive functions `nnfpos` and `nnfneg` are defined:

```
Define
   `(nnfpos (VAR x)   = VAR x)
 /\ (nnfpos (NOT p)   = nnfneg p)
 /\ (nnfpos (AND p q) = AND (nnfpos p) (nnfpos q))
 /\ (nnfpos (OR p q)  = OR  (nnfpos p) (nnfpos q))

 /\ (nnfneg (VAR x)   = NOT (VAR x))
 /\ (nnfneg (NOT p)   = nnfpos p)
 /\ (nnfneg (AND p q) = OR  (nnfneg p) (nnfneg q))
 /\ (nnfneg (OR p q)  = AND (nnfneg p) (nnfneg q))`
```

The system makes the definition and returns the theorem

```
|- (nnfpos (VAR x) = VAR x) /\
   (nnfpos (NOT p) = nnfneg p) /\
   (nnfpos (AND p q) = AND (nnfpos p) (nnfpos q)) /\
   (nnfpos (OR p q) = OR (nnfpos p) (nnfpos q)) /\
   (nnfneg (VAR x) = NOT (VAR x)) /\
   (nnfneg (NOT p) = nnfpos p) /\
   (nnfneg (AND p q) = OR (nnfneg p) (nnfneg q)) /\
   (nnfneg (OR p q) = AND (nnfneg p) (nnfneg q)) : thm
```

`Define` may also be used to define non-recursive functions.

```
Define
  `f x (y,z) = (x + 1 = y DIV z)`;
```

`Define` may also be used to define non-recursive functions with complex pattern-matching. The pattern-matching pre-processing of Define can be convenient for this purpose, but can also generate a large number of equations. For example:

```
Define
  `(g (0,_,_,_,_) = 1) /\
   (g (_,0,_,_,_) = 2) /\
   (g (_,_,0,_,_) = 3) /\
   (g (_,_,_,0,_) = 4) /\
   (g (_,_,_,_,0) = 5)`
```

yields a definition with thirty-one clauses.

4.5.2 When termination is not automatically proved

If the termination proof for a prospective definition fails, the invocation of `Define` (or `xDefine`) fails. In such situations, the ML function `Hol_defn` should be used.

```
Hol_defn : string -> term quotation -> Defn.defn
```

`Hol_defn` makes the requested definition, but defers the proof of termination to the user. For setting up termination proofs, there are several useful entrypoints, namely

```
Defn.tgoal  : Defn.defn -> GoalstackPure.proofs
Defn.tprove : Defn.defn * tactic -> thm * thm
```

`Defn.tgoal` is analogous to `set_goal` and `Defn.tprove` is analogous to `prove`. Thus, `Defn.tgoal` is used to take the result of `Hol_defn` and set up a goal for proving termination of the definition.

Example. An invocation of `Define` on the following equations for Quicksort will currently fail, since the termination proof is currently beyond the capabilities of the naive termination prover. Instead, we make an application of `Hol_defn`:

1
```
val qsort_def =
 Hol_defn "qsort"
   `(qsort ord [] = []) /\
    (qsort ord (h::t) =
        qsort ord (FILTER (\x. ord x h) t)
        ++ [h] ++
        qsort ord (FILTER (\x. ~(ord x h)) t))`
```

which returns the following value of type `defn`, but does not try to prove termination.

2
```
HOL function definition (recursive)

Equation(s) :
 [...] |- qsort ord [] = []
 [...]
|- qsort ord (h::t) =
   qsort ord (FILTER (\x. ord x h) t) ++ [h] ++
   qsort ord (FILTER (\x. ~ord x h) t)

Induction :
 [...]
|- !P.
     (!ord. P ord []) /\
     (!ord h t.
        P ord (FILTER (\x. ~ord x h) t) /\
        P ord (FILTER (\x. ord x h) t) ==>
        P ord (h::t)) ==>
     !v v1. P v v1

Termination conditions :
  0. !t h ord. R (ord,FILTER (\x. ~ord x h) t) (ord,h::t)
  1. !t h ord. R (ord,FILTER (\x. ord x h) t) (ord,h::t)
  2. WF R
```

The type `defn` has a prettyprinter installed for it: the above output is typical, showing the components of a `defn` in an understandable format. Although it is possible to directly work with elements of type `defn`, it is more convenient to invoke `Defn.tgoal`, which sets up a termination proof in a goalstack.

3

```
Defn.tgoal qsort_def;

> val it =
  Proof manager status: 1 proof.
  1. Incomplete:
       Initial goal:
       ?R.
         (!t h ord. R (ord,FILTER (\x. ~ord x h) t) (ord,h::t)) /\
         (!t h ord. R (ord,FILTER (\x. ord x h) t) (ord,h::t)) /\ WF R
```

The goal is to find a wellfounded relation on the arguments to `qsort` and show that the arguments to `qsort` are in the relation. The function `WF_REL_TAC` is almost invariably used at this point to initiate the termination proof. Clearly, `qsort` terminates because the list argument gets shorter. Invoking `WF_REL_TAC` with the appropriate measure function results in two subgoals, both of which are easy to prove.

4

```
- e (WF_REL_TAC `measure (LENGTH o SND)`);
OK..
2 subgoals:
> val it =
   !t h ord. LENGTH (FILTER (\x. ord x h) t) < LENGTH (h::t)

   !t h ord. LENGTH (FILTER (\x. ~ord x h) t) < LENGTH (h::t)
```

Execution of `WF_REL_TAC` has automatically proved the wellfoundedness of the termination relation `measure (LENGTH o SND)` and the remainder of the goal has been simplified into a pair of easy goals. Once both goals are proved, we can encapsulate the termination proof with `tDefine`, which takes a quotation (representing desired recursion equations) and a tactic t, defines the specified function, calculates the termination conditions, and applies t to them. If the termination conditions are proved by t then the recursion equations and induction theorem are stored in the current theory segment before the recursion equations are returned:

5

```
- val qsort_def =  tDefine "qsort"
   `(qsort ord [] = []) /\
    (qsort ord (h::t) =
        qsort ord (FILTER (\x. ord x h) t) ++ [h] ++
        qsort ord (FILTER (\x. ~(ord x h)) t))`
   (WF_REL_TAC `measure (LENGTH o SND)` THEN ...);

> val qsort_def =
    |- (qsort ord [] = []) /\
       (qsort ord (h::t) =
          qsort ord (FILTER (\x. ord x h) t) ++ [h] ++
          qsort ord (FILTER (\x. ~ord x h) t)) : thm
```

The custom induction theorem for a function can be obtained by using `fetch`, which returns named elements in the specified theory.[1]

```
- fetch "-" "qsort_ind";                                                    6
> val qsort_ind =
    |- !P.
         (!ord. P ord []) /\
         (!ord h t.
            P ord (FILTER (\x. ~ord x h) t) /\
            P ord (FILTER (\x. ord x h) t) ==> P ord (h::t))
          ==>
         !v v1. P v v1  : thm
```

The induction theorem produced by `Define` and `tDefine` can be applied by `recInduct`. See Section 5.3 for details.

4.5.2.1 Techniques for proving termination

There are two problems to deal with when trying to prove termination. First, one has to understand, intuitively and then mathematically, why the function under consideration terminates. Second, one must be able to phrase this in HOL. In the following, we shall give a few examples of how this is done.

There are a number of basic and advanced means of specifying wellfounded relations. The most common starting point for dealing with termination problems for recursive functions is to find some function, known as a a *measure* under which the arguments of a function call are larger than the arguments to any recursive calls that result.

For a very simple starter example, consider the following definition of a function that computes the greatest common divisor of two numbers:

```
- val gcd_defn =                                                            1
    Hol_defn "gcd"
       `(gcd (0,n) = n) /\
        (gcd (m,n) = gcd (n MOD m, m))`;

- Defn.tgoal gcd_defn;

> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
         Initial goal:
         ?R. WF R /\ !v2 n. R (n MOD SUC v2,SUC v2) (SUC v2,n)
```

The invocation `gcd(m,n)` recurses in its first argument, and since we know that `m` is not 0, it is the case that `n MOD m` is smaller than `m`. The way to phrase the termination of

[1] In a call to `fetch`, the first argument denotes a theory; the current theory may be specified by `"-"`.

gcd in HOL is to use a ‘measure‘ function to map from the domain of gcd—a pair of numbers—to a number. The definition of measure in HOL is equivalent to

```
|- measure f x y = (f x < f y).
```

Now we must pick out the argument position to measure and invoke WF_REL_TAC:

2
```
- e (WF_REL_TAC ‘measure FST‘);
OK..
1 subgoal:
> val it =
   !v2 n. n MOD SUC v2 < SUC v2
```

This goal is easy to prove with a few simple arithmetic facts.

Weighting Functions Sometimes one needs a measure function that is itself recursive. For example, consider a type of binary trees and a function that linearizes trees. The algorithm works by rotating the tree until it gets a Leaf in the left branch, then it recurses into the right branch. At the end of execution the tree has been linearized.

1
```
- Hol_datatype
    ‘btree = Leaf
           | Brh of btree => btree‘;

- val Unbal_defn =
    Hol_defn "Unbal"
      ‘(Unbal Leaf = Leaf)
   /\  (Unbal (Brh Leaf bt) = Brh Leaf (Unbal bt))
   /\  (Unbal (Brh (Brh bt1 bt2) bt) = Unbal (Brh bt1 (Brh bt2 bt)))‘;

- Defn.tgoal Unbal_defn;

> val it =
    Proof manager status: 1 proof.
    1. Incomplete:
       Initial goal:
        ?R. WF R /\
            (!bt. R bt (Brh Leaf bt)) /\
            !bt bt2 bt1. R (Brh bt1 (Brh bt2 bt)) (Brh (Brh bt1 bt2) bt)
```

Since the size of the tree is unchanged in the last clause in the definition of Unbal, a simple size measure will not work. Instead, we can assign weights to nodes in the tree such that the recursive calls of Unbal decrease the total weight in every case. One such assignment is

2
```
Define
 ‘(Weight (Leaf) = 0) /\
  (Weight (Brh x y) = (2 * Weight x) + (Weight y) + 1)‘
```

Now we can invoke WF_REL_TAC:

```
e (WF_REL_TAC ‘measure Weight‘);                                        3
OK..

2 subgoals:
> val it =
 !bt. Weight bt < Weight (Brh Leaf bt)

 !bt bt2 bt1.
    Weight (Brh bt1 (Brh bt2 bt)) < Weight (Brh (Brh bt1 bt2) bt)
```

Both of these goals are quite easy to prove. The technique of ‘weighting‘ nodes in a datatype in order to prove termination also goes by the name of *polynomial interpretation*. It must be admitted that finding the correct weighting for a termination proof is more an art than a science. Typically, one makes a guess and then tries the termination proof to see if it works.

Lexicographic Combinations Occasionally, there's a combination of factors that complicate the termination argument. For example, the following specification describes a naive pattern matching algorithm on strings (represented as lists here). The function takes four arguments: the first, p, is the remainder of the pattern being matched. The second, rst, is the remainder of the string being searched. The third argument, p_0, holds the original pattern to be matched. The fourth argument, s, is the string being searched.

```
val match_defn =                                                        1
  Hol_defn "match"
    ‘(match [] __ __ __ = T)  /\
     (match __ [] __ __ = F)  /\
     (match (a::pp) (b::ss) p0 s =
       if a=b then match pp ss p0 s
         else
       if NULL(s) then F
         else
       match p0 (TL s) p0 (TL s))‘;

- val Match = Define ‘Match pat str = match pat str pat str‘;
```

The first clause of the definition states that if p becomes exhausted, then a match has been found; the function returns T. The second clause represents the case where s becomes exhausted but p is not, in which case the function returns F. The remaining case is when there's more searching to do; the function checks if the head of the pattern p is the same as the head of rst. If yes, then the search proceeds recursively, using the tail of p and the tail of rst. If no, that means that p has failed to match, so the algorithm advances one character ahead in s and starts matching from the beginning of p_0. If s is empty, however, then we return F. Note that rst and s both represent the string being

searched: rst is a 'local' version of s: we recurse into rst as long as there are matches with the pattern p. However, if the search eventually fails, then s, which 'remembers' where the search started from, is used to restart the search.

So much for the behaviour of the function. Why does it terminate? There are two recursive calls. The first call reduces the size of p and rst, and leaves the other arguments unchanged. The second call can increase the size of p and rst, but reduces the size s. This is a classic situation in which to use a lexicographic ordering: some arguments to the function are reduced in some recursive calls, and some others are reduced in other recursive calls. Recall that `LEX` is an infix operator, defined in `pairTheory` as follows:

```
|- LEX R1 R2 = \(x,y) (p,q). R1 x p \/ ((x=p) /\ R2 y q)
```

In the second recursive call, the length of `s` is reduced, and in the first it stays the same. This motivates having the length of the s be the first component of the lexicographic combination, and the length of rst as the second component. Formally, we want to map from the four-tuple of arguments into a lexicographic combination of relations. This is enabled by `inv_image` from `relationTheory`:

```
|- inv_image R f = \x y. R (f x) (f y)
```

The desired relation maps from the four-tuple of arguments into a pair of numbers (m, n), where m is the length of the fourth argument, and n is the length of the second argument. These lengths are then compared lexicographically with respect to less-than ($<$).

2

```
Defn.tgoal match_defn;

- e (WF_REL_TAC `inv_image($< LEX $<) (\(w,x,y,z). (LENGTH z,LENGTH x))`);
OK..
2 subgoals:
> val it =
 !s ss a b.
   (a=b) ==> LENGTH s < LENGTH s \/ LENGTH ss < LENGTH (b::ss)

 !ss s a b.
   ~(a = b) /\ ~NULL s ==>
   LENGTH (TL s) < LENGTH s \/
   (LENGTH (TL s) = LENGTH s) /\ LENGTH (TL s) < LENGTH (b::ss)
```

The first subgoal needs a case-split on `s` before it is proved by rewriting, and the second is also easy to prove by rewriting.

4.5.2.2 How termination conditions are synthesized

It is occasionally important to understand, at least in part, how `Hol_defn` constructs termination constraints. In some cases, it is even necessary for users to influence this

process in order to have correct termination constraints extracted. The process is driven by so-called *congruence theorems* for particular HOL constants. For example, consider the following recursive definition of factorial:

```
fact n = if n=0 then 1 else n * fact (n-1)
```

In the absence of knowledge of how the 'if-then-else' construct affects the *context* of recursive calls, `Hol_defn` would extract the termination constraints:

```
0. WF R
1. !n. R (n - 1) n
```

which are unprovable, because the *context* of the recursive call has not been taken account of. This example is in fact not a problem for HOL, since the following congruence theorem is known to `Hol_defn`:

```
|- !b b' x x' y y'.
     (b = b') /\
     (b' ==> (x = x')) /\
     (~b' ==> (y = y')) ==>
      ((if b then x else y) = (if b' then x' else y'))
```

This theorem is understood by `Hol_defn` as an ordered sequence of instructions to follow when the termination condition extractor hits an 'if-then-else'. The theorem is read as follows: when an instance '`if` B `then` X `else` Y' is encountered while the extractor traverses the function definition, do the following:

1. Traverse B and extract termination conditions $TCs(B)$ from any recursive calls in it. This returns a theorem $TCs(B) \vdash B = B'$.

2. Assume B' and extract termination conditions from any recursive calls in X. This returns a theorem $TCs(X) \vdash X = X'$.

3. Assume $\neg B'$ and extract termination conditions from any recursive calls in Y. This returns a theorem $TCs(Y) \vdash Y = Y'$.

4. By equality reasoning with (1), (2), and (3), derive the theorem

$$TCs(B) \cup TCs(X) \cup TCs(Y) \vdash (\texttt{if } B \texttt{ then } X \texttt{ else } Y) = (\texttt{if } B' \texttt{ then } X' \texttt{ else } Y')$$

5. Replace `if` B `then` X `else` Y by `if` B' `then` X' `else` Y'.

The termination conditions are accumulated until the extraction process finishes, and appear as hypotheses in the final result. Thus the extracted termination conditions for `fact` are

```
0. WF R
1. !n. ~(n = 0) ==> R (n - 1) n
```

and are easy to prove. The notion of *context* of a recursive call is defined by the set of congruence rules used in extracting termination conditions. This set can be obtained by invoking `DefnBase.read_congs`, and manipulated by `DefnBase.add_cong`, `DefnBase.drop_cong` and `DefnBase.export_cong`. The 'add' and 'drop' functions only affect the current state of the congruence database; in contrast, the 'export' function provides a way for theories to specify that a particular theorem should be added to the congruence database in all descendent theories.

Higher Order Recursion and Congruence Rules A 'higher-order' recursion is one in which a higher-order function is used to apply the recursive function to arguments. In order for the correct termination conditions to be proved for such a recursion, congruence rules for the higher order function must be known to the termination condition extraction mechanism. Congruence rules for common higher-order functions, e.g., `MAP`, `EVERY`, and `EXISTS` for lists, are already known to the mechanism. However, at times, one must manually prove and install a congruence theorem for a new user-defined higher-order function.

For example, suppose we define a higher-order function `SIGMA` for summing the results of a function in a list.

1
```
Define `(SIGMA f [] = 0) /\
        (SIGMA f (h::t) = f h + SIGMA f t)`;
```

We then use `SIGMA` in the definition of a function for summing the results of a function in a arbitrarily (finitely) branching tree.

2
```
Hol_datatype `ltree = Node of 'a => ltree list`;

Defn.Hol_defn
  "ltree_sigma"
  `ltree_sigma f (Node v tl) = f v + SIGMA (ltree_sigma f) tl`;
```

In this definition, `SIGMA` is applied to a partial application `(ltree_sigma f)` of the function being defined. Such a situation is called a *higher-order recursion*. Since the recursive call of `ltree_sigma` is not fully applied, special efforts have to be made to extract the correct termination conditions. Otherwise, the following unhappy situation results:

```
HOL function definition (recursive)                                    3

Equation(s) :
  [..] |- ltree_sigma f (Node v tl)
            = f v + SIGMA (\a. ltree_sigma f a) tl

Induction :
  [..] |- !P. (!f v tl. (!a. P f a) ==> P f (Node v tl)) ==> !v v1. P v v1

Termination conditions :
  0. WF R
  1. !tl v f a. R (f,a) (f,Node v tl) : defn
```

The termination conditions for `ltree_sigma` seem to require finding a wellfounded relation `R` such that the pair `(f,a)` is `R`-less than `(f, Node v tl)`. However, this is a hopeless task, since there is no relation between `a` and `Node v tl`, besides the fact that they are both `ltrees`. The termination condition extractor has not performed properly, because it didn't know a congruence rule for `SIGMA`. Such a congruence theorem is the following:

```
SIGMA_CONG =
 |- !l1 l2 f g.
     (l1=l2) /\ (!x. MEM x l2 ==> (f x = g x)) ==>
     (SIGMA f l1 = SIGMA g l2)
```

Once `Hol_defn` has been told about this theorem, via `DefnBase`'s `add_cong` or `export_cong` functions, the termination conditions extracted for the definition are now provable, since `a` is a proper subterm of `Node v tl`.

```
val _ = DefnBase.add_cong SIGMA_CONG;                                  4

Defn.Hol_defn
  "ltree_sigma"
  `ltree_sigma f (Node v tl) = f v + SIGMA (ltree_sigma f) tl`;

> val it =
    HOL function definition (recursive)

    Equation(s) :  ...  (* as before *)
    Induction :    ...  (* as before *)

    Termination conditions :
      0. WF R
      1. !v f tl a. MEM a tl ==> R (f,a) (f,Node v tl)
```

4.5.3 Recursion schemas

In higher order logic, very general patterns of recursion, known as *recursion schemas* or sometimes *program schemas*, can be defined. One example is the following:

$$\mathsf{linRec}(x) = \texttt{if } d(x) \texttt{ then } e(x) \texttt{ else } f(\mathsf{linRec}(g\ x))$$

In this specification, the variables d, e, f, and g are functions, that, when instantiated in different ways, allow linRec to implement different recursive functions. In this, linRec is like many other higher order functions. However, notice that if $d(x) = \mathsf{F}$, $f(x) = x + 1$, and $g(x) = x$, then the resulting instantiation of linRec could be used to obtain a contradiction:

$$\mathsf{linRec}(x) = \mathsf{linRec}(x) + 1$$

This is not, however, derivable in HOL, because recursion schemas are defined by instantiating the wellfounded recursion theorem, and therefore certain abstract termination constraints arise that must be satisfied before recursion equations can be used in an unfettered manner. The entrypoint for defining a schema is `TotalDefn.DefineSchema`. On the linRec example it behaves as follows (note that the schematic variables should only occur on the right-hand side of the definition when making the definition of a schema):

1

```
- TotalDefn.DefineSchema
    `linRec (x:'a) = if d(x) then e(x) else f(linRec(g x))`;

<<HOL message: Definition is schematic in the following variables:
    "d", "e", "f", "g">>

Equations stored under "linRec_def".
Induction stored under "linRec_ind".
> val it =
   [..]
  |- linRec d e f g x = if d x then e x else f (linRec d e f g (g x))
```

The hypotheses of the returned theorem hold the abstract termination constraints. A similarly constrained induction theorem is also stored in the current theory segment.

2

```
hyp it;
> val it = [``!x. ~d x ==> R (g x) x``, ``WF R``] : term list
```

These constraints are abstract, since they place termination requirements on variables that have not yet been instantiated. Once instantiations for the variables are found, then the constraints may be eliminated by finding a suitable wellfounded relation for `R` and then proving the other constraints.

4.6 Inductive Relations

Inductive definitions are made with the function `Hol_reln`, found in the `bossLib` structure, and the resulting definitions and theorems are handled with functions defined in the library `IndDefLib`. The `Hol_reln` function takes a term quotation as input and attempts to define the relations there specified. The input term quotation must parse to a term that conforms to the following grammar:

$$
\begin{array}{rcl}
\langle inputFormat \rangle & ::= & \langle clause \rangle \texttt{ /\ } \langle inputFormat \rangle \mid \langle clause \rangle \\
\langle clause \rangle & ::= & (\texttt{!}\, x_1 \ldots x_n.\ \langle hypothesis \rangle \texttt{ ==> } \langle conclusion \rangle) \\
& \mid & (\texttt{!}\, x_1 \ldots x_n.\ \langle conclusion \rangle) \\
\langle conclusion \rangle & ::= & \langle con \rangle\ sv_1\ sv_2 \ldots \\
\langle hypothesis \rangle & ::= & \text{any term} \\
\langle con \rangle & ::= & \text{a new relation constant}
\end{array}
$$

The (optional) sv_i terms that appear after a constant name are so-called "schematic variables". The same variables must always follow all new constants throughout the definition. These variables and the names of the constants-to-be must not be quantified over in each $\langle clause \rangle$. A $\langle clause \rangle$ should have no other free variables. Any that occur will be universally quantified as part of the process of definition, and a warning message emitted. (Universal quantifiers at the head of the clause can be used to bind free variables, but it is also permissible to use existential quantification in the hypotheses. If a clause has no free variables, it is permissible to have no universal quantification.)

A successful invocation of `Hol_reln` returns three theorems $(rules, ind, cases)$. Each is also stored in the current theory segment.

- $rules$ is a conjunction of implications that will be the same as the input term quotation; the theorem is saved under the name `<stem>_rules`, where `<stem>` is the name of the first relation defined by the function.
- ind is the induction principle for the relations, saved under the name `<stem>_ind`.
- $cases$ is the so-called 'cases' or 'inversion' theorem for the relations, saved under the name `<stem>_cases`. A cases theorem is of the form

```
(!a0 .. an.  R1 a0 .. an = <R1's first rule possibility> \/
                           <R1's second rule possibility> \/ ...)
             /\
(!a0 .. am.  R2 a0 .. am = <R2's first rule possibility> \/
                           <R2's second rule possibility> \/ ...)
             /\
...
```

and is used to decompose an element in the relation into the possible ways of obtaining it by the rules.

If the "stem" of the first constant defined in a set of clauses is such that resulting ML bindings in an exported theory file will result in illegal ML, then the `xHol_reln` function should be used. The `xHol_reln` function is analogous to the `xDefine` function for defining recursive functions (see Section 4.5).

Strong Induction Principles So called "strong" versions of induction principles (in which instances of the relation being defined appear as extra hypotheses), are automatically proved when a definition is made with `Hol_reln`. The strong induction principle for a relation is used when the `Induct_on` tactic is used.

Adding Monotone Operators New constants may occur recursively throughout rules' hypotheses, as long as it can be shown that the rules remain monotone with respect to the new constants. `Hol_reln` automatically attempts to prove such monotonicity results, using a set of theorems held in a reference `IndDefLib.the_monoset`. Monotonicity theorems must be of the form

$$cond_1 \wedge \cdots \wedge cond_m \Rightarrow (Op\ arg_1 \ldots arg_n \Rightarrow Op\ arg'_1 \ldots arg'_n)$$

where each arg and arg' term must be a variable, and where there must be as many $cond_i$ terms as there are arguments to Op that vary. Each $cond_i$ must be of the form

$$\forall \vec{v}.\ arg\ \vec{v} \Rightarrow arg'\ \vec{v}$$

where the vector of variables $\vec{v}$ may be empty, and where the arg and arg' may actually be reversed (as in the rule for negation).

For example, the monotonicity rule for conjunction is

$$(P \Rightarrow P') \wedge (Q \Rightarrow Q') \Rightarrow (P \wedge Q \Rightarrow P' \wedge Q')$$

The monotonicity rule for the `EVERY` operator in the theory of lists (see Section 3.4.1), is

$$(\forall x.\ P(x) \Rightarrow Q(x)) \Rightarrow (\texttt{EVERY}\ P\ \ell \Rightarrow \texttt{EVERY}\ Q\ \ell)$$

With a monotonicity result available for an operator such as `EVERY`, it is then possible to write inductive definitions where hypotheses include mention of the new relation as arguments to the given operators.

Monotonicity results that the user derives may be stored in the global `the_monoset` variable by using the `export_mono` function. This function takes a string naming a theorem in the current theory segment, and adds that theorem to the monotonicity theorems immediately, and in such a way that this situation will also obtain when the current theory is subsequently reloaded.

Examples A simple example of defining two mutually recursive relations is the following:

```
Hol_reln                                                              1
  ‘EVEN 0 /\
   (!n. ODD n ==> EVEN (n + 1)) /\
   (!n. EVEN n ==> ODD (n + 1))‘;
```

The result is three theorems

```
> val it =                                                            2
  (|- EVEN 0 /\
      (!n. ODD n ==> EVEN (n + 1)) /\
      (!n. EVEN n ==> ODD (n + 1)),

   |- !EVEN’ ODD’.
        EVEN’ 0 /\
        (!n. ODD’ n ==> EVEN’ (n + 1)) /\
        (!n. EVEN’ n ==> ODD’ (n + 1))
        ==>
        (!a0. EVEN a0 ==> EVEN’ a0) /\
        (!a1. ODD a1 ==> ODD’ a1),

   |- (!a0. EVEN a0 = (a0 = 0) \/
                      ?n. (a0 = n + 1) /\ ODD n) /\
      (!a1. ODD a1 = ?n. (a1 = n + 1) /\ EVEN n)
  ) : thm * thm * thm
```

The next example shows how to inductively define the reflexive and transitive closure of relation R. Note that `R`, as a schematic variable, is not quantified in the rules. This is appropriate because it is `RTC R` that has the inductive characterisation, not `RTC` itself.

```
- Hol_reln ‘(!x. RTC R x x) /\                                        3
            (!x z. (?y. R x y /\ RTC R y z) ==> RTC R x z)‘;

> val it =
   (|- !R. (!x. RTC R x x) /\
           !x z. (?y. R x y /\ RTC R y z) ==> RTC R x z,

    |- !R RTC’.
         (!x. RTC’ x x) /\
         (!x z. (?y. R x y /\ RTC’ y z) ==> RTC’ x z)
         ==>
         (!a0 a1. RTC R a0 a1 ==> RTC’ a0 a1),

    |- !R a0 a1. RTC R a0 a1 = (a1 = a0) \/ ?y. R a0 y /\ RTC R y a1
   ) : thm * thm * thm
```

The `Hol_reln` function may be used to define multiple relations, as in the definition of `EVEN` and `ODD`. The relations may or may not be mutually recursive. The clauses for each relation need not be contiguous.

4.6.1 Proofs with Inductive Relations

The "rules" theorem of an inductive relation provides a straightforward way of proving arguments belong to a relation. If confronted with a goal of the form `R x y`, one might make progress by performing a `MATCH_MP_TAC` (or perhaps, an `HO_MATCH_MP_TAC`) with one of the implications in the "rules" theorem.

The "cases" theorem can be used for the same purpose because it is an equality, of the general form `R x y` $\iff \ldots$. Because the right-hand side of this theorem will often include other occurrences of the relation, it is generally not safe to simply rewrite with it. The rewriting-control directives `Once`, `SimpLHS` and `SimpRHS` can be useful here. In addition, the "cases" theorem can be used as an "elimination" form: if one has an assumption of the form `R x y`, rewriting this (perhaps with `FULL_SIMP_TAC` if the term occurs in the goal's assumptions) into the possible ways it may have come about is often a good approach.

Inductive relations naturally also support proof by induction. Because an inductive relation is the least relation satisfying the given rules, one can use induction to show goals of the form

```
∀x y. R x y ⇒ P
```

where `P` is an arbitrary predicate likely including references to variables `x` and `y`.

The low-level approach to goals of this form is to apply

```
HO_MATCH_MP_TAC R_ind
```

A slightly more high-level approach is use the `Induct_on` tactic. (This tactic is also used to perform structural inductions over algebraic data types; see Section 5.3.) When performing a rule induction, the quotation passed to `Induct_on` should be of the constant being used. For the sake of aesthetics, the constant may also be applied to arguments. Thus, one can write

```
Induct_on `R`
```

or

```
Induct_on `R x y`
```

and the effect will be the same.

Chapter 5

Libraries

A *library* is an abstraction intended to provide a higher level of organization for HOL applications. In general, a library can contain a collection of theories, proof procedures, and supporting material, such as documentation. Some libraries simply provide proof procedures, such as `simpLib`, while others provide theories and proof procedures, such as `intLib`. Libraries can include other libraries.

In the HOL system, libraries are typically represented by ML structures named following the convention that library *x* will be found in the ML structure `xLib`. Loading this structure should load all the relevant sub-components of the library and set whatever system parameters are suitable for use of the library.

When the HOL system is invoked in its normal configuration, several useful libraries are automatically loaded. The most basic HOL library is `boolLib`, which supports the definitions of the HOL logic, found in the theory `bool`, and provides a useful suite of definition and reasoning tools.

Another pervasively used library is found in the structure `Parse` (the reader can see that we are not strictly faithful to our convention about library naming). The parser library provides support for parsing and 'pretty-printing' of HOL types, terms, and theorems.

The `boss` library provides a basic collection of standard theories and high-level proof procedures, and serves as a standard platform on which to work. It is preloaded and opened when the HOL system starts up. It includes `boolLib` and `Parse`. Theories provided include `pair`, `sum`, `option`; the arithmetic theories `num`, `prim_rec`, `arithmetic`, and `numeral`; and `list`. Other libraries included in `bossLib` are `goalstackLib`, which provides a proof manager for tactic proofs; `simpLib`, which provides a variety of simplifiers; `numLib`, which provides a decision procedure for arithmetic; `Datatype`, which provides high-level support for defining algebraic datatypes; and `tflLib`, which provides support for defining recursive functions.

5.1 Parsing and Prettyprinting

Every type and term in HOL is ultimately built by application of the primitive (abstract) constructors for types and terms. However, in order to accommodate a wide variety of mathematical expression, HOL provides flexible infrastructure for parsing and pret-

typrinting types and terms through the `Parse` structure.

The term parser supports type inference, overloading, binders, and various fixity declaration (infix, prefix, postfix, and combinations). There are also flags for controlling the behaviour of the parser. Further, the structure of the parser is exposed so that new parsers can be quickly constructed to support user applications.

The parser is parameterized by grammars for types and terms. The behaviour of the parser and prettyprinter is therefore usually altered by grammar manipulations. These can be of two kinds: *temporary* or *permanent*. Temporary changes should be used in library implementations, or in script files for those changes that the user does not wish to have persist in theories descended from the current one. Permanent changes are appropriate for use in script-files, and will be in force in all descendant theories. Functions making temporary changes are signified by a leading `temp_` in their names.

5.1.1 Parsing types

The language of types is a simple one. An abstract grammar for the language is presented in Figure 5.1. The actual grammar (with concrete values for the infix symbols and type operators) can be inspected using the function `type_grammar`.

$$
\begin{array}{lcl}
\tau & ::= & \tau \odot \tau \mid \mathit{vtype} \mid \mathit{tyop} \mid \texttt{(}\ \mathit{tylist}\ \texttt{)}\ \mathit{tyop} \mid \tau\ \mathit{tyop} \mid \texttt{(}\ \tau\ \texttt{)} \mid \tau\texttt{[}\tau\texttt{]} \\
\odot & ::= & \texttt{->} \mid \texttt{\#} \mid \texttt{+} \mid \cdots \\
\mathit{vtype} & ::= & \texttt{'a} \mid \texttt{'b} \mid \texttt{'c} \mid \cdots \\
\mathit{tylist} & ::= & \tau \mid \tau\ \texttt{,}\ \mathit{tylist} \\
\mathit{tyop} & ::= & \texttt{bool} \mid \texttt{list} \mid \texttt{num} \mid \texttt{fun} \mid \cdots
\end{array}
$$

Figure 5.1: An abstract grammar for HOL types (τ). Infixes ($\odot$) always bind more weakly than type operators (tyop) (and type-subscripting ($\tau\texttt{[}\tau\texttt{]}$)), so that $\tau_1 \odot \tau_2\ \mathit{tyop}$ is always parsed as $\tau_1 \odot (\tau_2\ \mathit{tyop})$. Different infixes can have different priorities, and infixes at different priority levels can associate differently (to the left, to the right, or not at all). Users can extend the categories $\odot$ and tyop by making new type definitions, and by directly manipulating the grammar.

Type infixes Infixes may be introduced with the function `add_infix_type`. This sets up a mapping from an infix symbol (such as `->`) to the name of an existing type operator (such as `fun`). The binary symbol needs to be given a precedence level and an associativity. See *REFERENCE* for more details.

Type abbreviations Users can abbreviate common type patterns with *abbreviations*. This is done with the ML function `type_abbrev`:

```
type_abbrev : string * hol_type -> unit
```

An abbreviation is a new type operator, of any number of arguments, that expands into an existing type. For example, one might develop a light-weight theory of numbers extended with an infinity, where the representing type was `num option` (`NONE` would represent the infinity value). One might set up an abbreviation `infnum` that expanded to this underlying type. Polymorphic patterns are supported as well. For example, as described in Section 3.5.1, the abbreviation `set`, of one argument, is such that `:'a set` expands into the type `:'a -> bool`, for any type `:'a`.

When types come to be printed, the expansion of abbreviations done by the parser is reversed. For more information see the documentation of `type_abbrev` in the *REFERENCE*.

5.1.2 Parsing terms

The term parser provides a grammar-based infrastructure for supporting concrete syntax for formalizations. Usually, the HOL grammar gets extended when a new definition or constant specification is made. (The introduction of new constants is discussed in Sections 1.9.3.1 and 1.9.3.2.) However, any identifier can have a parsing status attached at any time. In the following, we explore some of the capabilities of the HOL term parser.

5.1.2.1 Parser architecture

The parser turns strings into terms. It does this in the following series of phases, all of which are influenced by the provided grammar. Usually this grammar is the default global grammar, but users can arrange to use different grammars if they desire. Strictly, parsing occurs after lexing has split the input into a series of tokens. For more on lexing, see Section 1.1.

Concrete Syntax: Features such as infixes, binders and mix-fix forms are translated away, creating an intermediate, "abstract syntax" form (ML type `Absyn`). The possible fixities are discussed in Section 5.1.2.7 below. Concrete syntax forms are added to the grammar with functions such as `add_rule` and `set_fixity` (for which, see the *REFERENCE*). The action of this phase of parsing is embodied in the function `Absyn`.

The `Absyn` data type is constructed using constructors `AQ` (an antiquote, see Section 5.1.3); `IDENT` (an identifier); `QIDENT` (a qualified identifier, given as `thy$ident`); `APP` (an application of one form to another); `LAM` (an abstraction of a variable over a body), and `TYPED` (a form accompanied by a type constraint[1], see Section 5.1.2.4). At this stage of the translation, there is no distinction made between

[1]The types in `Absyn` constraints are not full HOL types, but values from another intermediate type, `Pretype`.

constants and variables: though `QIDENT` forms must be constants, users are also able to refer to constants by giving their bare names.

It is possible for names that occur in the `Absyn` value to be different from any of the tokens that appeared in the original input. For example, the input

```
‘‘if P then Q else R‘‘
```

will turn into

```
APP (APP (APP (IDENT "COND", IDENT "P"), IDENT "Q"), IDENT "R")
```

(This is slightly simplified output: the various constructors for `Absyn`, including `APP`, also take location parameters.)

The standard grammar includes a rule that associates the special mix-fix form for if-then-else expressions with the underlying “name” `COND`. It is `COND` that will eventually be resolved as the constant `bool$COND`.

If the “quotation” syntax with a bare dollar is used, then this phase of the parser will not treat strings as part of a special form. For example, `‘‘$if P‘‘` turns into the `Absyn` form

```
APP(IDENT "if", IDENT "P")
```

not a form involving `COND`.

More typically, one often writes something like `‘‘$+ x‘‘`, which generates the abstract syntax

```
APP(IDENT "+", IDENT "x")
```

Without the dollar-sign, the concrete syntax parser would complain about the fact that the infix plus did not have a left-hand argument. When the successful result of parsing is handed to the next phase, the fact that there is a constant called `+` will give the input its desired meaning.

Symbols can also be “escaped” by enclosing them in parentheses. Thus, the above could be written `‘‘(+) x‘‘` for the same effect.

The user can insert intermediate transformation functions of their own design into the parsing processing at this point. This is done with the function

```
add_absyn_postprocessor
```

The user's function will be of type `Absyn -> Absyn` and can perform whatever changes are appropriate. Like all other aspects of parsing, these functions are part of a grammar: if the user doesn't want to see a particular function used, they can arrange for parsing to be done with respect to a different grammar.

Name Resolution: The bare `IDENT` forms in the `Absyn` value are resolved as free variables, bound names or constants. This process results in a value of the `Preterm` data type, which has similar constructors to those in `Absyn` except with forms for constants. A string can be converted straight to a `Preterm` by way of the `Preterm` function.

A bound name is the first argument to a `LAM` constructor, an identifier occurring on the left-hand side of a case-expression's arrow, or an identifier occurring within a set comprehension's pattern. A constant is a string that is present in the domain of the grammar's "overload map". Free variables are all other identifiers. Free variables of the same name in a term will all have the same type. Identifiers are tested to see if they are bound, and then to see if they are constants. Thus it is possible to write

```
\SUC. SUC + 3
```

and have the string `SUC` be treated as a number in the context of the given abstraction, rather than as the successor constant.

The "overload map" is a map from strings to lists of terms. The terms are usually just constants, but can be arbitrary terms (giving rise to "syntactic macros" or "patterns"). This facility is used to allow a name such as `+` to map to different addition constants in theories such as `arithmetic`, `integer`, and `words`. In this way the "real" names of the constants can be divorced from what the user types. In the case of addition, the natural number plus actually is called `+` (strictly, `arithmetic$+`); but over the integers, it is `int_add`, and over words it is `word_add`. (Note that because each constant is from a different theory and thus a different namespace, they could all have the name `+`.)

When name resolution determines that an identifier should be treated as a constant, it is mapped to a preterm form that lists all of the possibilities for that string. Subsequently, because the terms in the range of the overload map will typically have different types, type inference will often eliminate possibilities from the list. If multiple possibilities remain after type inference has been performed, then a warning will be printed, and one of the possibilities will be chosen. (Users can control which terms are picked when this situation arises.)

When a term in the overload map is chosen as the best option, it is substituted into the term at appropriate position. If the term is a lambda abstraction, then as

many β-reductions are done as possible, using any arguments that the term has been applied to. It is in this way that a syntactic pattern can process arguments. (See also Section 5.1.2.3 for more on syntactic patterns.)

Type Inference: All terms in the HOL logic are well-typed. The kernel enforces this through the API for the `term` data type. (In particular, the `mk_comb` function checks that the type of the first argument is a function whose domain is equal to the type of the second argument.) The parser's job is to turn user-supplied strings into terms. For convenience, it is vital that the user not have to provide types for all of the identifiers they type. (See Section 5.1.2.5 below.)

In the presence of overloaded identifiers, type inference may not be able to assign a unique type to all constants. If multiple possibilities exist, one will be picked when the `Preterm` is finally converted into a genuine term.

Conversion to Term: When a `Preterm` has been type-checked, the final conversion from that type to the `term` type is mostly straightforward. The user can insert further processing at this point as well, so that a user-supplied function modifies the result before the parser returns.

5.1.2.2 Unicode characters

It is possible to have the HOL parsing and printing infrastructure use Unicode characters (written in the UTF-8 encoding). This makes it possible to write and read terms such as

```
∀x. P x ∧ Q x
```

rather than

```
!x. P x /\ Q x
```

If they wish, users may simply define constants that have Unicode characters in their names, and leave it at that. The problem with this approach is that standard tools will likely then create theory files that include (illegal) ML bindings like `val →_def = ...`. The result will be `...Theory.sig` and `...Theory.sml` files that fail to compile, even though the call to `export_theory` may succeed. This problem can be finessed through the use of functions like `set_MLname`, but it's probably best practice to only use alphanumerics in the names of constants, and to then use functions like `overload_on` and `add_rule` to create Unicode syntax for the underlying constant.

If users have fonts with the appropriate repertoire of characters to display their syntax, and are confident that any other users of their theories will too, then this is perfectly reasonable. However, if users wish to retain some backwards compatibility with pure ASCII syntax, they can do so by defining a pure ASCII syntax first. Having done this, they can create a Unicode version of the syntax with the function `Unicode.unicode_version`.

Then, while the trace variable "Unicode" is 0, the ASCII syntax will be used for parsing and printing. If the trace is set to 1, then the Unicode syntax will also work in the parser, and the pretty-printer will prefer it when terms are printed.

For example, in boolScript.sml, the Unicode character for logical and (∧), is set up as a Unicode alternative for /\ with the call

```
val _ = unicode_version {u = UChar.conj, tmnm = "/\\"};
```

(In this context, the Unicode structure has been open-ed, giving access also to the structure UChar which contains bindings for the Greek alphabet, and some common mathematical symbols.)

The argument to unicode_version is a record with fields u and tmnm. Both are strings. The tmnm field can either be the name of a constant, or a token appearing in a concrete syntax rule (possibly mapping to some other name). If the tmnm is only the name of a constant, then, with the trace variable enabled, the string u will be overloaded to the same name. If the tmnm is the same as a concrete syntax rule's token, then the behaviour is to create a new rule mapping to the same name, but with the string u used as the token.

Lexing rules with Unicode characters Roughly speaking, HOL considers characters to be divided into three classes: alphanumerics, non-aggregating symbols and symbols. This affects the behaviour of the lexer when it encounters strings of characters. Unless there is a specific "mixed" token already in the grammar, tokens split when the character class changes. Thus, in the string

```
++a
```

the lexer will see two tokens, ++ and a, because + is a symbol and a is an alphanumeric. The classification of the additional Unicode characters is very simplistic: all Greek letters except λ are alphanumeric; the logical negation symbol ¬ is non-aggregating; and everything else is symbolic. (The exception for λ is to allow strings like λx.x to lex into *four* tokens.)

5.1.2.3 Syntactic patterns ("macros")

The "overload map" mentioned previously is actually a combination of maps, one for parsing, and one for printing. The parsing map is from names to lists of terms, and determines how the names that appear in a Preterm will translate into terms. In essence, bound names turn into bound variables, unbound names not in the domain of the map turn into free variables, and unbound names in the domain of the map turn into one of the elements of the set associated with the given name. Each term in the set of possibilities may have a different type, so type inference will choose from those that have types

consistent with the rest of the given term. If the resulting list contains more than one element, then the term appearing earlier in the list will be chosen.

The most common use-case for the overload map is have names map to constants. In this way, for example, the various numeric theories can map the string "+" to the relevant notions of addition, each of which is a different constant. However, the system has extra flexibility because names can map to arbitrary terms. For example, it is possible to map to specific type-instances of constants. Thus, the string "<=>" maps to equality, but where the arguments are forced to be of type ``:bool``.

Moreover, if the term mapped to is a lambda-abstraction (i.e., of the form $\lambda x.\ M$), then the parser will perform all possible β-reductions for that term and the arguments accompanying it. For example, in `boolTheory` and its descendants, the string "<>" is overloaded to the term ``\x y. ~(x = y)``. Additionally, "<>" is set up at the concrete syntax level as an infix. When the user inputs ``x <> y``, the resulting `Absyn` value is

```
APP(APP(IDENT "<>", IDENT "x"), IDENT "x")
```

The "x" and "y" identifiers will map to free variables, but the "<>" identifier maps to a list containing ``\x y. ~(x = y)``. This term has type

```
:'a -> 'a -> bool
```

and the polymorphic variables are generalisable, allowing type inference to give appropriate (identical) types to `x` and `y`. Assuming that this option is the only overloading for "<>" left after type inference, then the resulting term will be `~(x = y)`. Better, though this will be the underlying structure of the term in memory, it will actually print as ``x <> y``.

If the term mapped to in the overload map contains any free variables, these variables will not be instantiated in any way. In particular, if these variables have polymorphic types, then the type variables in those types will be constant: not subject to instantiation by type inference.

Pretty-printing and syntactic patterns The second part of the "overload map" is a map from terms to strings, specifying how terms should be turned back into identifiers. (Though it does not actually construct an `Absyn` value, this process reverses the name resolution phase of parsing, producing something that is then printed according to the concrete syntax part of the given grammar.)

Because parsing can map single names to complicated term structures, printing must be able to take a complicated term structure back to a single name. It does this by performing term matching.[2] If multiple patterns match the same term, then the printer picks the most specific match (the one that requires least instantiation of the pattern's

[2]The matching done is first-order; contrast the higher-order matching done in the simplifier.

variables). If this still results in multiple, equally specific, possibilities, the most recently added pattern takes precedence. (Users can thus manipulate the printer's preferences by making otherwise redundant calls to the `overload_on` function.)

In the example of the not-equal-to operator above, the pattern will be `~(?x = ?y)`, where the question-marks indicate instantiable pattern variables. If a pattern includes free variables (recall that the `x` and `y` in this example were bound by an abstraction), then these will not be instantiable.

There is one further nicety in the use of this facility: "bigger" matches, covering more of a term, take precedence. The difficulty this can cause is illustrated in the `IS_PREFIX` pattern from `rich_listTheory`. For the sake of backwards compatibility this identifier maps to

```
\x y. isPREFIX y x
```

where `isPREFIX` is a constant from `listTheory`. (The issue is that `IS_PREFIX` expects its arguments in reverse order to that expected by `isPREFIX`.) Now, when this macro is set up the overload map already contains a mapping from the string `"isPREFIX"` to the constant `isPREFIX` (this happens with every constant definition). But after the call establishing the new pattern for `IS_PREFIX`, the `isPREFIX` form will no longer be printed. Nor is it enough, to repeat the call

```
overload_on("isPREFIX", ``isPREFIX``)
```

Instead (assuming that `isPREFIX` is indeed the preferred printing form), the call must be

```
overload_on("isPREFIX", ``\x y. isPREFIX x y``)
```

so that `isPREFIX`'s pattern is as long as `IS_PREFIX`'s.

5.1.2.4 Type constraints

A term can be constrained to be of a certain type. For example, `X:bool` constrains the variable `X` to have type `bool`. An attempt to constrain a term inappropriately will raise an exception: for example,

```
if T then (X:ind) else (Y:bool)
```

will fail because both branches of a conditional must be of the same type. Type constraints can be seen as a suffix that binds more tightly than everything except function application. Thus *term* ... *term* : *hol_type* is equal to (*term* ... *term*) : *hol_type*, but $x < y$: `num` is a legitimate constraint on just the variable y.

The inclusion of : in the symbolic identifiers means that some constraints may need to be separated by white space. For example,

```
$=:bool->bool->bool
```

will be broken up by the HOL lexer as

```
$=: bool -> bool -> bool
```

and parsed as an application of the symbolic identifier `$=:` to the argument list of terms [`bool`, `->`, `bool`, `->`, `bool`]. A well-placed space will avoid this problem:

```
$= :bool->bool->bool
```

is parsed as the symbolic identifier "=" constrained by a type. Instead of the `$`, one can also use parentheses to remove special parsing behaviour from lexemes:

```
(=):bool->bool->bool
```

5.1.2.5 Type inference

Consider the term `x = T`: it (and all of its subterms) has a type in the HOL logic. Now, `T` has type `bool`. This means that the constant = has type `xty -> bool -> bool`, for some type `xty`. Since the type scheme for = is `'a -> 'a -> bool`, we know that `xty` must in fact be `bool` in order for the type instance to be well-formed. Knowing this, we can deduce that the type of `x` must be `bool`.

Ignoring the jargon ("scheme" and "instance") in the previous paragraph, we have conducted a type assignment to the term structure, ending up with a well-typed term. It would be very tedious for users to conduct such argumentation by hand for each term entered to HOL. Thus, HOL uses an adaptation of Milner's type inference algorithm for ML when constructing terms via parsing. At the end of type inference, unconstrained type variables get assigned names by the system. Usually, this assignment does the right thing. However, at times, the most general type is not what is desired and the user must add type constraints to the relevant subterms. For tricky situations, the global variable `show_types` can be assigned. When this flag is set, the prettyprinters for terms and theorems will show how types have been assigned to subterms. If you do not want the system to assign type variables for you, the global variable `guessing_tyvars` can be set to `false`, in which case the existence of unassigned type variables at the end of type inference will raise an exception.

5.1.2.6 Overloading

A limited amount of overloading resolution is performed by the term parser. For example, the 'tilde' symbol (`~`) denotes boolean negation in the initial theory of HOL, and it also denotes the additive inverse in the `integer` and `real` theories. If we load the `integer` theory and enter an ambiguous term featuring `~`, the system will inform us that overloading resolution is being performed.

```
- load "integerTheory";                                                    1
> val it = () : unit

- Term `~~x`;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = `~~x` : term

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

A priority mechanism is used to resolve multiple possible choices. In the example, ~ could be consistently chosen to have type `:bool -> bool` or `:int -> int`, and the mechanism has chosen the former. For finer control, explicit type constraints may be used. In the following session, the `~~x` in the first quotation has type `:bool`, while in the second, a type constraint ensures that `~~x` has type `:int`.

```
- show_types := true;                                                      2
> val it = () : unit

- Term `~(x = ~~x)`;
<<HOL message: more than one resolution of overloading was possible.>>
> val it = `~((x :bool) = ~~x)` : term

- Term `~(x:int = ~~x)`;
> val it = `~((x :int) = ~~x)` : term
```

Note that the symbol ~ stands for two different constants in the second quotation; its first occurrence is boolean negation, while the other two occurrences are the additive inverse operation for integers.

5.1.2.7 Fixities

In order to provide some notational flexibility, constants come in various flavours or *fixities*: besides being an ordinary constant (with no fixity), constants can also be *binders*, *prefixes*, *suffixes*, *infixes*, or *closefixes*. More generally, terms can also be represented using reasonably arbitrary *mixfix* specifications. The degree to which terms bind their associated arguments is known as precedence. The higher this number, the tighter the binding. For example, when introduced, + has a precedence of 500, while the tighter binding multiplication (∗) has a precedence of 600.

Binders A binder is a construct that binds a variable; for example, the universal quantifier. In HOL, this is represented using a trick that goes back to Alonzo Church: a binder is a constant that takes a lambda abstraction as its argument. The lambda binding is used to implement the binding of the construct. This is an elegant and uniform solution.

Thus the concrete syntax `!v. M` is represented by the application of the constant `!` to the abstraction `(\v. M)`.

The most common binders are `!`, `?`, `?!`, and `@`. Sometimes one wants to iterate applications of the same binder, e.g.,

`!x. !y. ?p. ?q. ?r.` *term*`.`

This can instead be rendered

`!x y. ?p q r.` *term*`.`

Infixes Infix constants can associate in one of three different ways: right, left or not at all. (If + were non-associative, then `3 + 4 + 5` would fail to parse; one would have to write `(3 + 4) + 5` or `3 + (4 + 5)` depending on the desired meaning.) The precedence ordering for the initial set of infixes is `/\`, `\/`, `==>`, `=`, `,` (comma[3]). Moreover, all of these constants are right associative. Thus

```
X /\ Y ==> C \/ D, P = E, Q
```

is equal to

```
((X /\ Y) ==> (C \/ D)), ((P = E), Q).
```

An expression

term `<infix>` *term*

is internally represented as

(`<infix>` *term*) *term*

Prefixes Where infixes appear between their arguments, prefixes appear before theirs. This might initially appear to be the same thing as happens with normal function application where the symbol on the left simply has no fixity: is f in $f(x)$ not acting as a prefix? Actually though, in a term such as $f(x)$, where f and x do not have fixities, the syntax is treated as if there is an invisible infix function application between the two tokens: $f \cdot x$. This infix operator binds tightly, so that when one writes $f\,x + y$, the parse is $(f \cdot x) + y$.[4] It is then useful to allow for genuine prefixes so that operators can live at different precedence levels than function application. An example of this is `~`, logical negation. This is a prefix with lower precedence than function application. Normally

$f\ x\ y$ is parsed as $(f\ x)\ y$

[3]When `pairTheory` has been loaded.

[4]There are tighter infix operators: the dot in field selection causes $f\,x.fld$ to parse as $f \cdot (x.fld)$.

but

$$\texttt{~}\ x\ y \qquad \text{is parsed as} \qquad \texttt{~}\ (x\ y)$$

because the precedence of ~ is lower than that of function application. The unary negation symbol would also typically be defined as a prefix, if only to allow one to write

$$negop\ negop\ 3$$

(whatever *negop* happened to be) without needing extra parentheses.

On the other hand, the `univ` syntax for the universal set (see Section 3.5.1) is an example of a prefix operator that binds more tightly than application. This means that `f univ(:'a)` is parsed as `f(univ(:'a))`, not `(f univ)(:'a)` (which parse would fail to type-check).

Suffixes Suffixes appear after their arguments. There are no suffixes introduced into the standard theories available in HOL, but users are always able to introduce their own if they choose. Suffixes are associated with a precedence just as infixes and prefixes are. If `p` is a prefix, `i` an infix, and `s` a suffix, then there are six possible orderings for the three different operators based on their precedences, giving five parses for `p` t_1 `i` t_2 `s` depending on the relative precedences:

Precedences (lowest to highest)	Parses
$p,\ i,\ s$	`p` $(t_1$ `i` $(t_2$ `s`$))$
$p,\ s,\ i$	`p` $((t_1$ `i` $t_2)$ `s`$)$
$i,\ p,\ s$	$($`p` $t_1)$ `i` $(t_2$ `s`$)$
$i,\ s,\ p$	$($`p` $t_1)$ `i` $(t_2$ `s`$)$
$s,\ p,\ i$	$($`p` $(t_1$ `i` $t_2))$ `s`
$s,\ i,\ p$	$(($`p` $t_1)$ `i` $t_2)$ `s`

Closefixes Closefix terms are operators that completely enclose their arguments. An example one might use in the development of a theory of denotational semantics is semantic brackets. Thus, the HOL parsing facilities can be configured to allow one to write `denotation x` as `[| x |]`. Closefixes are not associated with precedences because they can not compete for arguments with other operators.

5.1.2.8 Parser tricks and magic

Here we describe how to achieve some useful effects with the parser in HOL.

Aliasing If one wants a special syntax to be an "alias" for a normal HOL form, this is easy to achieve; both examples so far have effectively done this. However, if one

just wants to have a normal one-for-one substitution of one string for another, one can't use the grammar/syntax phase of parsing to do this. Instead, one can use the overloading mechanism. For example, let us alias `MEM` for `IS_EL`. We need to use the function `overload_on` to overload the original constant for the new name:

```
val _ = overload_on ("MEM", Term`IS_EL`);
```

Making addition right associative If one has a number of old scripts that assume addition is right associative because this is how HOL used to be, it might be too much pain to convert. The trick is to remove all of the rules at the given level of the grammar, and put them back as right associative infixes. The easiest way to tell what rules are in the grammar is by inspection (use `term_grammar()`). With just `arithmeticTheory` loaded, the only infixes at level 500 are + and -. So, we remove the rules for them:

```
val _ = app temp_remove_rules_for_term ["+", "-"];
```

And then we put them back with the appropriate associativity:

```
val _ = app (fn s => temp_add_infix(s, 500, RIGHT)) ["+", "-"];
```

Note that we use the `temp_` versions of these two functions so that other theories depending on this one won't be affected. Further note that we can't have two infixes at the same level of precedence with different associativities, so we have to remove both operators, not just addition.

Mix-fix syntax for *if-then-else*: The first step in bringing this about is to look at the general shape of expressions of this form. In this case, it will be:

```
if ... then ... else ...
```

Because there needs to be a "dangling" term to the right, the appropriate fixity is `Prefix`. Knowing that the underlying term constant is called `COND`, the simplest way to achieve the desired syntax is:

```
val _ = add_rule
   {term_name = "COND", fixity = Prefix 70,
    pp_elements = [TOK "if", BreakSpace(1,0), TM, BreakSpace(1,0),
                   TOK "then", BreakSpace(1,0), TM, BreakSpace(1,0),
                   TOK "else", BreakSpace(1,0)],
    paren_style = Always,
    block_style = (AroundEachPhrase, (PP.CONSISTENT, 0))};
```

The actual rule is slightly more complicated, and may be found in the sources for the theory `bool`.

Mix-fix syntax for term substitution: Here the desire is to be able to write something like:

[t_1 / t_2] t_3

denoting the substitution of t_1 for t_2 in t_3, perhaps translating to `SUB` t_1 t_2 t_3. This looks like it should be another `Prefix`, but the choice of the square brackets (`[` and `]`) as delimiters would conflict with the concrete syntax for list literals if this was done. Given that list literals are effectively of the `CloseFix` class, the new syntax must be of the same class. This is easy enough to do: we set up syntax

[t_1 / t_2]

to map to `SUB` t_1 t_2, a value of a functional type, that when applied to a third argument will look right.[5] The rule for this is thus:

```
val _ = add_rule
          {term_name = "SUB", fixity = Closefix,
           pp_elements = [TOK "[", TM, TOK "/", TM, TOK "]"],
           paren_style = OnlyIfNecessary,
           block_style = (AroundEachPhrase, (PP.INCONSISTENT, 2))};
```

5.1.2.9 Hiding constants

The following function can be used to hide the constant status of a name from the quotation parser.

```
val hide   : string -> ({Name : string, Thy : string} list *
                        {Name : string, Thy : string} list)
```

Evaluating `hide "`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

[5] Note that doing the same thing for the *if-then-else* example in the previous example would be inappropriate, as it would allow one to write

`if` P `then` Q `else`

without the trailing argument.

is still a constant for the constructors, theories, etc; `hide` affects parsing and printing by removing the given name from the "overload map" described above in Section 5.1.2.1. Note that the effect of `hide` is *temporary*; its effects do not persist in theories descended from the current one. See the *REFERENCE* entry for `hide` for more details, including an explanation of the return type.

The function

```
reveal : string -> unit
```

undoes hiding.

The function

```
hidden : string -> bool
```

tests whether a string is the name of a hidden constant.

5.1.2.10 Adjusting the pretty-print depth

The following ML reference can be used to adjust the maximum depth of printing

```
max_print_depth : int ref
```

The default print depth is -1, which is interpreted as meaning no maximum. Subterms nested more deeply than the maximum print depth are printed as `...`. For example:

1

```
- ADD_CLAUSES;
> val it =
    |- (0 + m = m) /\ (m + 0 = m) /\ (SUC m + n = SUC (m + n)) /\
       (m + SUC n = SUC (m + n)) : thm

- max_print_depth := 3;
> val it = () : unit
- ADD_CLAUSES;
> val it = |- (... + ... = m) /\ (... = ...) /\ ... /\ ... : thm
```

5.1.3 Quotations and antiquotation

Logic-related syntax in the HOL system is typically passed to the parser in special forms known as *quotations*. A basic quotation is delimited by single back-ticks (i.e., `, ASCII character 96). When quotation values are printed out by the ML interactive loop, they look rather ugly because of the special filtering that is done to these values before the ML interpreter even sees them:

```
- val q = ‘f x = 3‘;                                                    1
> val ’a q = [QUOTE " (*#loc 1 11*)f x = 3"] : ’a frag list
```

Quotations (Moscow ML prints the type as `’a frag list`) are the raw input form expected by the various HOL parsers. They are also polymorphic (to be explained below). Thus the function `Parse.Term` function takes a (term) quotation and returns a term, and is thus of type

```
term quotation -> term
```

The term and type parsers can also be called implicitly by using double back-ticks as delimiters. For the type parser, the first non-space character after the leading delimiter must also be a colon. Thus:

```
- val t = ‘‘p /\ q‘‘;                                                   2
> val t = ‘‘p /\ q‘‘ : term

- val ty = ‘‘:’a -> bool‘‘;
> val ty = ‘‘:’a -> bool‘‘ : hol_type
```

The expression bound to ML variable `t` above is actually expanded to an application of the function `Parse.Term` to the quotation argument `‘p /\ q‘`. Similarly, the second expression expands into an application of `Parse.Type` to the quotation `‘:’a -> bool‘`.

The significant advantage of quotations over normal ML strings is that they can include new-line and backslash characters without requiring special quoting. Newlines occur whenever terms get beyond the trivial in size, while backslashes occur in not just the representation of λ, but also the syntax for conjunction and disjunction.

If a quotation is to include a back-quote character, then this should be done by using the quotation syntax's own escape character, the caret (^, ASCII character 94). To get a bare caret, things are slightly more complicated. If a sequence of carets is followed by white-space (including a newline), then that sequence of carets is passed to the HOL parser unchanged. Otherwise, one caret can be obtained by writing two in a row. (This last rule is analogous to the way in ML string syntax treats the back-slash.) Thus:

```
- ‘‘f ^‘ x ‘‘;                                                          3
<<HOL message: inventing new type variable names: ’a, ’b, ’c>>
> val it = ‘‘f ‘ x‘‘ : term

- ‘‘f ^ x‘‘;
<<HOL message: inventing new type variable names: ’a, ’b, ’c>>
> val it = ‘‘f ^ x‘‘ : term
```

The rule for carets not followed by white-space is illustrated here, including an example of what happens when the quoting rule is not followed:

4
```
- ``f ^^+ x``;
<<HOL message: inventing new type variable names: 'a, 'b, 'c>>
> val it = ``f ^+ x`` : term

- ``f ^+ x``;
! Toplevel input:
! (Parse.Term [QUOTE " (*#loc 2 3*)f ", ANTIQUOTE (+),
!              QUOTE " (*#loc 2 7*) x"]);
!                                                 ^
! Ill-formed infix expression
```

The main use of the caret is to introduce *antiquotations* (as suggested in the last example above). 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 bound to an ML variable x. In this case `^(`x`)` can be abbreviated by `^`x.

The following session illustrates antiquotation.

1
```
- val y = ``x+1``;
> val y = ``x + 1`` : term

val z = ``y = ^y``;
> val z = ``y = x + 1`` : term

- ``!x:num.?y:num.^z``;
> val it = ``!x. ?y. y = x + 1`` : term
```

Types may be antiquoted as well:

2
```
- val pred = ``:'a -> bool``;
> val pred = ``:'a -> bool`` : hol_type

- ``:^pred -> bool``;
> val it = ``:('a -> bool) -> bool`` : hol_type
```

Quotations are polymorphic, and the type variable of a quotation corresponds to the type of entity that can be antiquoted into that quotation. Because the term parser expects only antiquoted terms, antiquoting a type into a term quotation requires the use of `ty_antiq`. For example,

```
- ‘‘!P:^pred. P x ==> Q x‘‘;                                              3

! Toplevel input:
! Term ‘!P:^pred. P x ==> Q x‘;
!           ^^^^
! Type clash: expression of type
!   hol_type
! cannot have type
!   term

- ‘‘!P:^(ty_antiq pred). P x ==> Q x‘‘;
> val it = ‘!P. P x ==> Q x‘ : term
```

5.1.4 Backwards compatibility of syntax

This section of the manual documents the (extensive) changes made to the parsing of HOL terms and types in the Taupo release (one of the HOL3 releases) and beyond from the point of view of a user who doesn't want to know how to use the new facilities, but wants to make sure that their old code continues to work cleanly.

The changes which may cause old terms to fail to parse are:

- The precedence of type annotations has completely changed. It is now a very tight suffix (though with a precedence weaker than that associated with function application), instead of a weak one. This means that `(x,y:bool # bool)` should now be written as `(x,y):bool # bool`. The previous form will now be parsed as a type annotation applying to just the `y`. This change brings the syntax of the logic closer to that of SML and should make it generally easier to annotate tuples, as one can now write

  $$(x : \tau_1,\ y : \tau_2, \ldots z : \tau_n)$$

  instead of

  $$(x : \tau_1,\ (y : \tau_2, \ldots (z : \tau_n)))$$

  where extra parentheses have had to be added just to allow one to write a frequently occurring form of constraint.

- Most arithmetic operators are now left associative instead of right associative. In particular, $+$, $-$, $*$ and `DIV` are all left associative. Similarly, the analogous operators in other numeric theories such as `integer` and `real` are also left associative. This brings the HOL parser in line with standard mathematical practice.

- The binding equality in `let` expressions is treated exactly the same way as equalities in other contexts. In previous versions of HOL, equalities in this context have a different, weak binding precedence.

- The old syntax for conditional expressions has been removed. Thus the string ``$p$ => $q$ | $r$`` must now be written ``if $p$ then $q$ else $r$`` instead.

- Some lexical categories are more strictly policed. String literals (strings inside double quotes) and numerals can't be used unless the relevant theories have been loaded. Nor can these literals be used as variables inside binding scopes.

5.2 A Simple Interactive Proof Manager

The *goal stack* provides a simple interface to tactic-based interactive proof. When one uses tactics to decompose a proof, many intermediate states arise; the goalstack takes care of the necessary bookkeeping. The implementation of goalstacks reported here is a re-design of Larry Paulson's original conception.

The goalstack library is automatically loaded when HOL starts up.

The abstract types `goalstack` and `proofs` are the focus of backwards proof operations. The type `proofs` can be regarded as a list of independent goalstacks. Most operations act on the head of the list of goalstacks; there are also operations so that the focus can be changed.

5.2.1 Starting a goalstack proof

```
g        : term quotation -> proofs
set_goal : goal -> proofs
```

Recall that the type `goal` is an abbreviation for `term list * term`. To start on a new goal, one gives `set_goal` a goal. This creates a new goalstack and makes it the focus of further operations.

A shorthand for `set_goal` is the function `g`: it invokes the parser automatically, and it doesn't allow the goal to have any assumptions.

Calling `set_goal`, or `g`, adds a new proof attempt to the existing ones, *i.e.*, rather than overwriting the current proof attempt, the new attempt is stacked on top.

5.2.2 Applying a tactic to a goal

```
expandf : tactic -> goalstack
expand  : tactic -> goalstack
e       : tactic -> goalstack
```

How does one actually do a goalstack proof then? In most cases, the application of tactics to the current goal is done with the function `expand`. In the rare case that one wants to apply an *invalid* tactic, then `expandf` is used. (For an explanation of invalid tactics, see Chapter 24 of Gordon & Melham.) The abbreviation `e` may also be used to expand a tactic.

5.2.3 Undo

```
b          : unit -> goalstack
drop       : unit -> proofs
dropn      : int  -> proofs
backup     : unit -> goalstack
restart    : unit -> goalstack
set_backup : int  -> unit
```

Often (we are tempted to say *usually*!) one takes a wrong path in doing a proof, or makes a mistake when setting a goal. To undo a step in the goalstack, the function `backup` and its abbreviation `b` are used. This will restore the goalstack to its previous state.

To directly back up all the way to the original goal, the function `restart` may be used. Obviously, it is also important to get rid of proof attempts that are wrong; for that there is `drop`, which gets rid of the current proof attempt, and `dropn`, which eliminates the top n proof attempts.

Each proof attempt has its own *undo-list* of previous states. The undo-list for each attempt is of fixed size (initially 12). If you wish to set this value for the current proof attempt, the function `set_backup` can be used. If the size of the backup list is set to be smaller than it currently is, the undo list will be immediately truncated. You can not undo a "proofs-level" operation, such as `set_goal` or `drop`.

5.2.4 Viewing the state of the proof manager

```
p            : unit -> goalstack
status       : unit -> proofs
top_goal     : unit -> goal
top_goals    : unit -> goal list
initial_goal : unit -> goal
top_thm      : unit -> thm
```

To view the state of the proof manager at any time, the functions `p` and `status` can be used. The former only shows the top subgoals in the current goalstack, while the second gives a summary of every proof attempt.

To get the top goal or goals of a proof attempt, use `top_goal` and `top_goals`. To get the original goal of a proof attempt, use `initial_goal`.

Once a theorem has been proved, the goalstack that was used to derive it still exists (including its undo-list): its main job now is to hold the theorem. This theorem can be retrieved with `top_thm`.

5.2.5 Switch focus to a different subgoal or proof attempt

```
r             : int -> goalstack
R             : int -> proofs
rotate        : int -> goalstack
rotate_proofs : int -> proofs
```

Often we want to switch our attention to a different goal in the current proof, or a different proof. The functions that do this are `rotate` and `rotate_proofs`, respectively. The abbreviations `r` and `R` are simpler to type in.

5.3 High Level Proof—`bossLib`

The library `bossLib` marshals some of the most widely used theorem proving tools in HOL and provides them with a convenient interface for interaction. The library currently focuses on three things: definition of datatypes and functions; high-level interactive proof operations, and composition of automated reasoners. Loading `bossLib` commits one to working in a context that already supplies the theories of booleans, pairs, sums, the option type, arithmetic, and lists.

5.3.1 Support for high-level proof steps

The following functions use information in the database to ease the application of HOL's underlying functionality:

```
type_rws     : hol_type -> thm list
Induct       : tactic
Cases        : tactic
Cases_on     : term quotation -> tactic
Induct_on    : term quotation -> tactic
```

The function `type_rws` will search for the given type in the underlying `TypeBase` database and return useful rewrite rules for that type. The rewrite rules of the datatype are built from the injectivity and distinctness theorems, along with the case constant definition. The simplification tactics `RW_TAC`, `SRW_TAC`, and the simpset `(srw_ss())` automatically include these theorems. Other tactics used with other simpsets will need these theorems to be manually added.

The `Induct` tactic makes it convenient to invoke induction. When it is applied to a goal, the leading universal quantifier is examined; if its type is that of a known datatype, the appropriate structural induction tactic is extracted and applied.

The `Cases` tactic makes it convenient to invoke case analysis. The leading universal quantifier in the goal is examined; if its type is that of a known datatype, the appropriate structural case analysis theorem is extracted and applied.

The `Cases_on` tactic takes a quotation, which is parsed into a term M, and then M is searched for in the goal. If M is a variable, then a variable with the same name is searched for. Once the term to split over is known, its type and the associated facts are obtained from the underlying database and used to perform the case split. If some free variables of M are bound in the goal, an attempt is made to remove (universal) quantifiers so that the case split has force. Finally, M need not appear in the goal, although it should at least contain some free variables already appearing in the goal. Note that the `Cases_on` tactic is more general than `Cases`, but it does require an explicit term to be given.

The `Induct_on` tactic takes a quotation, which is parsed into a term M, and then M is searched for in the goal. If M is a variable, then a variable with the same name is searched for. Once the term to induct on is known, its type and the associated facts are obtained from the underlying database and used to perform the induction. If M is not a variable, a new variable v not already occurring in the goal is created, and used to build a term $v = M$ which the goal is made conditional on before the induction is performed. First however, all terms containing free variables from M are moved from the assumptions to the conclusion of the goal, and all free variables of M are universally quantified. `Induct_on` is more general than `Induct`, but it does require an explicit term to be given.

Three supplementary entry-points have been provided for more exotic inductions:

`completeInduct_on` performs complete induction on the term denoted by the given quotation. Complete induction allows a seemingly [6] stronger induction hypothesis than ordinary mathematical induction: to wit, when inducting on n, one is allowed to assume the property holds for *all* m smaller than n. Formally: $\forall P.\ (\forall x.\ (\forall y.\ y < x \supset P\,y) \supset P\,x) \supset \forall x.\ P\,x$. This allows the inductive hypothesis to be used more than once, and also allows instantiating the inductive hypothesis to other than the predecessor.

`measureInduct_on` takes a quotation, and breaks it apart to find a term and a measure function with which to induct. For example, if one wanted to induct on the length of a list `L`, the invocation `measureInduct_on ‘LENGTH L‘` would be be appropriate.

[6] Complete induction and ordinary mathematical induction are each derivable from the other.

`recInduct` takes a induction theorem generated by `Define` or `Hol_defn` and applies it to the current goal.

5.3.2 Automated reasoners

`bossLib` brings together the most powerful reasoners in HOL and tries to make it easy to compose them in a simple way. We take our basic reasoners from `mesonLib`, `simpLib`, and `numLib`, but the point of `bossLib` is to provide a layer of abstraction so the user has to know only a few entry-points.[7] (These underlying libraries, and others providing similarly powerful tools are described in detail in sections below.)

```
PROVE      : thm list -> term -> thm
PROVE_TAC  : thm list -> tactic

METIS_TAC  : thm list -> tactic
METIS_PROVE: thm list -> term -> thm

DECIDE     : term quotation -> thm
DECIDE_TAC : tactic
```

The inference rule `PROVE` (and the corresponding tactic `PROVE_TAC`) takes a list of theorems and a term, and attempts to prove the term using a first order reasoner. The two `METIS` functions perform the same functionality but use a different underlying proof method. The `PROVE` entry-points refer to the `meson` library, which is further described in Section 5.4.1 below. The `METIS` system is described in Section 5.4.2. The inference rule `DECIDE` (and the corresponding tactic `DECIDE_TAC`) applies a decision procedure that (at least) handles statements of linear arithmetic.

```
RW_TAC   : simpset -> thm list -> tactic
SRW_TAC  : ssfrag list -> thm list -> tactic
&&       : simpset * thm list -> simpset  (* infix *)
std_ss   : simpset
arith_ss : simpset
list_ss  : simpset
srw_ss   : unit -> simpset
```

The rewriting tactic `RW_TAC` works by first adding the given theorems into the given simpset; then it simplifies the goal as much as possible; then it performs case splits on any conditional expressions in the goal; then it repeatedly (1) eliminates all hypotheses of the form $v = M$ or $M = v$ where v is a variable not occurring in M, (2) breaks down any equations between constructor terms occurring anywhere in the goal. Finally, `RW_TAC` lifts `let`-expressions within the goal so that the binding equations appear as abbreviations in the assumptions.

[7] In the mid 1980's Graham Birtwistle advocated such an approach, calling it 'Ten Tactic HOL'.

The tactic SRW_TAC is similar to RW_TAC, but works with respect to an underlying simpset (accessible through the function srw_ss) that is updated as new context is loaded. This simpset can be augmented through the addition of "simpset fragments" (ssfrag values) and theorems. In situations where there are many large types stored in the system, RW_TAC's performance can suffer because it repeatedly adds all of the rewrite theorems for the known types into a simpset before attacking the goal. On the other hand, SRW_TAC loads rewrites into the simpset underneath srw_ss() just once, making for faster operation in this situation.

bossLib provides a number of simplification sets. The simpset for pure logic, sums, pairs, and the option type is named std_ss. The simpset for arithmetic is named arith_ss, and the simpset for lists is named list_ss. The simpsets provided by bossLib strictly increase in strength: std_ss is contained in arith_ss, and arith_ss is contained in list_ss. The infix combinator && is used to build a new simpset from a given simpset and a list of theorems. HOL's simplification technology is described further in Section 5.5 below and in the *REFERENCE*.

```
by : term quotation * tactic -> tactic (* infix 8 *)
SPOSE_NOT_THEN : (thm -> tactic) -> tactic
```

The function by is an infix operator that takes a quotation and a tactic tac. The quotation is parsed into a term M. When the invocation "M by tac" is applied to a goal (A, g), a new subgoal (A, M) is created and tac is applied to it. If the goal is proved, the resulting theorem is broken down and added to the assumptions of the original goal; thus the proof proceeds with the goal $((M :: A), g)$. (Note however, that case-splitting will happen if the breaking-down of $\vdash M$ exposes disjunctions.) Thus by allows a useful style of 'assertional' or 'Mizar-like' reasoning to be mixed with ordinary tactic proof.[8]

The SPOSE_NOT_THEN entry-point initiates a proof by contradiction by assuming the negation of the goal and driving the negation inwards through quantifiers. It provides the resulting theorem as an argument to the supplied function, which will use the theorem to build and apply a tactic.

5.4 First Order Proof—mesonLib and metisLib

First order proof is a powerful theorem-proving technique that can finish off complicated goals. Unlike tools such as the simplifier, it either proves a goal outright, or fails. It can not transform a goal into a different (and more helpful) form.

[8]Proofs in the Mizar system are readable documents, unlike most tactic-based proofs.

5.4.1 Model elimination—mesonLib

The meson library is an implementation of the model-elimination method for finding proofs of goals in first-order logic. There are three main entry-points:

```
MESON_TAC     : thm list -> tactic
ASM_MESON_TAC : thm list -> tactic
GEN_MESON_TAC : int -> int -> int -> thm list -> tactic
```

Each of these tactics attempts to prove the goal. They will either succeed in doing so, or fail with a "depth exceeded" exception. If the branching factor in the search-space is high, the meson tactics may also take a very long time to reach the maximum depth.

All of the meson tactics take a list of theorems. These extra facts are used by the decision procedure to help prove the goal. MESON_TAC ignores the goal's assumptions; the other two entry-points include the assumptions as part of the sequent to be proved.

The extra parameters to GEN_MESON_TAC provide extra control of the behaviour of the iterative deepening that is at the heart of the search for a proof. In any given iteration, the algorithm searches for a proof of depth no more than a parameter d. The default behaviour for MESON_TAC and ASM_MESON_TAC is to start d at 0, to increment it by one each time a search fails, and to fail if d exceeds the value stored in the reference value mesonLib.max_depth. By way of contrast, GEN_MESON_TAC min max step starts d at min, increments it by step, and gives up when d exceeds max.

The PROVE_TAC function from bossLib performs some normalisation, before passing a goal and its assumptions to ASM_MESON_TAC. Because of this normalisation, in most circumstances, PROVE_TAC should be preferred to ASM_MESON_TAC.

5.4.2 Resolution—metisLib

The metis library is an implementation of the resolution method for finding proofs of goals in first-order logic. There are two main entry-points:

```
METIS_TAC   : thm list -> tactic
METIS_PROVE : thm list -> term -> thm
```

Both functions take a list of theorems, and these are used as lemmas in the proof. METIS_TAC is a tactic, and will either succeed in proving the goal, or if unsuccessful will either fail or loop forever. METIS_PROVE takes a term t and tries to prove a theorem with conclusion t: if successful, the theorem $\vdash t$ is returned. As for METIS_TAC, it might fail or loop forever if the proof search is unsuccessful.

The metisLib family of proof tools implement the ordered resolution and ordered paramodulation calculus for first order logic, which usually makes them better suited to goals requiring non-trivial equality reasoning than the tactics in mesonLib.

5.5 Simplification—simpLib

The simplifier is HOL's most sophisticated rewriting engine. It is recommended as a general purpose work-horse during interactive theorem-proving. As a rewriting tool, the simplifier's general role is to apply theorems of the general form

$$\vdash l = r$$

to terms, replacing instances of l in the term with r. Thus, the basic simplification routine is a *conversion*, taking a term t, and returning a theorem $\vdash t = t'$, or the exception UNCHANGED.

The basic conversion is

```
simpLib.SIMP_CONV : simpLib.simpset -> thm list -> term -> thm
```

The first argument, a simpset, is the standard way of providing a collection of rewrite rules (and other data, to be explained below) to the simplifier. There are simpsets accompanying most of HOL's major theories. For example, the simpset bool_ss in boolSimps embodies all of the usual rewrite theorems one would want over boolean formulas:

```
- SIMP_CONV bool_ss [] ``p /\ T \/ ~(q /\ r)``;                    1
> val it = |- p /\ T \/ ~(q /\ r) = p \/ ~q \/ ~r : thm
```

In addition to rewriting with the obvious theorems, bool_ss is also capable of performing simplifications that are not expressible as simple theorems:

```
- SIMP_CONV bool_ss [] ``?x. (\y. P (f y)) x /\ (x = z)``;         2
> val it = |- (?x. (\y. P (f y)) x /\ (x = z)) = P (f z) : thm
```

In this example, the simplifier performed a β-reduction in the first conjunct under the existential quantifier, and then did an "unwinding" or "one-point" reduction, recognising that the only possible value for the quantified variable x was the value z.

The second argument to SIMP_CONV is a list of theorems to be added to the provided simpset, and used as additional rewrite rules. In this way, users can temporarily augment standard simpsets with their own rewrites. If a particular set of theorems is often used as such an argument, then it is possible to build a simpset value to embody these new rewrites.

For example, the rewrite arithmeticTheory.LEFT_ADD_DISTRIB, which states that $p(m+n) = pm + pn$ is not part of any of HOL's standard simpsets. This is because it can cause an unappealing increase in term size (there are two occurrences of p on the right hand side of the theorem). Nonetheless, it is clear that this theorem may be appropriate on occasion:

```
- SIMP_CONV bossLib.arith_ss [LEFT_ADD_DISTRIB] ``p * (n + 1)``;   3
> val it = |- p * (n + 1) = p + n * p : thm
```

Note how the `arith_ss` simpset has not only simplified the intermediate `(p * 1)` term, but also re-ordered the addition to put the simpler term on the left, and sorted the multiplication's arguments.

5.5.1 Simplification tactics

The simplifier is implemented around the conversion `SIMP_CONV`, which is a function for 'converting' terms into theorems. To apply the simplifier to goals (alternatively, to perform tactic-based proofs with the simplifier), HOL provides five tactics, all of which are available in `bossLib`.

5.5.1.1 `SIMP_TAC : simpset -> thm list -> tactic`

`SIMP_TAC` is the simplest simplification tactic: it attempts to simplify the current goal (ignoring the assumptions) using the given simpset and the additional theorems. It is no more than the lifting of the underlying `SIMP_CONV` conversion to the tactic level through the use of the standard function `CONV_TAC`.

5.5.1.2 `ASM_SIMP_TAC : simpset -> thm list -> tactic`

Like `SIMP_TAC`, `ASM_SIMP_TAC` simplifies the current goal (leaving the assumptions untouched), but includes the goal's assumptions as extra rewrite rules. Thus:

4

```
1 subgoal:
> val it =
    P x
    ------------------------------------
      x = 3
     : goalstack

- e (ASM_SIMP_TAC bool_ss []);
OK..
1 subgoal:
> val it =
    P 3
    ------------------------------------
      x = 3
     : goalstack
```

In this example, `ASM_SIMP_TAC` used `x = 3` as an additional rewrite rule, and replaced the `x` of `P x` with `3`. When an assumption is used by `ASM_SIMP_TAC` it is converted into rewrite rules in the same way as theorems passed in the list given as the tactic's second argument. For example, an assumption `~P` will be treated as the rewrite `|- P = F`.

5.5.1.3 FULL_SIMP_TAC : simpset -> thm list -> tactic

The tactic FULL_SIMP_TAC simplifies not only a goal's conclusion but its assumptions as well. It proceeds by simplifying each assumption in turn, additionally using earlier assumptions in the simplification of later assumptions. After being simplified, each assumption is added back into the goal's assumption list with the STRIP_ASSUME_TAC tactic. This means that assumptions that become conjunctions will have each conjunct assumed separately. Assumptions that become disjunctions will cause one new sub-goal to be created for each disjunct. If an assumption is simplified to false, this will solve the goal.

FULL_SIMP_TAC attacks the assumptions in the order in which they appear in the list of terms that represent the goal's assumptions. Typically then, the first assumption to be simplified will be the assumption most recently added. Viewed in the light of goalstackLib's printing of goals, FULL_SIMP_TAC works its way up the list of assumptions, from bottom to top.

The following demonstrates a simple use of FULL_SIMP_TAC:

```
    x + y < z                                                    5
    ------------------------------------
      0.  f x < 10
      1.  x = 4
     : goalstack

- e (FULL_SIMP_TAC bool_ss []);
OK..
1 subgoal:
> val it =
    4 + y < z
    ------------------------------------
      0.  f 4 < 10
      1.  x = 4
     : goalstack
```

In this example, the assumption `x = 4` caused the `x` in the assumption `f x < 10` to be replaced by `4`. The `x` in the goal was similarly replaced. If the assumptions had appeared in the opposite order, only the `x` of the goal would have changed.

The next session more demonstrates more interesting behaviour:

```
> val it =                                                          6
    f x + 1 < 10
    ------------------------------------
      x <= 4
     : goalstack

- e (FULL_SIMP_TAC bool_ss [arithmeticTheory.LESS_OR_EQ]);
OK..
2 subgoals:
> val it =
    f 4 + 1 < 10
    ------------------------------------
      x = 4

    f x + 1 < 10
    ------------------------------------
      x < 4
     : goalstack
```

In this example, the goal was rewritten with the theorem stating

$$\vdash x \leq y \equiv x < y \vee x = y$$

Turning the assumption into a disjunction resulted in two sub-goals. In the second of these, the assumption `x = 4` further simplified the rest of the goal.

5.5.1.4 `RW_TAC : simpset -> thm list -> tactic`

Though its type is the same as the simplification tactics already described, `RW_TAC` is an "augmented" tactic. It is augmented in two ways:

- When simplifying the goal, the provided simpset is augmented not only with the theorems explicitly passed in the second argument, but also with all of the rewrite rules from the `TypeBase`, and also with the goal's assumptions.
- `RW_TAC` also does more than just perform simplification. It also repeatedly "strips" the goal. For example, it moves the antecedents of implications into the assumptions, splits conjunctions, and case-splits on conditional expressions. This behaviour can rapidly remove a lot of syntactic complexity from goals, revealing the kernel of the problem. On the other hand, this aggressive splitting can also result in a large number of sub-goals. `RW_TAC`'s augmented behaviours are intertwined with phases of simplification in a way that is difficult to describe.

5.5.1.5 `SRW_TAC : ssfrag list -> thm list -> tactic`

The tactic `SRW_TAC` has a different type from the other simplification tactics. It does not take a simpset as an argument. Instead its operation always builds on the built-in simpset `srw_ss()` (further described in Section 5.5.2.5). The theorems provided as

SRW_TAC's second argument are treated in the same way as by the other simplification tactics. Finally, the list of simpset fragments are merged into the underlying simpset, allowing the user to merge in additional simplification capabilities if desired.

For example, to include the Presburger decision procedure, one could write

```
SRW_TAC [ARITH_ss][]
```

Simpset fragments are described below in Section 5.5.3.

The SRW_TAC tactic performs the same mixture of simplification and goal-splitting as does RW_TAC. The main differences between the two tactics lie in the fact that the latter can be inefficient when working with a large TypeBase, and in the fact that working with SRW_TAC saves one from having to explicitly construct simpsets that include all of the current context's "appropriate" rewrites. The latter "advantage" is based on the assumption that (srw_ss()) never includes inappropriate rewrites. The presence of unused rewrites is never a concern: the presence of rewrites that do the wrong thing can be a major irritation.

5.5.2 The standard simpsets

HOL comes with a number of standard simpsets. All of these are accessible from within bossLib, though some originate in other structures.

5.5.2.1 pure_ss and bool_ss

The pure_ss simpset (defined in structure pureSimps) contains no rewrite theorems at all, and plays the role of a blank slate within the space of possible simpsets. When constructing a completely new simpset, pure_ss is a possible starting point. The pure_ss simpset has just two components: congruence rules for specifying how to traverse terms, and a function that turns theorems into rewrite rules. Congruence rules are further described in Section 5.5.5; the generation of rewrite rules from theorems is described in Section 5.5.4.

The bool_ss simpset (defined in structure boolSimps) is often used when other simpsets might do too much. It contains rewrite rules for the boolean connectives, and little more. It contains all of the de Morgan theorems for moving negations in over the connectives (conjunction, disjunction, implication and conditional expressions), including the quantifier rules that have $\neg(\forall x.\, P(x))$ and $\neg(\exists x.\, P(x))$ on their left-hand sides. It also contains the rules specifying the behaviour of the connectives when the constants T and F appear as their arguments. (One such rule is `|- T /\ p = p`.)

As in the example above, bool_ss also performs β-reductions and one-point unwindings. The latter turns terms of the form

$$\exists x.\, P(x) \wedge \ldots (x = e) \ldots \wedge Q(x)$$

into

$$P(e) \wedge \ldots \wedge Q(e)$$

Similarly, unwinding will turn $\forall x.\ (x = e) \Rightarrow P(x)$ into $P(e)$.

Finally, `bool_ss` also includes congruence rules that allow the simplifier to make additional assumptions when simplifying implications and conditional expressions. This feature is further explained in Section 5.5.4 below, but can be illustrated by some examples (the first also demonstrates unwinding under a universal quantifier):

7
```
- SIMP_CONV bool_ss [] ``!x. (x = 3) /\ P x ==> Q x /\ P 3``;
> val it = |- (!x. (x = 3) /\ P x ==> Q x /\ P 3) = P 3 ==> Q 3 : thm

- SIMP_CONV bool_ss [] ``if ~(x = 3) then P x else Q x``;
> val it = |- (if ~(x = 3) then P x else Q x) =
              (if ~(x = 3) then P x else Q 3) : thm
```

5.5.2.2 `std_ss`

The `std_ss` simpset is defined in `bossLib`, and adds rewrite rules pertinent to the types of sums, pairs, options and natural numbers to `bool_ss`.

8
```
- SIMP_CONV std_ss [] ``FST (x,y) + OUTR (INR z)``;
<<HOL message: inventing new type variable names: 'a, 'b>>
> val it = |- FST (x,y) + OUTR (INR z) = x + z : thm

- SIMP_CONV std_ss [] ``case SOME x of NONE => P | SOME y => f y``;
> val it = |- (case SOME x of NONE => P | SOME v => f v) = f x : thm
```

With the natural numbers, the `std_ss` simpset can calculate with ground values, and also includes a suite of "obvious rewrites" for formulas including variables.

9
```
- SIMP_CONV std_ss [] ``P (0 <= x) /\ Q (y + x - y)``;
> val it = |- P (0 <= x) /\ Q (y + x - y) = P T /\ Q x : thm

- SIMP_CONV std_ss [] ``23 * 6 + 7 ** 2 - 31 DIV 3``;
> val it = |- 23 * 6 + 7 ** 2 - 31 DIV 3 = 177 : thm
```

5.5.2.3 `arith_ss`

The `arith_ss` simpset (defined in `bossLib`) extends `std_ss` by adding the ability to decide formulas of Presburger arithmetic, and to normalise arithmetic expressions (collecting coefficients, and re-ordering summands). The underlying natural number decision procedure is that described in Section 5.7 below.

These two facets of the `arith_ss` simpset are demonstrated here:

```
- SIMP_CONV arith_ss [] ‘‘x < 3 /\ P x ==> x < 20 DIV 2‘‘;                    10
> val it = |- x < 3 /\ P x ==> x < 20 DIV 2 = T : thm

- SIMP_CONV arith_ss [] ‘‘2 * x + y - x + y‘‘;
> val it = |- 2 * x + y - x + y = x + 2 * y : thm
```

Note that subtraction over the natural numbers works in ways that can seem unintuitive. In particular, coefficient normalisation may not occur when first expected:

```
- SIMP_CONV arith_ss [] ‘‘2 * x + y - z + y‘‘;                               11
! Uncaught exception:
! UNCHANGED
```

Over the natural numbers, the expression $2x + y - z + y$ is not equal to $2x + 2y - z$. In particular, these expressions are not equal when $2x + y < z$.

5.5.2.4 list_ss

The last pure simpset value in bossLib, list_ss adds rewrite theorems about the type of lists to arith_ss. These rewrites include the obvious facts about the list type's constructors NIL and CONS, such as the fact that CONS is injective:

```
(h1 :: t1 = h2 :: t2) = (h1 = h2) /\ (t1 = t2)
```

Conveniently, list_ss also includes rewrites for the functions defined by primitive recursion over lists. Examples include MAP, FILTER and LENGTH. Thus:

```
- SIMP_CONV list_ss [] ‘‘MAP (\x. x + 1) [1;2;3;4]‘‘;                        12
> val it = |- MAP (\x. x + 1) [1; 2; 3; 4] = [2; 3; 4; 5] : thm

- SIMP_CONV list_ss [] ‘‘FILTER (\x. x < 4) [1;2;y + 4]‘‘;
> val it = |- FILTER (\x. x < 4) [1; 2; y + 4] = [1; 2] : thm

- SIMP_CONV list_ss [] ‘‘LENGTH (FILTER ODD [1;2;3;4;5])‘‘;
> val it = |- LENGTH (FILTER ODD [1; 2; 3; 4; 5]) = 3 : thm
```

These examples demonstrate how the simplifier can be used as a general purpose symbolic evaluator for terms that look a great deal like those that appear in a functional programming language. Note that this functionality is also provided by computeLib (see Section 5.6 below); computeLib is more efficient, but less general than the simplifier. For example:

```
- EVAL ‘‘FILTER (\x. x < 4) [1;2;y + 4]‘‘;                                   13
> val it =
    |- FILTER (\x. x < 4) [1; 2; y + 4] =
       1::2::(if y + 4 < 4 then [y + 4] else []) : thm
```

5.5.2.5 The "stateful" simpset—srw_ss()

The last simpset exported by bossLib is hidden behind a function. The srw_ss value has type unit -> simpset, so that one must type srw_ss() in order to get a simpset value. This use of a function type allows the underlying simpset to be stored in an ML reference, and allows it to be updated dynamically. In this way, referential transparency is deliberately broken. All of the other simpsets will always behave identically: SIMP_CONV bool_ss is the same simplification routine wherever and whenever it is called.

In contrast, srw_ss is designed to be updated. When a theory is loaded, when a new type is defined, the value behind srw_ss() changes, and the behaviour of SIMP_CONV applied to (srw_ss()) changes with it. The design philosophy behind srw_ss is that it should always be a reasonable first choice in all situations where the simplifier is used.

This versatility is illustrated in the following example:

14
```
- Hol_datatype `tree = Leaf | Node of num => tree => tree`;
<<HOL message: Defined type: "tree">>
> val it = () : unit

- SIMP_CONV (srw_ss()) [] ``Node x Leaf Leaf = Node 3 t1 t2``;
<<HOL message: Initialising SRW simpset ... done>>
> val it =
    |- (Node x Leaf Leaf = Node 3 t1 t2) =
       (x = 3) /\ (Leaf = t1) /\ (Leaf = t2) : thm

- load "pred_setTheory";
> val it = () : unit

- SIMP_CONV (srw_ss()) [] ``x IN { y | y < 6}``;
> val it = |- x IN {y | y < 6} = x < 6 : thm
```

Users can augment the stateful simpset themselves with the function

```
   BasicProvers.export_rewrites : string list -> unit
```

The strings passed to export_rewrites are the names of theorems in the current segment (those that will be exported when export_theory is called). Not only are these theorems added to the underlying simpset in the current session, but they will be added in future sessions when the current theory is reloaded.

15
```
- val tsize_def = Define‘
  (tsize Leaf = 0) /\
  (tsize (Node n t1 t2) = n + tsize t1 + tsize t2)
‘;
Definition has been stored under "tsize_def".
> val tsize_def =
    |- (tsize Leaf = 0) /\
       !n t1 t2. tsize (Node n t1 t2) = n + tsize t1 + tsize t2 : thm

- val _ = BasicProvers.export_rewrites ["tsize_def"];

- SIMP_CONV (srw_ss()) [] ‘‘tsize (Node 4 (Node 6 Leaf Leaf) Leaf)‘‘;
> val it = |- tsize (Node 4 (Node 6 Leaf Leaf) Leaf) = 10 : thm
```

As a general rule, `(srw_ss())` includes all of its context's "obvious rewrites", as well as code to do standard calculations (such as the arithmetic performed in the above example). It does not include decision procedures that may exhibit occasional poor performance, so the simpset fragments containing these procedures should be added manually to those simplification invocations that need them.

5.5.3 Simpset fragments

The simpset fragment is the basic building block that is used to construct simpset values. There is one basic function that performs this construction:

```
op ++  : simpset * ssfrag -> simpset
```

where ++ is an infix. In general, it is best to build on top of the `pure_ss` simpset or one of its descendants in order to pick up the default "filter" function for converting theorems to rewrite rules. (This filtering process is described below in Section 5.5.4.3.)

For major theories (or groups thereof), a collection of relevant simpset fragments is usually found in the module `<thy>Simps`, with `<thy>` the name of the theory. For example, simpset fragments for the theory of natural numbers are found in `numSimps`, and fragments for lists are found in `listSimps`.

Some of the distribution's standard simpset fragments are described in Table 5.1. These and other simpset fragments are described in more detail in the *REFERENCE*.

Simpset fragments are ultimately constructed with the `SSFRAG` constructor:

```
SSFRAG : {
  convs  : convdata list,
  rewrs  : thm list,
  ac     : (thm * thm) list,
  filter : (controlled_thm -> controlled_thm list) option,
  dprocs : Traverse.reducer list,
  congs  : thm list,
  name   : string option
} -> ssfrag
```

`BOOL_ss`	Standard rewrites for the boolean operators (conjunction, negation &c), as well as a conversion for performing β-reduction. (In `boolSimps`.)
`CONG_ss`	Congruence rules for implication and conditional expressions. (In `boolSimps`.)
`ARITH_ss`	The natural number decision procedure for universal Presburger arithmetic. (In `numSimps`.)
`PRED_SET_AC_ss`	AC-normalisation for unions and intersections over sets. (In `pred_setSimps`.)

Table 5.1: Some of HOL's standard simpset fragments

A complete description of the various fields of the record passed to `SSFRAG`, and their meaning is given in *REFERENCE*. The `rewrites` function provides an easy route to constructing a fragment that just includes a list of rewrites:

```
rewrites : thm list -> ssfrag
```

5.5.4 Rewriting with the simplifier

Rewriting is the simplifier's "core operation". This section describes the action of rewriting in more detail.

5.5.4.1 Basic rewriting

Given a rewrite rule of the form

$$\vdash \ell = r$$

the simplifier will perform a top-down scan of the input term t, looking for *matches* (see Section 5.5.4.4 below) of ℓ inside t. This match will occur at a sub-term of t (call it t_0) and will return an instantiation. When this instantiation is applied to the rewrite rule, the result will be a new equation of the form

$$\vdash t_0 = r'$$

Because the system then has a theorem expressing an equivalence for t_0 it can create the new equation

$$\vdash \underbrace{(\ldots t_0 \ldots)}_{t} = (\ldots r' \ldots)$$

The traversal of the term to be simplified is repeated until no further matches for the simplifier's rewrite rules are found. The traversal strategy is

1. While there are any matches for stored rewrite rules at this level, continue to apply them. The order in which rewrite rules are applied can *not* be relied on, except that when a simpset includes two rewrites with exactly the same left-hand sides, the rewrite added later will get matched in preference. (This allows a certain amount of rewrite-overloading in the construction of simpsets.)

2. Recurse into the term's sub-terms. The way in which terms are traversed at this step can be controlled by *congruence rules* (an advanced feature, see Section 5.5.5.1 below)

3. If step 2 changed the term at all, try another phase of rewriting at this level. If this fails, or if there was no change from the traversal of the sub-terms, try any embedded decision procedures (see Section 5.5.5.3). If the rewriting phase or any of the decision procedures altered the term, return to step 1. Otherwise, finish.

5.5.4.2 Conditional rewriting

The above description is a slight simplification of the true state of affairs. One particularly powerful feature of the simplifier is that it really uses *conditional* rewrite rules. These are theorems of the form

$$\vdash P \Rightarrow (\ell = r)$$

When the simplifier finds a match for term ℓ during its traversal of the term, it attempts to discharge the condition P. If the simplifier can simplify the term P to truth, then the instance of ℓ in the term being traversed can be replaced by the appropriate instantiation of r.

When simplifying P (a term that does not necessarily even occur in the original), the simplifier may find itself applying another conditional rewrite rule. In order to stop excessive recursive applications, the simplifier keeps track of a stack of all the side-conditions it is working on. The simplifier will give up on side-condition proving if it notices a repetition in this stack. There is also a user-accessible variable, `Cond_rewr.stack_limit` which specifies the maximum size of stack the simplifier is allowed to use.

Conditional rewrites can be extremely useful. For example, theorems about division and modulus are frequently accompanied by conditions requiring the divisor to be non-zero. The simplifier can often discharge these, particularly if it includes an arithmetic decision procedure. For example, the theorem `MOD_MOD` from the theory `arithmetic` states

$$\vdash 0 < n \;\Rightarrow\; (k \,\mathsf{MOD}\, n) \,\mathsf{MOD}\, n = k \,\mathsf{MOD}\, n$$

The simplifier (specifically, `SIMP_CONV arith_ss [MOD_MOD]`) can use this theorem to simplify the term `(k MOD (x + 1)) MOD (x + 1)`: the arithmetic decision procedure can prove that `0 < x + 1`, justifying the rewrite.

Though conditional rewrites are powerful, not every theorem of the form described above is an appropriate choice. A badly chosen rewrite may cause the simplifier's performance to degrade considerably, as it wastes time attempting to prove impossible side-conditions. For example, the simplifier is not very good at finding existential witnesses. This means that the conditional rewrite

$$\vdash x < y \wedge y < z \Rightarrow (x < z = \top)$$

will not work as one might hope. In general, the simplifier is not a good tool for performing transitivity reasoning. (Try first-order tools such as `PROVE_TAC` instead.)

5.5.4.3 Generating rewrite rules from theorems

There are two routes by which a theorem for rewriting can be passed to the simplifier: either as an explicit argument to one of the ML functions (`SIMP_CONV`, `ASM_SIMP_TAC` etc) that take theorem lists as arguments, or by being included in a simpset fragment which is merged into a simpset. In both cases, these theorems are transformed before being used. The transformations applied are designed to make interactive use as convenient as possible.

In particular, it is not necessary to pass the simplifier theorems that are exactly of the form

$$\vdash P \Rightarrow (\ell = r)$$

Instead, the simplifier will transform its input theorems to extract rewrites of this form itself. The exact transformation performed is dependent on the simpset being used: each simpset contains its own "filter" function which is applied to theorems that are added to it. Most simpsets use the filter function from the `pure_ss` simpset (see Section 5.5.2.1). However, when a simpset fragment is added to a full simpset, the fragment can specify an additional filter component. If specified, this function is of type `controlled_thm -> controlled_thm list`, and is applied to each of the theorems produced by the existing simpset's filter. (A "controlled" theorem is one that is accompanied by a piece of "control" data expressing the limit (if any) on the number of times it can be applied. See Section 5.5.5.4 for how users can introduce these limits. The "control" type appears in the ML module `BoundedRewrites`.)

The rewrite-producing filter in `pure_ss` strips away conjunctions, implications and universal quantifications until it has either an equality theorem, or some other boolean form. For example, the theorem `ADD_MODULUS` states

$$\begin{array}{ll} \vdash & (\forall n\ x.\ 0 < n \Rightarrow ((x + n)\,\mathsf{MOD}\,n = x\,\mathsf{MOD}\,n)) \ \wedge \\ & (\forall n\ x.\ 0 < n \Rightarrow ((n + x)\,\mathsf{MOD}\,n = x\,\mathsf{MOD}\,n)) \end{array}$$

This theorem becomes two rewrite rules

$$\vdash 0 < n \Rightarrow ((x + n)\,\mathsf{MOD}\,n = x\,\mathsf{MOD}\,n)$$
$$\vdash 0 < n \Rightarrow ((n + x)\,\mathsf{MOD}\,n = x\,\mathsf{MOD}\,n)$$

If looking at an equality where there are variables on the right-hand side that do not occur on the left-hand side, the simplifier transforms this to the rule

$$\vdash (\ell = r) = \top$$

Similarly, if a boolean negation $\neg P$, becomes the rule

$$\vdash P = \bot$$

and other boolean formulas P become

$$\vdash P = \top$$

Finally, if looking at an equality whose left-hand side is itself an equality, and where the right-hand side is not an equality as well, the simplifier transforms $(x = y) = P$ into the two rules

$$\vdash (x = y) = P$$
$$\vdash (y = x) = P$$

This is generally useful. For example, a theorem such as

$$\vdash \neg(\mathsf{SUC}\,n = 0)$$

will cause the simplifier to rewrite both $(\mathsf{SUC}\,n = 0)$ and $(0 = \mathsf{SUC}\,n)$ to false.

The restriction that the right-hand side of such a rule not itself be an equality is a simple heuristic that prevents some forms of looping.

5.5.4.4 Matching rewrite rules

Given a rewrite theorem $\vdash \ell = r$, the first stage of performing a rewrite is determining whether or not ℓ can be instantiated so as to make it equal to the term that is being rewritten. This process is known as matching. For example, if ℓ is the term $\mathsf{SUC}(n)$, then matching it against the term $\mathsf{SUC}(3)$ will succeed, and find the instantiation $n \mapsto 3$. In contrast with unification, matching is not symmetrical: a pattern $\mathsf{SUC}(3)$ will not match the term $\mathsf{SUC}(n)$.

The simplifier uses a special form of higher-order matching. If a pattern includes a variable of some function type (f say), and that variable is applied to an argument a that includes no variables except those that are bound by an abstraction at a higher scope, then the combined term $f(a)$ will match any term of the appropriate type as long as the only occurrences of the bound variables in a are in sub-terms matching a.

Assume for the following examples that the variable x is bound at a higher scope. Then, if $f(x)$ is to match $x + 4$, the variable f will be instantiated to $(\lambda x.\ x + 4)$. If $f(x)$ is to match $3 + z$, then f will be instantiated to $(\lambda x.\ 3 + z)$. Further $f(x + 1)$ matches $x + 1 < 7$, but does not match $x + 2 < 7$.

Higher-order matching of this sort makes it easy to express quantifier movement results as rewrite rules, and have these rules applied by the simplifier. For example, the theorem

$$\vdash (\exists x.\ P(x) \vee Q(x)) = (\exists x.\ P(x)) \vee (\exists x.\ Q(x))$$

has two variables of a function-type (P and Q), and both are applied to the bound variable x. This means that when applied to the input

$$\exists z.\ z < 4 \vee z + x = 5 * z$$

the matcher will find the instantiation

$$P \mapsto (\lambda z.\ z < 4)$$
$$Q \mapsto (\lambda z.\ z + x = 5 * z)$$

Performing this instantiation, and then doing some β-reduction on the rewrite rule, produces the theorem

$$\vdash (\exists z.\ z < 4 \vee z + x = 5 * z) = (\exists z.\ z < 4) \vee (\exists z.\ z + x = 5 * z)$$

as required.

Another example of a rule that the simplifier will use successfully is

$$\vdash f \circ (\lambda x.\ g(x)) = (\lambda x.\ f(g(x)))$$

The presence of the abstraction on the left-hand side of the rule requires an abstraction to appear in the term to be matched, so this rule can be seen as an implementation of a method to move abstractions up over function compositions.

An example of a possible left-hand side that will *not* match as generally as might be liked is $(\exists x.\ P(x + y))$. This is because the predicate P is applied to an argument that includes the free variable y.

5.5.5 Advanced features

This section describes some of the simplifier's advanced features.

5.5.5.1 Congruence rules

Congruence rules control the way the simplifier traverses a term. They also provide a mechanism by which additional assumptions can be added to the simplifier's context, representing information about the containing context. The simplest congruence rules are built into the `pure_ss` simpset. They specify how to traverse application and abstraction terms. At this fundamental level, these congruence rules are little more than the rules of inference `ABS`

$$\frac{\Gamma \vdash t_1 = t_2}{\Gamma \vdash (\lambda x.\ t_1) = (\lambda x.\ t_2)}$$

(where $x \notin \Gamma$) and `MK_COMB`

$$\frac{\Gamma \vdash f = g \qquad \Delta \vdash x = y}{\Gamma \cup \Delta \vdash f(x) = g(y)}$$

When specifying the action of the simplifier, these rules should be read upwards. With `ABS`, for example, the rule says "when simplifying an abstraction, simplify the body t_1 to some new t_2, and then the result is formed by re-abstracting with the bound variable x."

Further congruence rules should be added to the simplifier in the form of theorems, via the `congs` field of the records passed to the `SSFRAG` constructor. Such congruence rules should be of the form

$$cond_1 \Rightarrow cond_2 \Rightarrow \ldots (E_1 = E_2)$$

where E_1 is the form to be rewritten. Each $cond_i$ can either be an arbitrary boolean formula (in which case it is treated as a side-condition to be discharged) or an equation of the general form

$$\forall \vec{v}.\ ctxt_1 \Rightarrow ctxt_2 \Rightarrow \ldots (V_1(\vec{v}) = V_2(\vec{v}))$$

where the variable V_2 must occur free in E_2.

For example, the theorem form of `MK_COMB` would be

$$\vdash (f = g) \Rightarrow (x = y) \Rightarrow (f(x) = g(y))$$

and the theorem form of `ABS` would be

$$\vdash (\forall x.\ f(x) = g(x)) \Rightarrow (\lambda x.\ f(x)) = (\lambda x.\ g(x))$$

The form for `ABS` demonstrates how it is possible for congruence rules to handle bound variables. Because the congruence rules are matched with the higher-order match of Section 5.5.4.4, this rule will match all possible abstraction terms.

These simple examples have not yet demonstrated the use of $ctxt$ conditions on subequations. An example of this is the congruence rule (found in `CONG_ss`) for implications. This states

$$\vdash (P = P') \Rightarrow (P' \Rightarrow (Q = Q')) \Rightarrow (P \Rightarrow Q = P' \Rightarrow Q')$$

This rule should be read: "When simplifying $P \Rightarrow Q$, first simplify P to P'. Then assume P', and simplify Q to Q'. Then the result is $P' \Rightarrow Q'$."

The rule for conditional expressions is

$$\vdash\ (P = P') \Rightarrow (P' \Rightarrow (x = x')) \Rightarrow (\neg P' \Rightarrow (y = y')) \ \Rightarrow \\ (\mathsf{if}\ P\ \mathsf{then}\ x\ \mathsf{else}\ y = \mathsf{if}\ P'\ \mathsf{then}\ x'\ \mathsf{else}\ y')$$

This rule allows the guard to be assumed when simplifying the true-branch of the conditional, and its negation to be assumed when simplifying the false-branch.

The contextual assumptions from congruence rules are turned into rewrites using the mechanisms described in Section 5.5.4.3.

Congruence rules can be used to achieve a number of interesting effects. For example, a congruence can specify that sub-terms *not* be simplified if desired. This might be used to prevent simplification of the branches of conditional expressions:

$$\vdash (P = P') \Rightarrow (\mathsf{if}\ P\ \mathsf{then}\ x\ \mathsf{else}\ y = \mathsf{if}\ P'\ \mathsf{then}\ x\ \mathsf{else}\ y)$$

If added to the simplifier, this rule will take precedence over any other rules for conditional expressions (masking the one above from `CONG_ss`, say), and will cause the simplifier to only descend into the guard. With the standard rewrites (from `BOOL_ss`):

$$\begin{array}{lcl} \vdash\ \mathsf{if}\ \top\ \mathsf{then}\ x\ \mathsf{else}\ y & = & x \\ \vdash\ \mathsf{if}\ \bot\ \mathsf{then}\ x\ \mathsf{else}\ y & = & y \end{array}$$

users can choose to have the simplifier completely ignore a conditional's branches until that conditional's guard is simplified to either true or false.

5.5.5.2 AC-normalisation

The simplifier can be used to normalise terms involving associative and commutative constants. This process is known as *AC-normalisation*. The simplifier will perform AC-normalisation for those constants which have their associativity and commutativity theorems provided in a constituent simpset fragment's `ac` field.

For example, the following simpset fragment will cause AC-normalisation of disjunctions

```
SSFRAG {ac = [(DISJ_ASSOC, DISJ_COMM)],
        rewrs = [], filter = NONE, convs = [],
        dprocs = [], congs = []}
```

The pair of provided theorems must state

$$\begin{array}{rcl} x \oplus y & = & y \oplus x \\ x \oplus (y \oplus z) & = & (x \oplus y) \oplus z \end{array}$$

for a constant $\oplus$. The theorems may be universally quantified, and the associativity theorem may be oriented either way. Further, either the associativity theorem or the commutativity theorem may be the first component of the pair. Assuming the simpset fragment above is bound to the ML identifier DISJ_ss, its behaviour is demonstrated in the following example:

```
- SIMP_CONV (bool_ss ++ DISJ_ss) [] ``p /\ q \/ r \/ P z``;                16
<<HOL message: inventing new type variable names: 'a>>
> val it = |- p /\ q \/ r \/ P z = r \/ P z \/ p /\ q : thm
```

The order of operands in the AC-normal form that the simplifer's AC-normalisation works toward is unspecified. However, the normal form is always right-associated. Note also that the arith_ss simpset, and the ARITH_ss fragment which is its basis, have their own bespoke normalisation procedures for addition over the natural numbers. Mixing AC-normalisation, as described here, with arith_ss can cause the simplifier to go into an infinite loop.

AC theorems can also be added to simpsets via the theorem-list part of the tactic and conversion interface, using the special rewrite form AC:

```
- SIMP_CONV bool_ss [AC DISJ_ASSOC DISJ_COMM] ``p /\ q \/ r \/ P z``;      17
<<HOL message: inventing new type variable names: 'a>>
> val it = |- p /\ q \/ r \/ P z = r \/ P z \/ p /\ q : thm
```

See Section 5.5.5.4 for more on special rewrite forms.

5.5.5.3 Embedding code

The simplifier features two different ways in which user-code can be embedded into its traversal and simplification of input terms. By embedding their own code, users can customise the behaviour of the simplifier to a significant extent.

User conversions The simpler of the two methods allows the simplifier to include user-supplied conversions. These are added to simpsets in the convs field of simpset fragments. This field takes lists of values of type

```
{ name: string,
 trace: int,
   key: (term list * term) option,
  conv: (term list -> term -> thm) -> term list -> term -> thm}
```

The name and trace fields are used when simplifier tracing is turned on. If the conversion is applied, and if the simplifier trace level is greater than or equal to the trace field, then a message about the conversion's application (including its name) will be emitted.

The key field of the above record is used to specify the sub-terms to which the conversion should be applied. If the value is NONE, then the conversion will be tried at every

position. Otherwise, the conversion is applied at term positions matching the provided pattern. The first component of the pattern is a list of variables that should be treated as constants when finding pattern matches. The second component is the term pattern itself. Matching against this component is *not* done by the higher-order match of Section 5.5.4.4, but by a higher-order "term-net". This form of matching does not aim to be precise; it is used to efficiently eliminate clearly impossible matches. It does not check types, and does not check multiple bindings. This means that the conversion will not only be applied to terms that are exact matches for the supplied pattern.

Finally, the conversion itself. Most uses of this facility are to add normal HOL conversions (of type `term->thm`), and this can be done by ignoring the `conv` field's first two parameters. For a conversion `myconv`, the standard idiom is to write `K (K myconv)`. If the user desires, however, their code *can* refer to the first two parameters. The second parameter is the stack of side-conditions that have been attempted so far. The first enables the user's code to call back to the simplifier, passing the stack of side-conditions, and a new side-condition to solve. The `term` argument must be of type `:bool`, and the recursive call will simplify it to true (and call `EQT_ELIM` to turn a term t into the theorem $\vdash t$). This restriction may be lifted in a future version of HOL but as it stands, the recursive call can *only* be used for side-condition discharge. Note also that it is the user's responsibility to pass an appropriately updated stack of side-conditions to the recursive invocation of the simplifier.

A user-supplied conversion should never return the reflexive identity (an instance of $\vdash t = t$). This will cause the simplifier to loop. Rather than return such a result, raise a `HOL_ERR` or `Conv.UNCHANGED` exception. (Both are treated the same by the simplifier.)

Context-aware decision procedures Another, more involved, method for embedding user code into the simplifier is *via* the `dprocs` field of the simpset fragment structure. This method is more general than adding conversions, and also allows user code to construct and maintain its own bespoke logical contexts.

The `dprocs` field requires lists of values of the type `Traverse.reducer`. These values are constructed with the constructor `REDUCER`:

```
REDUCER : {initial : context,
           addcontext : context * thm list -> context,
           apply : {solver : term list -> term -> thm,
                    context : context,
                    stack : term list} -> term -> thm}
       -> reducer
```

The `context` type is an alias for the built-in ML type `exn`, that of exceptions. The exceptions here are used as a "universal type", capable of storing data of any type. For example, if the desired data is a pair of an integer and a boolean, then the following declaration could be made:

```
exception my_data of int * bool
```

It is not necessary to make this declaration visible with a wide scope. Indeed, only functions accessing and creating contexts of this form need to see it. For example:

```
fun get_data c = (raise c) handle my_data (i,b) => (i,b)
fun mk_ctxt (i,b) = my_data(i,b)
```

When creating a value of `reducer` type, the user must provide an initial context, and two functions. The first, `addcontext`, is called by the simplifier's traversal mechanism to give every embedded decision procedure access to theorems representing new context information. For example, this function is called with theorems from the current assumptions in `ASM_SIMP_TAC`, and with the theorems from the theorem-list arguments to all of the various simplification functions. As a term is traversed, the congruence rules governing this traversal may also provide additional theorems; these will also be passed to the `addcontext` function. (Of course, it is entirely up to the `addcontext` function as to how these theorems will be handled; they may even be ignored entirely.)

When an embedded reducer is applied to a term, the provided `apply` function is called. As well as the term to be transformed, the `apply` function is also passed a record containing a side-condition solver, the decision procedure's current context, and the stack of side-conditions attempted so far. The stack and solver are the same as the additional arguments provided to user-supplied conversions. The power of the reducer abstraction is having access to a context that can be built appropriately for each decision procedure.

Decision procedures are applied last when a term is encountered by the simplifier. More, they are applied *after* the simplifier has already recursed into any sub-terms and tried to do as much rewriting as possible. This means that although simplifier rewriting occurs in a top-down fashion, decision procedures will be applied bottom-up and only as a last resort.

As with user-conversions, decision procedures must raise an exception rather than return instances of reflexivity.

5.5.5.4 Special rewrite forms

Some of the simplifier's features can be accessed in a relatively simple way by using ML functions to construct special theorem forms. These special theorems can then be passed in the simplification tactics' theorem-list arguments.

Two of the simplifier's advanced features, AC-normalisation and congruence rules can be accessed in this way. Rather than construct a custom simpset fragment including the required AC or congruence rules, the user can instead use the functions `AC` or `Cong`:

```
AC : thm -> thm -> thm
Cong : thm -> thm
```

For example, if the theorem value

```
AC DISJ_ASSOC DISJ_COMM
```

appears amongst the theorems passed to a simplification tactic, then the simplifier will perform AC-normalisation of disjunctions. The `Cong` function provides a similar interface for the addition of new congruence rules.

Two other functions provide a crude mechanism for controlling the number of times an individual rewrite will be applied.

```
Once : thm -> thm
Ntimes : thm -> int -> thm
```

A theorem "wrapped" in the `Once` function will only be applied once when the simplifier is applied to a given term. A theorem wrapped in `Ntimes` will be applied as many times as given in the integer parameter.

Simplifying at particular sub-terms We have already seen (Section 5.5.5.1 above) that the simplifier's congruence technology can be used to force the simplifier to ignore particular terms. The example in the section above discussed how a congruence rule might be used to ensure that only the guards of conditional expressions should be simplified.

In many proofs, it is common to want to rewrite only on one side or the other of a binary connective (often, this connective is an equality). For example, this occurs when rewriting with equations from complicated recursive definitions that are not just structural recursions. In such definitions, the left-hand side of the equation will have a function symbol attached to a sequence of variables, e.g.:

```
|- f x y = ... f (g x y) z ...
```

Theorems of a similar shape are also returned as the "cases" theorems from inductive definitions.

Whatever their origin, such theorems are the classic example of something to which one would want to attach the `Once` qualifier. However, this may not be enough: one may wish to prove a result such as

```
f (constructor x) y = ... f (h x y) z ...
```

(With relations, the goal may often feature an implication instead of an equality.) In this situation, one often wants to expand just the instance of `f` on the left, leaving the other occurrence alone. Using `Once` will expand only one of them, but without specifying which one is to be expanded.

The solution to this problem is to use special congruence rules, constructed as special forms that can be passed as theorems like `Once`. The functions

```
SimpL : term -> thm
SimpR : term -> thm
```

construct congruence rules to force rewriting to the left or right of particular terms. For example, if opn is a binary operator, SimpL ``(opn)`` returns Cong applied to the theorem

```
|- (x = x') ==> (opn x y = opn x' y)
```

Because the equality case is so common, the special values SimpLHS and SimpRHS are provided to force simplification on the left or right of an equality respectively. These are just defined to be applications of SimpL and SimpR to equality.

Note that these rules apply throughout a term, not just to the uppermost occurrence of an operator. Also, the topmost operator in the term need not be that of the congruence rule. This behaviour is an automatic consequence of the implementation in terms of congruence rules.

5.5.5.5 Limiting simplification

In addition to the Once and Ntimes theorem-forms just discussed, which limit the number of times a particular rewrite is applied, the simplifier can also be limited in the total number of rewrites it performs. The limit function (in simpLib and bossLib)

```
limit : int -> simpset -> simpset
```

records a numeric limit in a simpset. When a limited simpset then works over a term, it will never apply more than the given number of rewrites to that term. When conditional rewrites are used, the rewriting done in the discharge of side-conditions counts against the limit, as long as the rewrite is ultimately applied. The application of user-provided congruence rules, user-provided conversions and decision procedures also all count against the limit.

When the simplifier yields control to a user-provided conversion or decision procedure it cannot guarantee that these functions will ever return (and they may also take arbitrarily long to work, often a worry with arithmetic decision procedures), but use of limit is otherwise a good method for ensuring that simplification terminates.

5.5.5.6 Rewriting with arbitrary pre-orders

In addition to simplifying with respect to equality, it is also possible to use the simplifier to "rewrite" with respect to a relation that is reflexive and transitive (a *preorder*). This can be a very powerful way of working with transition relations in operational semantics.

Imagine, for example, that one has set up a "deep embedding" of the λ-calculus. This will entail the definition of a new type (lamterm, say) within the logic, as well

as definitions of appropriate functions (e.g., substitution) and relations over `lamterm`. One is likely to work with the reflexive and transitive closure of β-reduction ($\rightarrow_\beta^*$). This relation has congruence rules such as

$$\frac{M_1 \ \rightarrow_\beta^* \ M_2}{M_1\, N \ \rightarrow_\beta^* \ M_2\, N} \qquad \frac{N_1 \ \rightarrow_\beta^* \ N_2}{M\, N_1 \ \rightarrow_\beta^* \ M\, N_2}$$

$$\frac{M_1 \ \rightarrow_\beta^* \ M_2}{(\lambda v.M_1) \ \rightarrow_\beta^* \ (\lambda v.M_2)}$$

and one important rewrite

$$\frac{}{(\lambda v.M)\, N \ \rightarrow_\beta^* \ M[v := N]}$$

Having to apply these rules manually in order to show that a given starting term can reduce to particular destination is usually very painful, involving many applications, not only of the theorems above, but also of the theorems describing reflexive and transitive closure (see Section 3.5.3).

Though the λ-calculus is non-deterministic, it is also confluent, so the following theorem holds:

$$\frac{\beta\text{-nf } N \quad M_1 \ \rightarrow_\beta^* \ M_2}{M_1 \ \rightarrow_\beta^* \ N \ = \ M_2 \ \rightarrow_\beta^* \ N}$$

This is the critical theorem that justifies the switch from rewriting with equality to rewriting with $\rightarrow_\beta^*$. It says that if one has a term $M_1 \rightarrow_\beta^* N$, with N a β-normal form, and if M_1 rewrites to M_2 under $\rightarrow_\beta^*$, then the original term is equal to $M_2 \rightarrow_\beta^* N$. With luck, M_2 will actually be syntactically identical to N, and the reflexivity of $\rightarrow_\beta^*$ will prove the desired result. Theorems such as these, that justify the switch from one rewriting relation to another are known as *weakening congruences*.

When adjusted appropriately, the simplifier can be modified to exploit the five theorems above, and automatically prove results such as

$$u((\lambda f\, x.f(f\, x))v) \rightarrow_\beta^* u(\lambda x.v(v\, x))$$

(on the assumption that the terms u and v are λ-calculus variables, making the result a β-normal form).

In addition, one will quite probably have various rewrite theorems that one will want to use in addition to those specified above. For example, if one has earlier proved a theorem such as

$$K\, x\, y \rightarrow_\beta^* x$$

then the simplifier can take this into account as well.

The function achieving all this is

```
simpLib.add_relsimp  : {trans: thm, refl: thm, weakenings: thm list,
                        subsets: thm list, rewrs : thm list} ->
                       simpset -> simpset
```

The fields of the record that is the first argument are:

`trans` The theorem stating that the relation is transitive, in the form $\forall xyz.R\,x\,y \wedge R\,y\,z \Rightarrow Rxz$.

`refl` The theorem stating that the relation is reflexive, in the form $\forall x.R\,x\,x$.

`weakenings` A list of weakening congruences, of the general form $P_1 \Rightarrow P_2 \Rightarrow \cdots (t_1 = t_2)$, where at least one of the P_i will presumably mention the new relation R applied to a variable that appears in t_1. Other antecedents may be side-conditions such as the requirement in the example above that the term N be in β-normal form.

`subsets` Theorems of the form $R'\,x\,y \Rightarrow R\,x\,y$. These are used to augment the resulting simpset's "filter" so that theorems in the context mentioning R' will derive useful rewrites involving R. In the example of β-reduction, one might also have a relation $\rightarrow^*_{wh}$ for weak-head reduction. Any weak-head reduction is also a β-reduction, so it can be useful to have the simplifier automatically "promote" facts about weak-head reduction to facts about β-reduction, and to then use them as rewrites.

`rewrs` Possibly conditional rewrites, presumably mostly of the form $P \Rightarrow R\,t_1\,t_2$. Rewrites over equality can also be included here, allowing useful additional facts to be included. For example, when working with the λ-calculus, one might include both the rewrite for K above, as well as the definition of substitution.

The application of this function to a simpset `ss` will produce an augmented `ss` that has all of `ss`'s existing behaviours, as well as the ability to rewrite with the given relation.

5.6 Efficient Applicative Order Reduction—computeLib

Section 4.1 and Section 4.5 show the ability of HOL to represent many of the standard constructs of functional programming. If one then wants to 'run' functional programs on arguments, there are several choices. First, one could apply the simplifier, as demonstrated in Section 5.5. This allows all the power of the rewriting process to be brought to bear, including, for example, the application of decision procedures to prove constraints on conditional rewrite rules. Second, one could write the program, and all the programs it transitively depends on, out to a file in a suitable concrete syntax, and invoke a compiler or interpreter. This functionality is available in HOL via use of `EmitML.exportML`.

Third, computeLib can be used. This library supports call-by-value evaluation of HOL functions by deductive steps. In other words, it is quite similar to having an ML interpreter inside the HOL logic, working by forward inference. When used in this way, functional programs can be executed more quickly than by using the simplifier.

The most accessible entry-points for using the computeLib library are the conversion EVAL and its tactic counterpart EVAL_TAC. These depend on an internal database that stores function definitions. In the following example, loading sortingTheory augments this database with relevant definitions, that of Quicksort (QSORT) in particular, and then we can evaluate QSORT on a concrete list.

1
```
- load "sortingTheory";

- EVAL ``QSORT (<=) [76;34;102;3;4]``;
> val it = |- QSORT $<= [76; 34; 102; 3; 4] = [3; 4; 34; 76; 102] : thm
```

Often, the argument to a function has no variables: in that case application of EVAL ought to return a ground result, as in the above example. However, EVAL can also evaluate functions on arguments with variables—so-called *symbolic* evaluation—and in that case, the behaviour of EVAL depends on the structure of the recursion equations. For example, in the following session, if there is sufficient information in the input, symbolic execution can deliver an interesting result. However, if there is not enough information in the input to allow the algorithm any traction, no expansion will take place.

2
```
- EVAL ``REVERSE [u;v;w;x;y;z]``;
> val it = |- REVERSE [u; v; w; x; y; z] = [z; y; x; w; v; u] : thm

- EVAL ``REVERSE alist``;
> val it = |- REVERSE alist = REVERSE alist : thm
```

5.6.1 Dealing with divergence

The major difficulty with using EVAL is termination. All too often, symbolic evaluation with EVAL will diverge, or generate enormous terms. The usual cause is conditionals with variables in the test. For example, the following definition is provably equal to FACT,

3
```
Define `fact n = if n=0 then 1 else n * fact (n-1)`;
> val it = |- fact n = (if n = 0 then 1 else n * fact (n - 1)) : thm
```

But the two definitions evaluate completely differently.

4
```
EVAL ``FACT n``;
> val it = |- FACT n = FACT n : thm

- EVAL ``fact n``;
<.... interrupt key struck ...>
> Interrupted.
```

The primitive-recursive definition of FACT does not expand at all, while the destructor-style recursion of fact never stops expanding. A rudimentary monitoring facility shows the behaviour, first on a ground argument, then on a symbolic argument.

5

```
- val [fact] = decls "fact";
- computeLib.monitoring := SOME (same_const fact);

- EVAL ‘‘fact 4‘‘;
fact 4 = (if 4 = 0 then 1 else 4 * fact (4 - 1))
fact 3 = (if 3 = 0 then 1 else 3 * fact (3 - 1))
fact 2 = (if 2 = 0 then 1 else 2 * fact (2 - 1))
fact 1 = (if 1 = 0 then 1 else 1 * fact (1 - 1))
fact 0 = (if 0 = 0 then 1 else 0 * fact (0 - 1))
> val it = |- fact 4 = 24 : thm

- EVAL ‘‘fact n‘‘;
fact n = (if n = 0 then 1 else n * fact (n - 1))
fact (n - 1) = (if n - 1 = 0 then 1 else (n - 1) * fact (n - 1 - 1))
fact (n - 1 - 1) =
(if n - 1 - 1 = 0 then 1 else (n - 1 - 1) * fact (n - 1 - 1 - 1))
fact (n - 1 - 1 - 1) =
(if n - 1 - 1 - 1 = 0 then
   1
 else
   (n - 1 - 1 - 1) * fact (n - 1 - 1 - 1 - 1))
   .
   .
   .
```

In each recursive expansion, the test involves a variable, and hence cannot be reduced to either T or F. Thus, expansion never stops.

Some simple remedies can be adopted in trying to deal with non-terminating symbolic evaluation.

- RESTR_EVAL_CONV behaves like EVAL except it takes an extra list of constants. During evaluation, if one of the supplied constants is encountered, it will not be expanded. This allows evaluation down to a specified level, and can be used to cut-off some looping evaluations.

- set_skip can also be used to control evaluation. See the *REFERENCE* entry for CBV_CONV for discussion of set_skip.

Custom evaluators For some problems, it is desirable to construct a customized evaluator, specialized to a fixed set of definitions. The compset type found in computeLib is the type of definition databases. The functions new_compset, bool_compset, add_funs,

and add_convs provide the standard way to build up such databases. Another quite useful compset is reduceLib.num_compset, which may be used for evaluating terms with numbers and booleans. Given a compset, the function CBV_CONV generates an evaluator: it is used to implement EVAL. See *REFERENCE* for more details.

Dealing with Functions over Peano Numbers Functions defined by pattern-matching over Peano-style numbers are not in the right format for EVAL, since the calculations will be asymptotically inefficient. Instead, the same definition should be used over numerals, which is a positional notation described in Section 3.3.3. However, it is preferable for proofs to work over Peano numbers. In order to bridge this gap, the function numLib.SUC_TO_NUMERAL_DEFN_CONV is used to convert a function over Peano numbers to one over numerals, which is the format that EVAL prefers. Define will automatically call SUC_TO_NUMERAL_DEFN_CONV on its result.

Storing definitions Define automatically adds its definition to the global compset used by EVAL and EVAL_TAC. However, when Hol_defn is used to define a function, its defining equations are not added to the global compset until tprove is used to prove the termination constraints. Moreover, tprove does not add the definition persistently into the global compset. Therefore, one must use add_persistent_funs in a theory to be sure that definitions made by Hol_defn are available to EVAL in descendant theories. Another point: one must call add_persistent_funs before export_theory is called.

5.7 Arithmetic Libraries—numLib, intLib and realLib

Each of the arithmetic libraries of HOL provide a suite of definitions and theorems as well as automated inference support.

numLib The most basic numbers in HOL are the natural numbers. The numLib library encompasses the theories numTheory, prim_recTheory, arithmeticTheory, and numeralTheory. This library also incorporates an evaluator for numeric expression from reduceLib and a decision procedure for linear arithmetic ARITH_CONV. The evaluator and the decision procedure are integrated into the simpset arith_ss used by the simplifier. As well, the linear arithmetic decision procedure can be directly invoked through DECIDE and DECIDE_TAC, both found in bossLib.

intLib The intLib library comprises integerTheory, an extensive theory of the integers, plus two decision procedures for full Presburger arithmetic. These are available as intLib.COOPER_CONV and intLib.ARITH_CONV. These decision procedures are able to deal with linear arithmetic over the integers and the natural numbers, as well as

dealing with arbitrary alternation of quantifiers. The ARITH_CONV procedure is an implementation of the Omega Test, and seems to generally perform better than Cooper's algorithm. There are problems for which this is not true however, so it is useful to have both procedures available.

realLib The realLib library provides a foundational development of the real numbers and analysis. See Section 3.3.6 for a quick description of the theories. Also provided is a theory of polynomials, in polyTheory. A decision procedure for linear arithmetic on the real numbers is also provided by realLib, under the name REAL_ARITH_CONV and REAL_ARITH_TAC.

5.8 Bit Vector Library—wordsLib

The library wordsLib provides tool support for bit-vectors, this includes facilities for: evaluation, parsing, pretty-printing and simplification.

5.8.1 Evaluation

The library wordsLib should be loaded when evaluating ground bit-vector terms. This library provides a *compset* words_compset, which can be used in the construction of custom *compsets* and conversions.

1
```
- load "wordsLib";
> val it = () : unit

- EVAL ``8w + 9w:word4``;
> val it = |- 8w + 9w = 1w : thm
```

Note that a type annotation is used here to designate the word size. When the word size is represented by a type variable (i.e. for arbitrary length words), evaluation may give partial or unsatisfactory results.

5.8.2 Parsing and pretty-printing

Words can be parsed in binary, decimal and hexadecimal. For example:

2
```
- ``0b111011w : word8``;
> val it = ``58w`` : term

- ``0x3Aw : word8``;
> val it = ``58w`` : term
```

It is possible to parse octal numbers, but this must be enabled first by setting the reference base_tokens.allow_octal_input to true. For example:

```
- ``072w : word8``;                                                    3
> val it = ``72w`` : term

- base_tokens.allow_octal_input:=true;
> val it = () : unit

- ``072w : word8``;
> val it = ``58w`` : term
```

Words can be pretty-printed using the standard number bases. For example, the function `wordsLib.output_words_as_bin` will select binary format:

```
- wordsLib.output_words_as_bin();                                      4
> val it = () : unit

- EVAL ``($FCP ODD):word16``;
> val it = |- $FCP ODD = 0b1010101010101010w : thm
```

The function `output_words_as` is more flexible and allows the number base to vary depending on the word length and numeric value. The default pretty-printer (installed when loading `wordsLib`) prints small values in decimal and large values in hexadecimal. The function `output_words_as_oct` will automatically enable the parsing of octal numbers.

The trace variable `"word printing"` provides an alternative method for changing the output number base — it is particularly suited to temporarily selecting a number base, for example:

```
- Feedback.trace ("word printing", 1) Parse.print_term ``32w``;        5
<<HOL message: inventing new type variable names: 'a>>
0b100000w> val it = () : unit
```

The choices are as follows: 0 (default) – small numbers decimal, large numbers hexadecimal; 1 – binary; 2 – octal; 3 – decimal; and 4 – hexadecimal.

5.8.2.1 Types

You may have noticed that *:word4* and *:word8* have been used as convenient parsing abbreviations for *:bool[4]* and *:bool[8]* — this facility is available for many standard word sizes. Users wishing to use this notation for non-standard word sizes can use the function `wordsLib.mk_word_size`:

```
- ``:word15``;                                                         6
! Uncaught exception:
! HOL_ERR

- wordsLib.mk_word_size 15;
> val it = () : unit

- ``:word15``;
> val it = ``:bool[15]`` : hol_type
```

5.8.2.2 Operator overloading

The symbols for the standard arithmetic operations (addition, subtraction and multiplication) are overloaded with operators from other standard theories, i.e. for the natural, integer, rational and real numbers. In many cases type inference will resolve overloading, however, in some cases this is not possible. The choice of operator will then depend upon the order in which theories are loaded. To change this behaviour the functions `wordsLib.deprecate_word` and `wordsLib.prefer_word` are provided. For example, in the following session, the selection of word operators is deprecated:

```
- type_of ``a + b``;                                                    7
<<HOL message: more than one resolution of overloading was possible>>
<<HOL message: inventing new type variable names: 'a>>
> val it = ``:bool['a]`` : hol_type

- wordsLib.deprecate_word();
> val it = () : unit

- type_of ``a + b``;
<<HOL message: more than one resolution of overloading was possible>>
> val it = ``:num`` : hol_type
```

In the above, natural number addition is chosen in preference to word addition. Conversely, words are preferred over the integers below:

```
- load "intLib"; ...                                                    8

- type_of ``a + b``;
<<HOL message: more than one resolution of overloading was possible>>
> val it = ``:int`` : hol_type

- wordsLib.prefer_word();
> val it = () : unit
- type_of ``a + b``;
<<HOL message: more than one resolution of overloading was possible>>
<<HOL message: inventing new type variable names: 'a>>
> it = ``:bool['a]`` : hol_type
```

Of course, type annotations could have been added to avoid this problem entirely. Note that, unlike `deprecate_int`, the function `deprecate_word` does not remove the overloadings, it simply lowers their priority.

5.8.2.3 Guessing word lengths

It can be a nuisance to add type annotations when specifying the return type for operations such as: `word_extract`, `word_concat`, `concat_word_list` and `word_replicate`.

This is because there is often a "standard" length that could be guessed, e.g. concatenation usually sums the constituent word lengths. A facility for word length guessing is controlled by the reference `wordsLib.guessing_word_lengths`, which is false by default. The guesses are made during a post-processing step that occurs after the application of `Parse.Term`. This is demonstrated below.

9

```
- wordsLib.guessing_word_lengths:=true;
> val it = () : unit

- ``concat_word_list [(4 >< 1) (w:word32); w2; w3]``;
<<HOL message: inventing new type variable names: 'a, 'b>>
<<HOL message: assigning word length: 'a <- 4>>
<<HOL message: assigning word length: 'b <- 12>>
> val it =
    ``concat_word_list [(4 >< 1) w; w2; w3]``
     : term
```

In the example above, word length guessing is turned on. Two guesses are made: the extraction is expected to give a four bit word, and the concatenation gives a twelve bit word (3×4). If non-standard numeric lengths are required then type annotations can be added to avoid guesses being made. With guessing turned off the result types would remain as invented type variables, i.e. as alpha and beta above.

5.8.3 Simplification and conversions

The following *simpset* fragments are provided:

`SIZES_ss` evaluates a group of functions that operate over numeric types, such as `dimindex` and `dimword`.

`BIT_ss` tries to simplify occurrences of the function `BIT`.

`WORD_LOGIC_ss` simplifies bitwise logic operations.

`WORD_ARITH_ss` simplifies word arithmetic operations. Subtraction is replaced with multiplication by -1.

`WORD_SHIFT_ss` simplifies shift operations.

`WORD_ss` contains all of the above fragments, and also does some extra ground term evaluation. This fragment is added to `srw_ss`.

`WORD_ARITH_EQ_ss` simplifies ``` ``a = b`` ``` to ``` ``a - b = 0w`` ```.

`WORD_BIT_EQ_ss` aggressively expands non-arithmetic bit-vector operations into Boolean expressions. (Should be used with care – it includes `fcpLib.FCP_ss`.)

WORD_EXTRACT_ss simplification for a variety of operations: word-to-word conversions; concatenation; shifts and bit-field extraction. Can be used in situations where WORD_BIT_EQ_ss is unsuitable.

WORD_MUL_LSL_ss simplifies multiplication by a word literal into a sum of partial products.

Many of these *simpset* fragments have corresponding conversions. For example, the conversion WORD_ARITH_CONV is based on WORD_ARITH_EQ_ss, however, it does some extra work to ensure that ``a = b`` and ``b = a`` convert into the same expression. Therefore, this conversion is suited to reasoning about the equality of arithmetic word expressions.

The behaviour of the fragments listed above are demonstrated using the following function:

10
```
- fun conv ss = SIMP_CONV (pure_ss++ss) [];
> val conv = fn : ssfrag -> term -> thm
```

The following session demonstrates SIZES_ss:

11
```
- conv wordsLib.SIZES_ss ``dimindex(:12)``;
> val it = |- dimindex (:12) = 12 : thm

- conv wordsLib.SIZES_ss ``FINITE univ(:32)``;
> val it = |- FINITE univ(:32) <=> T : thm
```

The fragment BIT_ss converts BIT into membership test over a set of (high) bit positions:

12
```
- conv wordsLib.BIT_ss ``BIT 3 5``;
> val it = |- BIT 3 5 <=> (3 = 0) \/ (3 = 2) : thm

- conv wordsLib.BIT_ss ``BIT i 123``;
> val it = |- BIT i 123 <=> i IN {0; 1; 3; 4; 5; 6} :
  thm
```

This simplification provides some support for reasoning about bitwise operations over arbitrary word lengths. The arithmetic, logic and shift fragments help tidy up basic word expressions:

```
- conv wordsLib.WORD_LOGIC_ss ‘‘a && 12w || 11w && a‘‘;                    13
<<HOL message: inventing new type variable names: 'a>>
> val it =
    |- a && 12w || 11w && a = 15w && a :
  thm

- conv wordsLib.WORD_ARITH_ss ‘‘3w * b + a + 2w * b - a * 4w:word2‘‘;
> val it =
    |- 3w * b + a + 2w * b - a * 4w = a + b
     : thm

- conv wordsLib.WORD_SHIFT_ss ‘‘0w << 12 + a >>> 0 + b << 2 << 3‘‘;
<<HOL message: inventing new type variable names: 'a>>
> val it =
    |- 0w << 12 + a >>> 0 + b << 2 << 3 = 0w + a + b << (2 + 3)
     : thm
```

The remaining fragments are not included in `wordsLib.WORD_ss` or `srw_ss`. The bit equality fragment is demonstrated below.

```
- SIMP_CONV (std_ss++wordsLib.WORD_BIT_EQ_ss) [] ‘‘a && b = ~0w : word2‘‘;  14
> val it =
    |- (a && b = ~0w) <=> (a ' 1 /\ b ' 1) /\ a ' 0 /\ b ' 0
     : thm
```

The extract fragment is useful for reasoning about bit-field operations and is best used in combination with `wordsLib.SIZES_ss` or `wordsLib.WORD_ss`, for example:

```
- SIMP_CONV (std_ss++wordsLib.SIZES_ss++wordsLib.WORD_EXTRACT_ss) []      15
   ‘‘(4 -- 1) ((a:word3) @@ (b:word2)) : word5‘‘;
> val it =
    |- (4 -- 1) (a @@ b) = (2 >< 0) a << 1 || (1 >< 1) b
     : thm
```

Finally, the fragment `WORD_MUL_LSL_ss` is demonstrated below.

```
- conv wordsLib.WORD_MUL_LSL_ss ‘‘5w * a : word8‘‘;                       16
> val it = |- 5w * a = a << 2 + a : thm
```

Rewriting with the theorem `wordsTheory.WORD_MUL_LSL` provides an means to undo this simplification, for example:

```
- SIMP_CONV (std_ss++wordsLib.WORD_ARITH_ss) [wordsTheory.WORD_MUL_LSL]   17
    ‘‘a << 2 + a : word8‘‘;
> val it = |- a << 2 + a = 5w * a : thm
```

Obviously, without adding safeguards, this rewrite theorem cannot be deployed when used in combination with the `WORD_MUL_LSL_ss` fragment.

5.8.3.1 Decision procedures

A decision procedure for words is provided in the form of `blastLib.BBLAST_PROVE`. This procedure uses *bit-blasting* — converting word expressions into propositions and then using a SAT solver to decide the goal.[9] This approach is reasonably general and can tackle a wide range of bit-vector problems. However, there are some limitations: the approach only works for constant word lengths, linear arithmetic (multiplication by literals) and for shifts and bit-field extractions with respect to literal values. Also note that some problems will be potentially slow to prove, e.g. when word sizes are large and/or when there are many nested additions (perhaps through multiplication).

The following examples show `BBLAST_PROVE` in use:

```
- blastLib.BBLAST_PROVE ‘‘a + 2w <+ 4w = a <+ 2w \/ 13w <+ a :word4‘‘;                18
> val it =
    |- a + 2w <+ 4w <=> a <+ 2w \/ 13w <+ a
     : thm

- blastLib.BBLAST_PROVE ‘‘w2w (a:word8) <+ 256w : word16‘‘;
> val it = |- w2w a <+ 256w : thm
```

The decision procedure `BBLAST_PROVE` is based on the conversion `BBLAST_CONV`. This conversion can be used to convert bit-vector problems into a propositional form; for example:

```
- blastLib.BBLAST_CONV ‘‘(((a : word16) + 5w) << 3) ’ 5‘‘;                          19
> val it =
   |- ((a + 5w) << 3) ’ 5 <=> (~a ’ 2 <=> ~(a ’ 1 /\ a ’ 0))
   : thm
```

There are also bit-blasting tactics: `BBLAST_TAC` and `FULL_BBLAST_TAC`; with only the latter making use of goal assumptions.

5.9 The `HolSat` Library

The purpose of `HolSatLib` is to provide a platform for experimenting with combinations of theorem proving and SAT solvers. Only black box functionality is provided at the moment; an incremental interface is not available.

`HolSatLib` provides a function `SAT_PROVE` for propositional satisfiability testing and for proving propositional tautologies. It uses an external SAT solver (currently MiniSat 1.14p) to find an unsatisfiability proof or satisfying assignment, and then reconstructs the proof or checks the assignment deductively in HOL.

Alternatively, the function `SAT_ORACLE` has the same behaviour as `SAT_PROVE` but asserts the result of the solver without proof. The theorem thus asserted is tagged with

[9] This approach enables counter-examples to be given when a goal's negation is satisfiable.

"HolSatLib" to indicate that it is unchecked. Since proof reconstruction can be expensive, the oracle facility can be useful during prototyping, or if proof is not required.

The following example illustrates the use of HolSatLib for proving propositional tautologies:

1
```
- load "HolSatLib"; open HolSatLib;
(* output omitted *)
> val it = () : unit

- show_tags := true;
> val it = () : unit

- SAT_PROVE ``(a ==> b) /\ (b ==> a) = (a=b)``;
> val it = [oracles: DISK_THM] [axioms: ] []
           |- (a ==> b) /\ (b ==> a) = (a = b) : thm

- SAT_PROVE ``(a ==> b) ==> (a=b)``
   handle HolSatLib.SAT_cex th => th;
> val it = [oracles: DISK_THM] [axioms: ] []
           |- ~a /\ b ==> ~((a ==> b) ==> (a = b)) : thm

- SAT_ORACLE ``(a ==> b) /\ (b ==> a) = (a=b)``;
> val it = [oracles: DISK_THM, HolSatLib] [axioms: ] []
           |- (a ==> b) /\ (b ==> a) = (a = b) : thm
```

Setting show_tags to true makes the HOL top-level print theorem tags. The DISK_THM oracle tag has nothing to do with HolSatLib. It just indicates the use of theorems from HOL libraries read in from permanent storage.

Note that in the case where the putative tautology has a falsifying interpretation, a counter-model can be obtained by capturing the special exception SAT_cex, which contains a theorem asserting the counter-model.

The next example illustrates using HolSatLib for satisfiability testing. The idea is to negate the target term before passing it to HolSatLib.

2
```
- SAT_PROVE ``~((a ==> b) ==> (a=b))``
   handle HolSatLib.SAT_cex => th;
> val it = [oracles: DISK_THM ] [axioms: ] []
           |- a /\ ~b ==> ~~((a ==> b) ==> (a = b)) : thm

- SAT_PROVE ``~(a /\ ~a)``;
> val it = [oracles: DISK_THM ] [axioms: ] []
           |- ~(a /\ ~a) : thm
```

As expected, if the target term is unsatisfiable we get a theorem saying as much.

HolSatLib can only handle purely propositional terms (atoms must be propositional variables or constants) involving the usual propositional connectives as well as Boolean-valued conditionals. If you wish to prove tautologies that are instantiations of propositional terms, use tautLib (see §5.9.1 below).

If MiniSat failed to build when HOL was built, or proof replay fails for some other reason, SAT_PROVE falls back to a DPLL-based propositional tautology prover implemented in SML, due to Michael Norrish (see the HOL Tutorial). HolSatLib prints out a warning if this happens. On problems with more than a thousand or so clauses (in conjunctive normal form (CNF)), the SML prover will likely take too long to be of any use.

HolSatLib will delete temporary files generated by the SAT solver, such as the proof file and any statistics. This is to avoid accumulating thousands of possibly large files. Currently HolSatLib has only been tested on Linux, and on Windows XP using MinGW.

5.9.1 tautLib

tautLib predates HolSatLib by over a decade. It used a Boolean case analysis algorithm suggested by Tom Melham and implemented by R. J. Boulton. This algorithm has since been superseded and the functions in the tautLib signature now act as wrappers around calls to HolSatLib. However, the wrappers are able to provide the following extra functionality on top of HolSatLib:

1. They can handle top level universal quantifiers.
2. They can reason about (the propositional structure of) terms that are instances of purely propositional terms. This is done by a preprocessing step that replaces each unique instantiation with a fresh propositional variable.

For details, see the source file src/taut/tautLib.sml which contains comprehensive comments. Note however that the extra functionality in tautLib was not engineered for very large problems and can become a performance bottleneck.

5.9.2 Support for other SAT solvers

The ZChaff SAT solver has a proof production mode and is supported by HolSatLib. However, the ZChaff end user license is not compatible with the HOL license, so we are unable to distribute it with HOL. If you wish to use ZChaff, download and unpack it in the directory src/HolSat/sat_solvers/ under the main HOL directory, and compile it with proof production mode enabled (which is not the default). This should create a binary zchaff in the directory src/HolSat/sat_solvers/zchaff/. ZChaff can now be used as the external proof engine instead of MiniSat, by using the HolSatLib functions described above, prefixed with a "Z", e.g., ZSAT_PROVE.

A file resolve_trace may be created in the current working directory, when working with ZChaff. This is the proof trace file produced by ZChaff, and is hardwired.

Other SAT solvers are currently not supported. If you would like such support to be added for your favourite solver, please send a feature request via http://github.com/mn200/HOL.

5.9.3 The general interface

The functions described above are wrappers for the function `GEN_SAT`, which is the single entry point for `HolSatLib`. `GEN_SAT` can be used directly if more flexibility is required. `GEN_SAT` takes a single argument, of type `sat_config`, defined in `satConfig.sml`. This is an opaque record type, currently containing the following fields:

1. `term : Term.term`

   The input term.

2. `solver : SatSolvers.sat_solver`

   The external SAT solver to use. The default is `SatSolvers.minisatp`. If ZChaff is installed (see §5.9.2), then `SatSolvers.zchaff` may also be used.

3. `infile : string option`

   The name of a file in DIMACS format.[10]. Overrides `term` if set. The input term is instead read from the file.

4. `proof : string option`

   The name of a proof trace file. Overrides `solver` if set. The file must be in the native format of `HolSatLib`, and must correspond to a proof for `infile`, which must also be set. The included version of MiniSat has been modified to produce proofs in the native format, and ZChaff proofs are translated to this format using the included proof translator `src/HolSat/sat_solvers/zc2hs` (type `zc2hs -h` for usage help). `zc2hs` is used internally by `ZSAT_PROVE` etc.

5. `is_cnf : bool`

   If true then the input term is expected to be a negated CNF term. This is set automatically if `infile` is set. Typically a user will never need to modify this field directly.

6. `is_proved : bool`

   If true then HOL will prove the SAT solver's results.

A special value `base_config : sat_config` is provided for which the term is `T`, the solver is MiniSat, the options are unset, the CNF flag is false and the proof flag is true. This value can be inspected and modified using getter and setter functions provided in `src/HolSat/satConfig.sig`. For example, to invoke ZChaff (assuming it is installed), on a file `unsat.cnf` containing an unsatisfiable problem, we do:

[10] `http://www.satlib.org/Benchmarks/SAT/satformat.ps`

```
- open satConfig;                                                                    3
(* output omitted *)

- val c = (set_infile "unsat.cnf" o set_solver SatSolvers.zchaff) base_config;
> val c = <sat_config> : sat_config

- GEN_SAT c;
> val it = [oracles: DISK_THM ] [axioms: ] []
           |- ~<unsat.cnf term here> : thm
```

If the problem were satisfiable, the model can be captured via exception, as shown earlier.

Normally, `HolSatLib` will delete the files generated by the SAT solver, such as the output proof, counter-model, and result status. However, if `infile` is set, the files are not deleted, in case they are required elsewhere.

5.9.4 Notes

On Linux and MacOS, `g++` must be installed on the system for MiniSat and `zc2hs` to build.

Temporary files are generated using the Moscow ML function `FileSys.tmpName`. This usually writes to the standard temporary file space on the operating system. If that file space is full, or if it is inaccessible for some other reason, `HolSatLib` calls may fail mysteriously.

The function `dimacsTools.readDimacs` *file* reads a DIMACS format file and returns a CNF HOL term corresponding to the SAT problem in the file named by *file*. Since DIMACS uses numbers to denote variables, and numbers are not legal identifiers in HOL, each variable number is prefixed with the string "`v`". This string is defined in the reference variable `dimacsTools.prefix` and can be changed if required. This function can be used independently of `HolSatLib` to read in DIMACS format files.

5.10 The `HolQbf` Library

`HolQbfLib` provides a rudimentary platform for experimenting with combinations of theorem proving and Quantified Boolean Formulae (QBF) solvers. `HolQbfLib` was developed as part of a research project on *Expressive Multi-theory Reasoning for Interactive Verification* (EPSRC grant EP/F067909/1) from 2008 to 2011. It is loosely inspired by `HolSatLib` (Section 5.9), and has been described in parts in the following publications:

- Tjark Weber: *Validating QBF Invalidity in HOL4*. In Matt Kaufmann and Lawrence C. Paulson, editors, Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11–14, 2010. Proceedings, volume 6172 of Lecture Notes in Computer Science, pages 466–480. Springer, 2010.

- Ramana Kumar and Tjark Weber: *Validating QBF Validity in HOL4*. In Marko C. J. D. van Eekelen, Herman Geuvers, Julien Schmaltz, and Freek Wiedijk, editors, Interactive Theorem Proving, Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22–25, 2011. Proceedings, volue 6898 of Lecture Notes in Computer Science, pages 168–183. Springer, 2011.

`HolQbfLib` uses an external QBF solver, Squolem, to decide Quantified Boolean Formulae.

5.10.1 Installing Squolem

`HolQbfLib` has been tested with (the x86 Linux version of) Squolem 2.02 (release date 2010-11-10). This is Squolem's latest version at the time of writing. Squolem can be obtained from `http://www.cprover.org/qbv/download.html`. After installation, you must make the executable available as `squolem2`, e.g., by placing it into a folder that is in your `$PATH`. This name is currently hard-coded: there is no configuration option to tell HOL about the location and name of the Squolem executable.

5.10.2 Interface

The library provides four functions, each of type `term -> thm`, to invoke Squolem: `decide`, `decide_prenex`, `disprove`, and `prove`. These are defined in the `HolQbfLib` structure, which is the library's main entry point.

Calling `prove` ϕ will invoke Squolem on the QBF ϕ to establish its validity. If this succeeds, `prove` will then validate the certificate of validity generated by Squolem in HOL to return a theorem $\vdash \phi$.

Similarly, calling `disprove` ϕ will invoke Squolem to establish that ϕ is invalid. If this succeeds, `disprove` will then validate the certificate of invalidity generated by Squolem in HOL to return a theorem $\phi \vdash \bot$.

`decide_prenex` ϕ combines the functionality of `prove` and `disprove` into a single function. It will invoke Squolem on ϕ and return either $\vdash \phi$ or $\phi \vdash \bot$, depending on Squolem's answer.

Finally, `decide` does the same job as `decide_prenex` but accepts QBFs in a less restricted form. Restrictions on ϕ are described below.

1
```
- load "HolQbfLib";
metis: r[+0+3]#
r[+0+6]#
> val it = () : unit

- open HolQbfLib;
> val decide = fn: term -> thm
val decide_prenex = fn: term -> thm
val disprove = fn: term -> thm
val prove = fn: term -> thm

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

- decide ``?x. x``;
<<HOL message: HolQbfLib: calling external command
  'squolem2 -c /tmp/filedH1K2x >/dev/null 2>&1'>>
> val it =  [] |- ?x. x: thm

- decide ``(?y. x \/ y) ==> ~x``;
> val it = [!x. (?y. x \/ y) ==> ~x] |- F: thm

- decide ``~(?x. x ==> y) \/ (?x. y ==> x)``;
<<HOL message: HolQbfLib: calling external command
  'squolem2 -c /tmp/fileyap3oD >/dev/null 2>&1'>>
> val it = [] |- ~(?x. x ==> y) \/ ?x. y ==> x: thm

- decide_prenex ``!x. ?y. x /\ y``;
<<HOL message: HolQbfLib: calling external command
  'squolem2 -c /tmp/fileZAGj4m >/dev/null 2>&1'>>
> val it = [!x. ?y. x /\ y] |- F : thm

- disprove ``!x. ?y. x /\ y``;
<<HOL message: HolQbfLib: calling external command
  'squolem2 -c /tmp/file0Pw2Tg >/dev/null 2>&1'>>
> val it = [!x. ?y. x /\ y] |- F : thm

- prove ``?x. x``;
<<HOL message: HolQbfLib: calling external command
  'squolem2 -c /tmp/fileKi4Lkz >/dev/null 2>&1'>>
- val it =  [] |- ?x. x: thm
```

Supported subset of higher-order logic The argument given to `decide` must be a Boolean term built using only conjunction, disjunction, implication, negation, universal/existential quantification, and variables. Free variables are considered universally quantified. Every quantified variable must occur.

The argument given to the other functions must be a QBF in prenex form, i.e., a term

of the form $Q_1 x_1 . Q_2 x_2 . \ldots Q_n x_n . \phi$, where

- $n \geq 0$,
- each Q_i is an (existential or universal) quantifier,
- Q_n is the existential quantifier,
- each x_i is a Boolean variable,
- ϕ is a propositional formula in CNF, i.e., a conjunction of disjunctions of (possibly negated) Boolean variables,
- ϕ must actually contain each x_i,
- all x_i must be distinct, and
- ϕ does not contain variables other than $x_1, \ldots, x_n$.

The behavior is undefined if any of these restrictions are violated.

Support for the QDIMACS file format The QDIMACS standard defines an input file format for QBF solvers. `HolQbfLib` provides a structure `QDimacs` that implements (parts of) the QDIMACS standard, version 1.1 (released on December 21, 2005), as described at `http://www.qbflib.org/qdimacs.html`. The `QDimacs` structure does not require Squolem (or any other QBF solver) to be installed.

`QDimacs.write_qdimacs_file path` ϕ creates a QDIMACS file (with name `path`) that encodes the QBF ϕ, where ϕ must meet the requirements detailed above. The function returns a dictionary that maps each variable in ϕ to its corresponding variable index (a positive integer) used in the QDIMACS file.

`QDimacs.read_qdimacs_file f path` parses an existing QDIMACS file (with name `path`) and returns the encoded QBF as a HOL term. Since variables are only given as integers in the QDIMACS format, variables in HOL are obtained by applying `f` (which is a function of type `int -> term`) to each integer. `f` is expected to return Boolean variables only, not arbitrary HOL terms.

Tracing Tracing output can be controlled via `Feedback.set_trace "HolQbfLib"`. See the source code in `QbfTrace.sml` for possible values.

Communication between HOL and Squolem is via temporary files. These files are located in the standard temporary directory, typically `/tmp` on Unix machines. The actual file names are generated at run-time, and can be shown by setting the above tracing variable to a sufficiently high value.

The default behavior of HolQbfLib is to delete temporary files after successful invocation of Squolem. This also can be changed via the above tracing variable. If there is an error, files are retained in any case (but note that the operating system may delete temporary files automatically, e.g., when HOL exits).

5.10.3 Wishlist

The following features have not been implemented yet. Please submit additional feature requests (or code contributions) via http://github.com/mn200/HOL.

Support for other QBF solvers So far, Squolem is the only QBF solver that has been integrated with HOL. Several other QBF solvers can produce proofs, and it would be nice to offer HOL users more choice (also because Squolem's performance is not necessarily state-of-the-art anymore).

QBF solvers as a web service The need to install a QBF solver locally poses an entry barrier. It would be much more convenient to have a web server running one (or several) QBF solvers, roughly similar to the "System on TPTP" interface that G. Sutcliffe provides for first-order theorem provers (http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP).

5.11 The HolSmt library

The purpose of HolSmtLib is to provide a platform for experimenting with combinations of interactive theorem proving and Satisfiability Modulo Theories (SMT) solvers. HolSmtLib was developed as part of a research project on *Expressive Multi-theory Reasoning for Interactive Verification* (EPSRC grant EP/F067909/1) from 2008 to 2011. It is loosely inspired by HolSatLib (Section 5.9), and has been described in parts in the following publications:

- Tjark Weber: *SMT Solvers: New Oracles for the HOL Theorem Prover*. To appear in International Journal on Software Tools for Technology Transfer (STTT), 2011.
- Sascha Böhme, Tjark Weber: *Fast LCF-Style Proof Reconstruction for Z3*. In Matt Kaufmann and Lawrence C. Paulson, editors, Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11–14, 2010. Proceedings, volume 6172 of Lecture Notes in Computer Science, pages 179–194. Springer, 2010.

HolSmtLib uses external SMT solvers to prove instances of SMT tautologies, i.e., formulas that are provable using (a combination of) propositional logic, equality reasoning,

linear arithmetic on integers and reals, and decision procedures for bit vectors and arrays. The supported fragment of higher-order logic varies with the SMT solver used, and is discussed in more detail below. At least for Yices, it is a superset of the fragment supported by `bossLib.DECIDE` (and the performance of `HolSmtLib`, especially on big problems, should be much better).

5.11.1 Interface

The library currently provides three tactics to invoke different SMT solvers, namely `YICES_TAC`, `Z3_ORACLE_TAC`, and `Z3_TAC`. These tactics are defined in the `HolSmtLib` structure, which is the library's main entry point. Given a goal (Γ, φ) (where Γ is a list of assumptions, and φ is the goal's conclusion), each tactic returns (i) an empty list of new goals, and (ii) a validation function that returns a theorem $\Gamma' \vdash \varphi$ (with $\Gamma' \subseteq \Gamma$). These tactics fail if the SMT solver cannot prove the goal.[11] In other words, these tactics solve the goal (or fail). As with other tactics, `Tactical.TAC_PROOF` can be used to derive functions of type `goal -> thm`.

For each tactic, the `HolSmtLib` structure additionally provides a corresponding function of type `term -> thm`. These functions are called `YICES_PROVE`, `Z3_ORACLE_PROVE`, and `Z3_PROVE`, respectively. Applied to a formula φ, they return the theorem $\emptyset \vdash \varphi$ (or fail).

Oracles vs. proof reconstruction `YICES_TAC` and `Z3_ORACLE_TAC` use the SMT solver (Yices and Z3, respectively) as an oracle: the solver's result is trusted. Bugs in the SMT solver or in `HolSmtLib` could potentially lead to inconsistent theorems. Accordingly, the derived theorem is tagged with an oracle tag.

`Z3_TAC`, on the other hand, performs proof reconstruction. It requests a detailed proof from Z3, which is then checked in HOL. One obtains a proper HOL theorem; no (additional) oracle tags are introduced. However, Z3's proofs do not always contain enough information to allow efficient checking in HOL; therefore, proof reconstruction may be slow or fail.

Supported subsets of higher-order logic `YICES_TAC` employs a translation into Yices's native input format. The interface supports types `bool`, `num`, `int`, `real`, `->` (i.e., function types), `prod` (i.e., tuples), fixed-width word types, inductive data types, records, and the following terms: equality, Boolean connectives (`T`, `F`, `==>`, `/\`, `\/`, negation, `if-then-else`, `bool-case`), quantifiers (`!`, `?`), numeric literals, arithmetic operators (`SUC`, `+`, `-`, `*`, `/`, unary minus, `DIV`, `MOD`, `ABS`, `MIN`, `MAX`), comparison operators (`<`, `<=`,

[11] Internally, the goal's assumptions and the *negated* conclusion are passed to the SMT solver. If the SMT solver determines that these formulas are unsatisfiable, then the (unnegated) conclusion must be provable from the assumptions.

>, >=, both on num, int, and real), function application, lambda abstraction, tuple selectors FST and SND, and various word operations.

Z3 is integrated via a more restrictive translation into SMT-LIB 2 format, described below. Therefore, Yices is typically the solver of choice at the moment (unless you need proof reconstruction, which is available for Z3 only). However, there are a few operations (e.g., specific word operations) that are supported by the SMT-LIB format, but not by Yices. See selftest.sml for further details.

Terms of higher-order logic that are not supported by the respective target solver/ translation are typically treated in one of three ways:

1. Some unsupported terms are replaced by equivalent suppported terms during a pre-processing step. For instance, all tactics first generalize the goal's conclusion by stripping outermost universal quantifiers, and attempt to eliminate certain set expressions by rewriting them into predicate applications: e.g., `y IN {x | P x}` is replaced by `P y`. The resulting term is β-normalized. Depending on the target solver, further simplifications are performed.

2. Remaining unsupported constants are treated as uninterpreted, i.e., replaced by fresh variables. This should not affect soundness, but it may render goals unprovable and lead to spurious counterexamples. To see all fresh variables introduced by the translation, you can set HolSmtLib's tracing variable (see below) to a sufficiently high value.

3. Various syntactic side conditions are currently not enforced by the translation and may result in invalid input to the SMT solver. For instance, Yices only allows *linear* arithmetic (i.e., multiplication by constants) and word-shifts by numeric literals (constants). If the goal is outside the allowed syntactic fragment, the SMT solver will typically fail to decide the problem. HolSmtLib at present only provides a generic error message in this case. Inspecting the SMT solver's output might provide further hints.

1
```
- load "HolSmtLib"; open HolSmtLib;
(* output omitted *)
> val it = () : unit

- show_tags := true;
> val it = () : unit

- YICES_PROVE ``(a ==> b) /\ (b ==> a) = (a=b)``;
> val it = [oracles: DISK_THM, HolSmtLib] [axioms: ] []
           |- (a ==> b) /\ (b ==> a) = (a = b) : thm

- Z3_ORACLE_PROVE ``(a ==> b) /\ (b ==> a) = (a=b)``;
> val it = [oracles: DISK_THM, HolSmtLib] [axioms: ] []
           |- (a ==> b) /\ (b ==> a) = (a = b) : thm

- Z3_PROVE ``(a ==> b) /\ (b ==> a) = (a=b)``;
> val it = [oracles: DISK_THM] [axioms: ] []
           |- (a ==> b) /\ (b ==> a) = (a = b) : thm
```

Support for the SMT-LIB 2 file format SMT-LIB (see `http://combination.cs.uiowa.edu/smtlib/`) is the standard input format for SMT solvers. `HolSmtLib` supports (a subset of) version 2.0 of this format. A translation of HOL terms into SMT-LIB 2 format is available in `SmtLib.sml`, and a parser for SMT-LIB 2 files (translating them into HOL types, terms, and formulas) can be found in `SmtLib_Parser.sml`, with auxiliary functions in `SmtLib_{Logics,Theories}.sml`.

The SMT-LIB 2 translation supports types `bool`, `int` and `real`, fixed-width word types, and the following terms: equality, Boolean connectives, quantifiers, numeric literals, arithmetic operators, comparison operators, function application, and various word operations. Notably, the SMT-LIB interface does *not* support type `num`, data types or records, and higher-order formulas. See the files mentioned above and the examples in `selftest.sml` for further details.

Tracing Tracing output can be controlled via `Feedback.set_trace "HolSmtLib"`. See the source code in `Library.sml` for possible values.

Communication between HOL and external SMT solvers is via temporary files. These files are located in the standard temporary directory, typically `/tmp` on Unix machines. The actual file names are generated at run-time, and can be shown by setting the above tracing variable to a sufficiently high value.

The default behavior of `HolSmtLib` is to delete temporary files after successful invocation of the SMT solver. This also can be changed via the above tracing variable. If there is an error, files are retained in any case (but note that the operating system may delete temporary files automatically, e.g., when HOL exits).

5.11.2 Installing SMT solvers

HolSmtLib has been tested with Yices 1.0.29 and Z3 2.19. Later versions may or may not work. (Yices 2 is not supported.) To use HolSmtLib, you need to install at least one of these SMT solvers on your machine. As mentioned before, Yices supports a larger fragment of higher-order logic than Z3, but proof reconstruction has been implemented only for Z3.

Yices is available for various platforms from http://yices.csl.sri.com/. After installation, you must set the environment variable $HOL4_YICES_EXECUTABLE to the name of the Yices executable, e.g., /bin/yices, before you invoke HOL.

The Z3 website, http://research.microsoft.com/en-us/um/redmond/projects/z3/, provides Windows and Linux versions of the solver. Alternatively, the Windows version can be installed on Linux and Mac OS X—see the instructions at http://www4.in.tum.de/~boehmes/z3.html.[12] After installation, you must set the environment variable $HOL4_Z3_EXECUTABLE to the name of the Z3 executable, e.g., /bin/z3, before you invoke HOL.

It should be relatively straightforward to integrate other SMT solvers that support the SMT-LIB 2 input format as oracles. However, this will involve a (typically small) amount of Standard ML programming, e.g., to interpret the solver's output. See Z3.sml for some relevant code.

5.11.3 Wishlist

The following features have not been implemented yet. Please submit additional feature requests (or code contributions) via http://github.com/mn200/HOL.

Counterexamples For satisfiable input formulas, SMT solvers typically return a satisfying assignment. This assignment could be displayed to the HOL user as a counterexample. It could also be turned into a theorem, similar to the way HolSatLib treats satisfying assignments.

Proof reconstruction for other SMT solvers Proof reconstruction has been implemented only for Z3. Several other SMT solvers can produce proofs, and it would be nice to offer HOL users more choice. However, in the absence of a standard proof format for SMT solvers, it is perhaps not worth the implementation effort.

Support for Z3's SMT-LIB extensions Z3 supports extensions of the SMT-LIB language, e.g., data types. HolSmtLib does not utilize these extensions yet when calling

[12] Later versions of Z3 than 2.19 are available for Mac OS X directly, but not supported by HOL.

Z3. This would require the translation for Z3 to be distinct from the generic SMT-LIB translation.

SMT solvers as a web service The need to install an SMT solver locally poses an entry barrier. It would be much more convenient to have a web server running one (or several) SMT solvers, roughly similar to the "System on TPTP" interface that G. Sutcliffe provides for first-order theorem provers (`http://www.cs.miami.edu/~tptp/cgi-bin/SystemOnTPTP`). For Isabelle/HOL, such a web service has been installed by S. Böhme in Munich, but unfortunately it is not publicly available. Perhaps the SMT-EXEC initiative (`http://www.smtexec.org/`) could offer hardware or implementation support.

Chapter 6

Miscellaneous Features

This section describes some of the features that exist for managing the interface to the HOL system.

- The help system.
- The trace system for controlling feedback and printing.
- `Holmake`: a tool for dependency maintenance in large developments.
- Functions for counting the number of primitive inferences done in an evaluation, and timing it.
- A tool for embedding pretty-printed HOL theorems, terms and types in LaTeX documents.

6.1 Help

There are several kinds of help available in HOL, all accessible through the same incantation:

```
help <string>;
```

The kinds of help available are:

Moscow ML help. (When using Moscow ML HOL) This is uniformly excellent. Information for library routines is available, whether the library is loaded or not *via* `help "Lib"`.

HOL overview. This is a short summary of important information about HOL.

HOL help. This on-line help is intended to document all HOL-specific functions available to the user. It is very detailed and often accurate; however, it can be out-of-date, refer to earlier versions of the system, or even be missing!

HOL structure information. For most structures in the HOL source, one can get a listing of the entrypoints found in the accompanying signature. This is helpful for locating functions and is automatically derived from the system sources, so it is alway up-to-date.

Theory facts. These are automatically derived from theory files, so they are always up-to-date. The signature of each theory is available (since theories are represented by structures in HOL). Also, each axiom, definition, and theorem in the theory can be accessed by name in the help system; the theorem itself is given.

Therefore the following example queries can be made:

`help "installPP"`	Moscow ML help
`help "hol"`	HOL overview
`help "aconv"`	on-line HOL help
`help "Tactic"`	HOL source structure information
`help "boolTheory"`	theory structure signature
`help "list_Axiom"`	theory structure signature and theorem statement

6.2 The Trace System

The trace system gives the user one central interface with which to control most of HOL's many different flags, though they be scattered all over the system, and defined in different modules. These flags are typically those that determine the level to which HOL tools provide information to the user while operating. For example, a trace level of zero will usually make a tool remain completely silent while it operates. The tool may still raise an exception when it fails, but it won't also output any messages saying so.

There are three core functions, all in the `Feedback` structure:

```
traces : unit ->
         {default: int, max: int, name: string, trace_level: int} list

set_trace : string -> int -> unit
trace     : (string * int) -> ('a -> 'b) -> ('a -> 'b)
```

The `traces` function returns a list of all the traces in the system. The `set_trace` function allows the user to set a trace directly. The effect of this might be seen in a subsequent call to `traces()`. Finally, the `trace` function allows for a trace to be temporarily set while a function executes, restoring the trace to its old value when the function returns (whether normally, or with an exception).

6.3 Maintaining HOL Formalizations with Holmake

The purpose of Holmake is to maintain dependencies in a HOL source directory. A single invocation of Holmake will compute dependencies between files, (re)compile plain ML code, (re)compile and execute theory scripts, and (re)compile the resulting theory modules. Holmake does not require the user to provide any explicit dependency information themselves. Holmake can be very convenient to use, but there are some conventions and restrictions on it that must be followed, described below.

Holmake can be accessed through

```
<hol-dir>/bin/Holmake.
```

The development model that Holmake is designed to support is that there are two modes of work: theory construction and system revision. In 'theory construction' mode, the user builds up a theory by interacting with HOL, perhaps over many sessions. In 'system rebuild' mode, a component that others depend on has been altered, so all modules dependent on it have to be brought up to date. System rebuild mode is simpler so we deal with it first.

6.3.1 System rebuild

A system rebuild happens when an existing theory has been improved in some way (augmented with a new theorem, a change to a definition, etc.), or perhaps some support ML code has been modified or added to the formalization under development. The user needs to find and recompile just those modules affected by the change. This is what an invocation of Holmake does, by identifying the out-of-date modules and re-compiling and re-executing them.

6.3.2 Theory construction

To start a theory construction, some context (semantic, and also proof support) is established, typically by loading parent theories and useful libraries. In the course of building the theory, the user keeps track of the ML—which, for example, establishes context, makes definitions, builds and invokes tactics, and saves theorems—in a text file. This file is used to achieve inter-session persistence of the theory being constructed. For example, the text file resulting from session n is "use"-d to start session $n + 1$; after that, theory construction resumes.

Once the user finishes the perhaps long and arduous task of constructing a theory, the user should

1. make the script separately compilable;

2. invoke Holmake. This will (a) compile and execute the script file; and (b) compile the resulting theory file. After this, the theory file is available for use.

6.3.3 Making the script separately compilable

First, the invocation

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

should be added at the end of the file. When the script is finally executed, this call writes the theory to disk.

Second, we address a crucial environmental issue: if a theory script has been constructed using <holdir>/bin/hol, then it has been developed in an environment where some commonly used structures, e.g., Tactic, have already been loaded and opened for the user's convenience. When we wish to apply Holmake to a script developed in this way, we have to take some extra steps to ensure that the compilation environment also provides these structures. In the common case, this is simple; one must only add, at the head of the theory script, the following "boilerplate":

```
open HolKernel Parse boolLib;
```

This will duplicate the starting environment that one obtains with hol.bare and hol.bare.noquote. If the script was developed interactively with hol or hol.noquote, then one must also add

```
open bossLib
```

Now the script should be separately compilable. Invoke Holmake to check; the ML compiler will flag any unaccounted-for identifiers it finds. The user has to resolve these, either by using the 'dot' notation to locate the identifier for the compiler, or by opening the relevant module. This "compile/resolve-identifier" loop should continue until Holmake succeeds in compiling the module.

The following notes may be of some further help.

1. The filenames of theory scripts must follow the following convention: a HOL theory script for theory "x" should be named xScript.sml. When export_theory is called during an invocation of Holmake, the files xTheory.sig and xTheory.sml will be generated and then compiled.

2. In ML, modules are not allowed to include unbound top-level expressions. Hence, something like the following is not allowed:

   ```
   new_theory "ted";
   ```

To make ML happy, one must instead write something like

```
val _ = new_theory "ted";
```

This is because (due to restrictions imposed by Moscow ML) the script file is required to be an ML structure, and the contents of a structure must be *declarations*, not expressions. Indeed, one is allowed to (and often will) omit the bracketing `structure foo = struct - end` lines, but the contents of the file are still interpreted as if belonging to a structure.

3. In the interactive system, one has to explicitly `load` modules; on the other hand, the batch compiler will load modules automatically. For example, in order to execute `open Foo` (or refer to values in structure `Foo`) in the interactive system, one must first have executed `load "Foo"`. (This is on the assumption that structure `Foo` is defined in a file `Foo.sml`.) Contrarily, the batch compiler will reject files having occurrences of `load`, since `load` is only defined for the interactive system.

4. Take care not to have the string "Theory" embedded in the name of any of your files. HOL generates files containing this string, and when it cleans up after itself, it removes such files using a regular expression. This will also remove other files with names containing "Theory". For example, if, in your development directory, you had a file of ML code named `MyTheory.sml` and you were also managing a HOL development there with `Holmake`, then `MyTheory.sml` would get deleted if `Holmake clean` were invoked.

6.3.4 Summary

A complete theory construction might be performed by the following steps:

- Construct theory script, perhaps over many sessions;
- Transform script into separately compilable form;
- Invoke `Holmake` to generate the theory and compile it.

After that, the theory is usable as an ML module. This flow is demonstrated in the Euclid example of *TUTORIAL*.

Alternatively, and probably with the help of one of the editor modes,[1] one can develop a theory with a script file that is always separately compilable.

[1] There are editor modes for `emacs` and `vim`.

6.3.5 What `Holmake` doesn't do

`Holmake` only works properly on the current directory. `Holmake` will rebuild files in the current directory if something it depends on from another directory is fresher than it is, but it will not do any analysis on files in other directories.

However, one can indicate that there is a dependency on other directories by using the `-I` flag, or the `INCLUDES` variable in a `Holmakefile`. Such a specification will cause `Holmake` to look in the specified directories for other theory files that the current directory may depend on. Moreover, by default `Holmake` will recursively call itself on all those "include" directories before doing anything in the current directory. In this way, one can get a staged application of `Holmake` across multiple directories.[2]

6.3.6 `Holmake`'s command-line arguments

Like `make`, `Holmake` takes command-line arguments corresponding to the targets that the user desires to build. If there are none, then `Holmake` will attempt to build all ML modules and HOL theories it can detect in the current directory. In addition, there are three special targets that can be used:

`clean` Removes all compiled files (unless over-ridden by a make-file target of the same name, see section 6.3.7 below).

`cleanDeps` Removes all of the pre-computed dependency files. This can be an important thing to do if, for example, you have introduced a new `.sig` file on top of an existing `.sml` file.

`cleanAll` Removes all compiled files as well as all of the hidden dependency information.

Finally, users can directly affect the workings of `Holmake` with the following command-line options:

`-I <directory>` Look in specified directory for additional Moscow ML object files, including other HOL theories. This option can be repeated, with multiple `-I`'s to allow for multiple directories to be referenced. As above, directories specified in this way will also be rebuilt before the current targets are built.

`-d <file>` Ignore the given file and don't try to build it. The file may be rebuilt anyway if other files you have specified depend on it. This is useful to stop Holmake from attempting to compile files that are interactive scripts (include use of `load` or `use`, for example).

[2] See *Recursive Make Considered Harmful* by Peter Miller for why this is not ideal.

`-f <theory>` Toggles whether or not a theory should be built in "fast" mode. Fast building causes tactic proofs (invocations of `prove` and `store_thm`) to automatically succeed. This lack of soundness is marked by the `fast_proof` oracle tag. This tag will appear on all theorems proved in this way and all subsequent theorems that depend on such theorems. `Holmake`'s default is not to build in fast mode.

`--fast` Makes `Holmake`'s default be to build in fast mode (see above).

`--help` **or** `-h` Prints out a useful option summary and exits.

`--holdir <directory>` Associate this build with the given HOL directory, rather than the one this version of `Holmake` was configured to use by default.

`--holmakefile <file>` Use the given file as a make-file. See section 6.3.7 below for more on this.

`--interactive` **or** `-i` Causes the HOL code that runs when a theory building file is executed to have the flag `Globals.interactive` set to true. This will alter the diagnostic output of a number of functions within the system.

`-k` **or** `--keep-going` Causes `Holmake` to try to build all specified targets, rather than stopping as soon as one fails to build.

`--logging` Causes `Holmake` to record the times taken to build any theory files it encounters. The times are logged in a file in the current directory. The name of this file includes the time when `Holmake` completed, and when on a Unix system, the name of the machine where the job was run. If `Holmake` exits unsuccessfully, the filename is preceded by the string "`bad-`". Each line in the log-file is of the form *theory-name time-taken*, with the time recorded in seconds.

`--no_holmakefile` Do not use a make-file, even if a file called `Holmakefile` is present in the current directory.

`--no_overlay` Do not use an overlay file. All HOL builds require the presence of a special overlay file from the kernel when compiling scripts and libraries. This is not appropriate for compiling code that has no connection to HOL, so this option makes the compilation not use the overlay file. This option is also used in building the kernel before the overlay itself has been compiled.

`--no_prereqs` Do not recursively attempt to build "include" directories before working in the current directory.

`--no_sigobj` Do not link against HOL system's directory of HOL system files. Use of this option goes some way towards turning `Holmake` into a general Moscow ML `make`

system. However, it will still attempt to do "HOL things" with files whose names end in `Script` and `Theory`. This option implies `--no_overlay`.

`--overlay <file>` Use the given file as the overlay rather than the default.

`--qof` Standing for "quit on failure". If a tactic fails to prove a theorem, quit the build. The default is to use `mk_thm` to assert that the failed goal is true so that the build can continue and other theorems proved.

`--quiet` Minimise the amount of output produced by `Holmake`. Fatal error messages will still be written to the standard error stream. Note that other programs called by `Holmake` will not be affected.

`--rebuild_deps` **or** `-r` Forces `Holmake` to always rebuild the dependency information for files it examines, whether or not it thinks it needs to. This option is implemented by having `Holmake` wipe all of its dependency cache (as per the `cleanDeps` option above) before proceeding with the build.

`Holmake` should never exit with the Moscow ML message "Uncaught exception". Such behaviour is a bug, please report it!

6.3.7 Using a make-file with `Holmake`

`Holmake` will use a make-file to augment its behaviour if one is present in the current directory. By default it will look for a file called `Holmakefile`, but it can be made to look at any file at all with the `--holmakefile` command-line option. The combination of `Holmake` and a make-file is supposed to behave as much as possible like a standard implementation of `make`.

A make-file consists of two types of entries, variable definitions and rules. Outside of these entries, white-space is insignificant, but newline and `TAB` characters are very significant within them. Comments can be started with hash (`#`) characters and last until the end of the line. Quoting is generally done with use of the back-slash (`\`) character. In particular, a backslash-newline pair always allows a line to be continued as if the newline wasn't present at all.

A variable definition is of the form

Ident `=` *text* `<NEWLINE>`

and a rule is of the form

text : *text* `<NEWLINE>`(`<TAB>`*text* `<NEWLINE>`)*

Henceforth, the text following a `TAB` character in a rule will be referred to as the *command text*. Text elsewhere will be referred to as *normal text*. Normal text has comments

stripped from it, so hash characters there must be escaped with a back-slash character. An *Ident* is any non-empty sequence of alpha-numeric characters, including the underscore (_).

In some contexts, normal text is interpreted as a list of words. These lists use white-space as element separators. If a word needs to include white-space itself, those white-space characters should be escaped with back-slashes.

Variable definitions The text on the RHS of a variable definition can be substituted into any other context by using a *variable reference*, of the form `$(VARNAME)`. References are evaluated *late*, not at time of definition, so it is quite permissible to have forward references. On the other hand, this makes it impossible to write things like

```
VAR = $(VAR) something_new
```

because the evaluation of `$(VAR)` would lead to an infinite loop. GNU make's facility for immediate definition of variables with `:=` is not supported.

Note also that white-space around the equals-sign in a variable definition is stripped. This means that

```
VAR =<whitespace><NEWLINE>
```

gives `VAR` the empty string as its value.[3]

Finally, note that the text inside a variable reference is itself evaluated. This means that one can write something like `$(FOO_$(OS))` and have this first expand the `OS` variable, presumably giving rise to some useful string (such as `unix`), and then have the resulting variable (`FOO_unix`, say) expanded. This effectively allows the construction of functions by cases (define variables `FOO_unix`, `FOO_macos` etc; then use the nested variable reference above). If the internal variable expands to something containing spaces, this will not turn a normal variable reference into a function call (see below). On the other hand, if the initial reference contains a space, the function name component *will* be expanded, allowing implementation of a function by cases determining which text-manipulation function should be called.

Rules Make-file rules are interpreted in the same way as by traditional make. The files specified after the colon (if any) are those files that each target (the files before the colon) is said to "depend" on. If any of these are newer than a target, then `Holmake` rebuilds that target according to the commands. If there are no dependencies, then

[3]It is possible to give a variable a value of pure whitespace by writing

```
NOTHING =
ONE_SPACE = $(NOTHING) $(NOTHING)
```

the commands are executed iff the target doesn't exist. If there are no commands, and the target is not of a type that Holmake already knows how to build, then it will just make sure that the dependencies are up to date (this may or may not create the target). If there are no commands attached to a rule, and the target is one that Holmake does know how to build, then the rule's extra dependencies are added to those that Holmake has managed to infer for itself, and Holmake will build the target using its built-in rule. If commands are provided for a type of file that Holmake knows how to build itself, then the make-file's commands and dependencies take precedence, and only they will be executed.

If a command-line is preceded by a hyphen (-) character, then the rest of the line is executed, but its error-code is ignored. (Normally, a command-line raising an error will cause Holmake to conclude that the target can not be built.) If a command-line is preceded by an at-sign (@), then that command-line will not be echoed to the screen when it is run. These two options can be combined in either order at the start of a command-line.

Command text is interpreted only minimally by Holmake. On Unix, back-slashes are not interpreted at all. On Windows, back-slashes followed by newlines are turned into spaces. Otherwise, command text is passed as is to the underlying command interpreter (/bin/sh say, on Unix, or COMMAND.COM on Windows). In particular, this means that hash-characters do *not* start comments on command-lines, and such "comments" will be passed to the shell, which may or may not treat them as comments when it sees them.

Functions Holmake supports some simple functions for manipulating text. All functions are written with the general form $(*function-name* arg_1,arg_2...$,$arg_n$)$. Arguments can not include commas (use variable references to variables whose value are commas instead), but can otherwise be arbitrary text.

$(dprot arg) quotes (or "protects") the space characters that occur in a string so that the string will be treated as a unit if it occurs in a rule's dependency list. For example, the file

```
dep = foo bar
target: $(dep)
   do_something
```

will see target as having two dependencies, not one, because spaces are used to delimit dependencies. If a dependency's name includes spaces, then this function can be used to quote them for Holmake's benefit. Note that the dprot function does *not* do the same thing as protect on either Unix or Windows systems.

`$(findstring arg1,arg2)` checks if `arg1` occurs in (is a sub-string of) `arg2`. If it does so occur, the result is `arg1`, otherwise the result is the empty string.

`$(if arg1,arg2,arg3)` examines `arg1`. If it is the empty string, then the value of the whole is equal to the value of `arg3`. Otherwise, the value is that of `arg2`.

`$(patsubst arg1,arg2,text)` splits `text` into component words, and then transforms each word by attempting to see if it matches the pattern in `arg1`. If so, it replaces that word with `arg2` (suitably instantiated). If not, the word is left alone. The modified words are then reassembled into a white-space separated list and returned as the value.

A pattern is any piece of text including no more than one occurrence of the percent (`%`) character. The percent character matches any non-empty string. All other characters must be matched literally. The instantiation for `%` is remembered when the replacement is constructed. Thus,

```
$(patsubst %.sml,%.uo,$(SMLFILES))
```

turns a list of files with suffixes `.sml` into the same list with the suffixes replaced with `.uo`.

`$(protect arg)` wraps `arg` in appropriate quote characters to ensure that it will pass through the operating system's command shell unscathed. This is important in the presence of file-names that include spaces or other shell-significant characters like less-than and greater-than. Those make-file variables that point directly at executables (`MOSMLC`, `MOSMLLEX` etc) are automatically protected in this way. Others, which might be used in concatenation with other elements, are not so protected. Thus, if `DIR` might include spaces, one should write

```
$(protect $(DIR)/subdirectory/program)
```

so that the above will be read as one unit by the underlying shell.

`$(subst arg1,arg2,text)` replaces every occurrence of `arg1` in `text` with `arg2`.

Special and pre-defined variables If defined, the `INCLUDES` variable is used to add directories to the list of directories consulted when files are compiled and linked. The effect is as if the directories specified had all been included on the command-line with `-I` options. The `PRE_INCLUDES` variable works similarly, but the directories specified here are placed before the `-I <holdir>` option that is used in invocations of compiler. This option gives the user a way of over-riding code in the core distribution as the compiler will find their code before the distribution's.

The OPTIONS variable is used for the specification of just four possible options (others are ignored): NO_SIGOBJ, NO_OVERLAY, NO_PREREQS and QUIT_ON_FAILURE. The OPTIONS variable should be a list of strings, containing some of the above four options. Those present are enabled. These have the same effect as the corresponding command-line options. The EXTRA_CLEANS variable is used to specify the name of additional files that should be deleted when a `Holmake clean` command is issued.

Within a command, the variable $< is used to stand for the name of the first dependency of the rule. The variable $@ is used to stand for the target of the rule.

Finally there are variables that expand to program names and other useful information:

CP This variable is replaced by an operating-system appropriate program to perform a file copy. The file to be copied is the first argument, the second is the place to copy to. The second argument can be a directory. (Under Unix, CP expands to `/bin/cp`; under Windows, it expands to `copy`.)

HOLDIR The root of the HOL installation.

HOLMOSMLC This variable is replaced by an invocation of the Moscow ML compiler along with the `-q` flag (necessary for handling quotations), and the usual `-I` include specifications (pre-includes, the hol-directory include, and the normal includes).

HOLMOSMLC-C This variable is the same as HOLMOSMLC except that it finishes with a closing `-c` option (hence the name) followed by the name of the system's overlay file. This is needed for compilation of HOL source files, but not for linking of HOL object code, which can be done with HOLMOSMLC.

ML_SYSNAME The name of the ML system being used: either `mosml` or `poly`.

MLLEX This is the path of the `mllex` tool that is built as part of HOL's configuration.

MLYACC This is the path of the `mlyacc` tool that is built as part of HOL's configuration.

MOSMLC This is replaced by an invocation of the compiler along with just the normal includes.

MOSMLLEX This is replaced by an invocation of the `mosmllex` program that comes with the Moscow ML distribution.

MOSMLYAC This is replaced by an invocation of the `mosmlyac` program that comes with the Moscow ML distribution.

MV This variable is replaced by an operating-system appropriate program to perform a file movement. The file to be moved is the first argument, the second is the place

to move to. The second argument can be a directory. (Under Unix, MV expands to mv; under Windows, it expands to rename.)

OS This variable is replaced by the name of the current operating system, which will be one of the strings "linux", "solaris", "macosx", "unix" (for all other Unices), or "winNT", for all Microsoft Windows operating systems (those of the 21st century, anyway).

SIGOBJ Effectively $(HOLDIR)/sigobj, where HOL object code is stored.

UNQUOTE The location of the quotation-filter executable.

The MOSMLLEX and MOSMLYAC abbreviations are really only useful if the originals aren't necessarily going to be on the user's "path". For backwards compatibility, the five variables above including the sub-string "MOSML" in their names can also be used by simply writing their names directly (i.e., without the enclosing $(...)), as long as these references occur first on a command-line.

If a reference is made to an otherwise undefined string, then it is treated as a reference to an environment variable. If there is no such variable in the environment, then the variable is silently given the empty string as its value.

Conditional Parts of Makefiles As in GNU make, parts of a Holmakefile can be included or excluded dynamically, depending on tests that can be performed on strings including variables. This is similar to the way directives such as #ifdef can be used to control the C preprocessor.

There are four possible directives in a Holmakefile: ifdef, ifndef, ifeq and ifneq. The versions including the extra 'n' character reverse the boolean sense of the test. Conditional directives can be chained together with else directives, and must be terminated by the endif command. The following example is a file that only has any content if the POLY variable is defined, which happens when Poly/ML is the underlying ML system.

```
ifdef POLY
TARGETS = target1 target2

target1: dependency1
      build_command -o target1 dependency1
endif
```

The next example includes chained else commands:

```
ifeq "$(HOLDIR)" "foo"
VAR = X
else ifneq "$(HOLDIR)" "bar"
VAR = Y
else
VAR = Z
endif
```

The `ifneq` and `ifeq` forms test for string equality. They can be passed their arguments as in the example, or delimited with apostrophes, or in parentheses with no delimiters, as in:

```
ifeq ($(HOLDIR),$(OTHERDIR))
VAR = value
endif
```

The definedness tests `ifdef` and `ifndef` test if a name has a non-null expansion in the current environment. This test is just of one level of expansion. In the following example, `VAR` is defined even though it ultimately expands to the emptry string, but `NULL` is not. The variable `FOOBAR` is also not defined.

```
NULL =
VAR = $(NULL)
```

Note that environment variables with non-empty values are also considered to be defined.

6.4 Generating and Using Heaps in Poly/ML HOL

Poly/ML has a nice facility whereby the state of one of its interactive sessions can be stored on disk and then reloaded. This allows for an efficient resumption of work in a known state. The HOL implementation uses this facility to implement the `hol` executable. In Poly/ML, `hol` starts immediately. In Moscow ML, `hol` starts up by visibly (and relatively slowly) "loading" the various source files that provide the system's functionality (e.g., `bossLib`).

Users can use the same basic technology to "dump" heaps of their own. Such heaps can be preloaded with source code implementing special-purpose reasoning facilities, and with various necessary background theories. This can make developing big mechanisations considerably more pleasant.

6.4.1 Generating HOL Heaps

The easiest way to generate a HOL heap is to use the `buildheap` executable that is built as part of the standard build process for (Poly/ML) HOL. This program takes a list of object files to include in a heap, an optional heap to build upon (use the `-b` command-line switch; the default is to use the heap behind the core `hol` executable), and an optional name for the new heap (the default is the traditional Unix `a.out`). Thus the command-line

```
buildheap -o realheap transcTheory polyTheory
```

```
ifdef POLY
HOLHEAP = realheap
OBJNAMES = polyTheory transcTheory
DEPS = $(patsubst %,$(dprot $(SIGOBJ)/%),$(OBJNAMES))

$(HOLHEAP): $(DEPS)
        $(protect $(HOLDIR)/bin/buildheap) -o $@ $(OBJNAMES)
endif
```

Figure 6.1: A `Holmakefile` fragment for building a custom HOL heap embodying the standard real number theories. If the heap's dependencies are not core HOL theories as they are here, then both the dependency line and the arguments to `buildheap` will need to be adjusted to link to the directory containing the files. For core HOL theories, the dependency has to mention the `SIGOBJ` directory, but when passing arguments to `buildheap`, that information doesn't need to be provided as `SIGOBJ` is always consulted by all HOL builds. Finally, note how the use of the `dprot` and `protect` functions will ensure that `Holmake` will do the right thing even when `HOLDIR` contains spaces.

would build a heap in the current directory called `realheap`, and would preload it with the standard theories of transcendental numbers and real-valued polynomials.

A reasonable way to manage the generation of heaps is to use a `Holmakefile`. For example, the `realheap` above might be generated with the source in Figure 6.1. The use of the special variable `HOLHEAP` has a number of nice side effects. First, it makes the given file a dependency of all other products in the current directory. This means that the HOL heap will be built first. Secondly, the other products in the current directory will be built on top of that heap, not the default heap behind `hol`.

6.4.2 Using HOL Heaps

As just described, if a `Holmakefile` specifies a `HOLHEAP`, then files in that directory will be built on top of that heap rather than the default. This is also true if the specified heap is in another directory (i.e., the `HOLHEAP` line might specify a file such as `otherdir/myheap`). In this case, the `Holmakefile` won't (shouldn't) include instructions on how to build that heap, but the advantages of that heap are still available. Again, that heap is also considered a dependency for all files in the current directory, so that they will be rebuilt if it is newer than they are.

It is obviously important to be able to use heaps interactively. If the standard `hol` executable is invoked in a directory where there is a `Holmakefile` specifying a heap, the default heap will not be used and the given heap will be used instead. The fact that this is happening is mentioned as the interactive session begins. For example:

```
------------------------------------------------------------------
       HOL-4 [Kananaskis 8 (stdknl, built Tue Jul 24 16:48:44 2012)]

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

[extending loadPath with Holmakefile INCLUDES variable]
[In non-standard heap: computability-heap]
Poly/ML 5.4.1 Release
>
```

Finally, note that heaps are required to be built first in a directory, and that heaps embody theories or ML sources that are *ancestral* to the directory in which the heap occurs. This has the unfortunate consequence that one cannot package up a heap embodying the standard theories for the real numbers in `src/real`, but that such a heap has to built in some other directory. This is counter-intuitive.

6.5 Timing and Counting Theorems

HOL can be made to record its use of primitive inferences, axioms, definitions and use of oracles. Such recording is enabled with the function

```
val counting_thms : bool -> unit
```

(This function as with all the others in this section is found in the `Count` structure.)

Calling `counting_thms true` enables counting, and `counting_thms false` disables it. The default is for counting to be disabled. If it is enabled, whenever HOL performs a primitive inference (or accepts an axiom or definition) a counter is incremented. A total count as well as counts per primitive inference are maintained. The value of this counter is returned by the function:

```
val thm_count : unit ->
 {ASSUME : int, REFL : int, BETA_CONV : int, SUBST : int,
  ABS : int, DISCH : int, MP : int, INST_TYPE : int, MK_COMB : int,
  AP_TERM : int, AP_THM : int, ALPHA : int, ETA_CONV : int,
  SYM : int, TRANS : int, EQ_MP : int, EQ_IMP_RULE : int,
  INST : int, SPEC : int, GEN : int, EXISTS : int, CHOOSE : int,
  CONJ : int, CONJUNCT1 : int, CONJUNCT2 : int, DISJ1 : int,
  DISJ2 : int, DISJ_CASES : int, NOT_INTRO : int, NOT_ELIM : int,
  CCONTR : int, GEN_ABS : int, definition : int, axiom : int,
  from_disk : int, oracle :int, total :int }
```

This counter can be reset with the function:

```
val reset_thm_count : unit -> unit
```

Finally, the `Count` structure also includes another function which easily enables the number of inferences performed by an ML procedure to be assessed:

```
val apply : ('a -> 'b) -> 'a -> 'b
```

An invocation, `Count.apply f x`, applies the function `f` to the argument `x` and performs a count of inferences during this time. This function also records the total time taken in the execution of the application.

For example, timing the action of `numLib`'s `ARITH_CONV`:

```
- Count.apply numLib.ARITH_CONV ``x > y ==> 2 * x > y``;                    2
runtime: 0.010s,    gctime: 0.000s,     systime: 0.000s.
Axioms asserted: 0.
Definitions made: 0.
Oracle invocations: 0.
Theorems loaded from disk: 0.
HOL primitive inference steps: 165.
Total: 165.
> val it = |- x > y ==> 2 * x > y = T : thm
```

6.6 Embedding HOL in LaTeX

When writing documents in LaTeX about one's favourite HOL development, one frequently wants to include pretty-printed terms, types and theorems from that development. Done manually, this will typically require involved use of the `alltt` environment, and cutting and pasting from a HOL session or theory file. The result is that one must also keep two copies of HOL texts synchronised: if the HOL development changes, the LaTeX document should change as well.

This manual, and error-prone process is not necessary: the standard HOL distribution comes with a tool called `munge.exe` to automate the process, and to remove the duplicate copies of HOL text. (Strictly speaking, the distribution comes with a tool that itself creates `munge.exe`; see Section 6.6.2 below.) The basic philosophy is that a LaTeX document can be written "as normal", but that three new LaTeX-like commands are available to the author.

The commands are not really processed by LaTeX: instead the source file must first be passed through the `munge.exe` filter. For example, one might write a document called `article.htex`. This document contains instances of the new commands, and cannot be processed as is by LaTeX. Instead one first runs

```
munge.exe < article.htex > article.tex
```

and then runs LaTeX on `article.tex`. One would probably automate this process with a makefile of course.

6.6.1 Munging Commands

Before Starting In order to use the munger, one must "include" (use the `\usepackage` command) the `holtexbasic.sty` style-file, which is found in the HOL source directory `src/TeX`.

There are then three commands for inserting text corresponding to HOL entities into LaTeX documents: `\HOLtm`, `\HOLty` and `\HOLthm`. Each takes one argument, specifying something of the corresponding HOL type. In addition, options can be specified in square brackets, just as would be done with a genuine LaTeX command. For example, one can write

```
\HOLtm[tt]{P(SUC n) /\ q}
```

and one will get

$$P\ \texttt{(SUC}\ n\texttt{)}\ \wedge\ q$$

or something very close to it, appearing in the resulting document.[4] Note how the spacing in the input (nothing between the `P` and the `SUC n`) is *not* reflected in the output; this is because the input is parsed and pretty-printed with HOL. This means that if the HOL input is malformed, the `munge.exe` program will report errors. Note also how the system knows that `P`, `n` and `q` are variables, and that `SUC` is not. This analysis would not be possible without having HOL actually parse and print the term itself.

The default behaviours of each command are as follows:

[4] The output is a mixture of typewriter font and math-mode characters embedded in a `\texttt` block within an `\mbox`.

\HOLty{*string*} Parses the string argument as a type (the input must include the leading colon), and prints it. The output is suited for inclusion in the normal flow of LaTeX (it is an \mbox).

\HOLtm{*string*} Parses the string argument as a term, and prints it. Again, the output is wrapped in an \mbox.

Important: If the string argument includes a right-brace character (i.e., the character }, which has ASCII code 125), then it must be escaped by preceding it with a backslash (\). Otherwise, the munger's lexer will incorrectly determine that the argument ends at that right-brace character rather than at a subsequent one.

\HOLthm{*thmspecifier*} The argument should be of the form ⟨*theory*⟩.⟨*theorem-name*⟩. For example, \HOLthm{bool.AND_CLAUSES}. This prints the specified theorem with a leading turnstile. However, as a special case, if the theorem specified is a "datatype theorem" (with a name of the form datatype_⟨*type-name*⟩), a BNF-style description of the given type (one that has been defined with Hol_datatype) will be printed. Datatype theorems with these names are automatically generated when Hol_datatype is run.

By default, the output is *not* wrapped in an \mbox, making it best suited for inclusion in an environment such as alltt. (The important characteristics of the alltt environment are that it respects layout in terms of newlines, while also allowing the insertion of LaTeX commands. The verbatim environment does the former, but not the latter.)

Munging Command Options There are a great many options for controlling the behaviour of each of these commands. Some apply to all three commands, others are specific to a subset. If multiple options are desired, they should be separated by commas. For example: \HOLthm[nosp,p/t,>>]{bool.AND_CLAUSES}.

alltt Makes the argument suitable for inclusion in an alltt environment. This is the default for \HOLthm.

case (Only for use with \HOLtm.) Causes the string to be parsed in such a way that any embedded case terms are only partly parsed, allowing their input form to appear when they are output. This preserves underscore-patterns, for example.

conj$n$ (Only for use with \HOLthm.) Extracts the n^{th} conjunct of a theorem. The conjuncts are numbered starting at 1, not 0. For example,

```
\HOLthm[conj3]{bool.AND_CLAUSES}
```

extracts the conjunct $\vdash$ F $\wedge\, t \iff$ F.

`def` (Only for use with `\HOLthm`.) Causes the theorem to be split into its constituent conjuncts, for each conjunct to have any outermost universal quantifiers removed, and for each to be printed on a line of its own. The turnstiles usually printed in front of theorems are also omitted. This works well with definitions (or characterising theorems) over multiple data type constructors, changing

$\vdash$ (FACT 0 = 1) $\wedge$ ($\forall n$. FACT (SUC n) = SUC n * FACT n)

into

FACT 0 = 1
FACT (SUC n) = SUC n * FACT n

`K` (Only for use with `\HOLtm`.) The argument must be the name of a theorem (as per the `\HOLthm` command), and the theorem should be of the form

$\vdash f\ x\ t$

for some term t. The command prints the term t. The expectation is that f will be the combinator `K` from `combin` (see Section 3.2.2), and that x will be truth (`T`), allowing t to be anything at all. In this way, large complicated terms that are not themselves theorems (or even of boolean type), can be stored in HOL theories, and then printed in LaTeX documents.

`merge`, `nomerge` (For use with `\HOLtm` and `\HOLthm`.) By default, the HOL pretty-printer is paranoid about token-merging, and will insert spaces between the tokens it emits to try to ensure that what is output can be read in again without error. This behaviour can be frustrating when getting one's LaTeX to look "just so", so it can be turned off with the `nomerge` option.

Additionally, this behaviour can be turned off globally with the `--nomergeanalysis` option to the munger. If this has been made the default, it may be useful to occasionally turn the merge analysis back on for a particular term or theorem; this is done with the `merge` option. (In interactive HOL, the token-merging analysis is controlled by a trace variable called `"pp_avoids_symbol_merges"`.)

`nodollarparens` (For use with `\HOLtm` and `\HOLthm`.) Causes the default escaping of syntactic sugar to be suppressed. The default behaviour is to use parentheses, so that

```
\HOLtm{$/\ p}
```

would get printed as $(\wedge)\ p$. Note that this doesn't reflect the default behaviour in the interactive loop, which is to use dollar-signs (as in the input above); see Section 5.1.2.1. However, with the `nodollarparens` option specified, nothing at all is printed to indicate that the special syntax has been "escaped".

`nosp` (Only for use with `\HOLthm`.) By default, arguments to `\HOLthm` are fully specialised (i.e., they have `SPEC_ALL` applied to them), removing outermost universal quantifiers. The `nosp` option prevents this.

`nostile` (Only for use with `\HOLthm`.) By default, arguments to `\HOLthm` are printed with a turnstile ($\vdash$). If this option is present, the turnstile is not printed (and the theorem will have its left margin three spaces further left).

`of` (Only for use with `\HOLty`.) The argument is a string that parses to a *term*, not a type. The behaviour is to print the type of this term. Thus `\HOLty[of]{p /\ q}` will print `bool`.

If the string includes right-braces, they must be escaped with back-slashes, just as with the arguments to `\HOLtm`.

`rule` (Only for use with `\HOLtm` and `\HOLthm`.) Prints a term (or a theorem's conclusion) using the `\infer` command (available as part of the `proof.sty` package). This gives a nice, "natural deduction" presentation. For example, the term

```
(p \/ q) /\ (p ==> r) /\ (q ==> r) ==> r
```

will print as

$$\frac{p \vee q \quad p \Rightarrow r \quad q \Rightarrow r}{r}$$

Conjuncts to the left of the outermost implication (if any) will be split into hypotheses separated by whitespace. For large rules, this style of presentation breaks down, as there may not be enough horizontal space on the page to fit in all the hypotheses. In this situation, the `stackedrule` option is appropriate.

The term or theorem must be within a LaTeX math-environment (it is typeset as if inline, with the `tt` option).

`showtypes` (For use with `\HOLthm` and `\HOLtm`.) Causes the term or theorem to be printed with the `types` trace set to level 1 (equivalent to having the `show_types` reference set to `true`).

`stackedrule` (For use with `\HOLthm` and `\HOLtm`.) This is similar to the `rule` option, but causes implication hypotheses to be presented as a "stack", centered in a LaTeX array on top of one another. Thus,

```
(p \/ q) /\ (p ==> r) /\ (q ==> r) ==> r
```

will print as

$$\frac{p \vee q \quad p \Rightarrow r \quad q \Rightarrow r}{r}$$

For this purely propositional example with single-letter variable names, the result looks a little odd, but if the hypotheses are textually larger, this option is indispensable.

`tt` Causes the term to be type-set as the argument to a LaTeX command `\HOLinline`. The default definition for `\HOLinline` is

```
\newcommand{\HOLinline}[1]{\mbox{\textup{\texttt{#1}}}}
```

This makes the argument suitable for inclusion in standard LaTeX positions. This is the default for `\HOLtm` and `\HOLty`. (The `\HOLinline` command is defined in the `holtexbasic.sty` style file.)

`width=`n Causes the argument to be typeset in lines of width n. The default width is 63, which seems to work well with 11pt fonts. This default can also be changed at the time the `munge.exe` command is run (see Section 6.6.3 below).

`>>` Indents the argument. This option only makes sense when used with the `alltt` option (the additional spaces will have no effect when inside an `\mbox`). The default indentation is two spaces; if a different indentation is desired, the option can be followed by digits specifying the number of space characters desired. For example, `\HOLthm[>>10,...]{...}` will indent by 10 spaces.

Note that simply placing a command such as `\HOLthm` within its `alltt` block with a given indentation, for example

```
\begin{alltt}
   \HOLthm{bool.AND_CLAUSES}
\end{alltt}
```

will not do the right thing if the output spans multiple lines. Rather the first line of HOL output will be indented, and the subsequent lines will not. The `>>` option lets the pretty-printer know that it is printing with a given indentation, affecting all lines of its output.

nm_1/nm_2 (For use with `\HOLtm` and `\HOLthm`.) Causes name nm_1 to be substituted for name nm_2 in the term or theorem. This will rename both free and bound variables, wherever they occur throughout a term. Because it uses instantiation, free variables in theorem hypotheses will get renamed, but bound variables in hypotheses are not affected. (Hypotheses are not printed by default anyway of course.)

If nm_1 and nm_2 both begin with the colon character then they are parsed as types, and type instantiation is performed on the term or theorem argument instead of variable substitution.

6.6.2 Creating a Munger

The HOL distribution comes with a tool called `mkmunge.exe`. This executable is used to create munge executables that behave as described in this section. A typical invocation of `mkmunge.exe` is

`mkmunge.exe` ⟨thy_1⟩`Theory ...` ⟨thy_n⟩`Theory`

Each commandline argument to `mkmunge.exe` is the name of a HOL object file, so in addition to theory files, one can also include special purpose SML such as `monadsyntax`.

The `mkmunge.exe` program can also take an optional `-o` argument that is used to specify the name of the output munger (the default is `munge.exe`). For example

```
mkmunge.exe -o bagtexprocess bagTheory
```

The theories specified as arguments to `mkmunge.exe` determine what theorems are in scope for calls to `\HOLthm`, and also determine the grammars that will govern the parsing and printing of the HOL types, terms and theorems.

6.6.3 Running a Munger

Once created, a munger can be run as a filter command, consuming its standard input, and writing to standard output. It may also write error messages and warnings to its standard error.

Thus, a standard pattern of use is something like

```
munge.exe < article.htex > article.tex
```

However, there are two ways of further modifying the behaviour of the munger, with command-line options.

Overrides Most importantly, one can specify an "overrides file" to provide token-to-LaTeX replacements of what is pretty-printed. The command-line would then look like

```
munge.exe overrides_file < article.htex > article.tex
```

The overrides file is a text file containing lines of the form

```
tok width tex
```

where `tok` is a HOL token, `width` is a number giving the width of the LaTeX, and `tex` is a LaTeX string.

As a very simple example, an overrides file might consist of just one line:

```
pi1 2 \ensuremath{\pi_1}
```

This would cause the string `pi1` (presumably occurring in the various HOL entities as a variable name) to be replaced with the rather prettier π_1. The `2` records the fact that the printer should record the provided LaTeX as being 2 characters wide. This is important for the generation of reasonable line-breaks.

Overrides for HOL tokens can also be provided within HOL theories, using the `TeX_notation` command (see Section 6.6.5 below).

Default width A munger can specify the default width in which HOL will print its output with a `-w` option. For example,

```
munge.exe -w70 < article.htex > article.tex
```

This default width can be overridden on a case-by-case basis with the `width=` option to any of the commands within a LaTeX document.

Preventing Merge Analysis As mentioned above in the description of the `merge` and `nomerge` options to the `\HOLtm` and `\HOLthm` commands, the munger can be configured to not do token-merging avoidance by passing the `--nomergeanalysis` option to the munger.

The `-w`, `--nomergeanalysis` and overrides file options can be given in any order.

6.6.4 Holindex

Till now, it has been explained how the munger can be used as a preprocessor of LaTeX sources. Sometimes a tighter interaction with LaTeX is beneficial. Holindex is a LaTeX package that provides genuine LaTeX commands for inserting HOL-theorems, types and terms as well as many related commands. This allows it to generate an index of all HOL-theorems, types and terms that occur in the document as well as providing citation commands for HOL entities in this index. Holindex can be found in `src/TeX/`. There is also a demonstration file available in this directory.

Using Holindex To use Holindex add `\usepackage{holindex}` to the header of the LaTeX source file `article.tex`. Holindex loads the `underscore` package which might cause trouble with references and citations. In order to avoid problems, `holindex` should be included after packages like `natbib`. Holindex is used like BibTex or MakeIndex. A run of LaTeX on `jobname.tex` creates an auxiliary file called `article.hix`. The munger is used to process this file via

```
munge.exe -index article
```

This call generates two additional auxiliary files, `article.tde` and `article.tid`. The following runs of LaTeX use these files. After modifying the source file, the munger can be rerun to update `article.tde` and `article.tid`. If you are using emacs with AUCTeX to write your latex files, you might want to add

```
(eval-after-load "tex" '(add-to-list 'TeX-command-list
   '("Holindex" "munge.exe -index %s"
     TeX-run-background t t :help "Run Holindex") t))
```

to your emacs configuration file. This will allow you to run Holindex using AUCTeX.

Holindex commands

`\blockHOLthm{id}`, `\blockHOLtm{id}`, `\blockHOLty{id}` These commands typeset the theorem, term or type with the given `id` as the argument to a LaTeX command `\HOLblock`. They are intended for typesetting multiple lines in a new block. For theorem ids of the form `theory.thm` are predefined. All other ids have to be defined before usage as explained below.

`\inlineHOLthm{id}`, `\inlineHOLtm{id}`, `\inlineHOLty{id}` These commands are similar to `\blockHOLthm{id}`, `\blockHOLtm{id}` and `\blockHOLty{id}`. However, they are intended for inline typesetting and use `\HOLinline` instead of `\HOLblock`.

`\citeHOLthm{id}`, `\citeHOLtm{id}`, `\citeHOLty{id}` These commands cite a theorem, term or type.

`\mciteHOLthm{id,id,...id}`, `\mciteHOLtm{ids}`, `\mciteHOLty{ids}` These commands cite multiple theorems, terms or types.

`\citePureHOLthm{id}`, `\citePureHOLtm{id}`, `\citePureHOLty{id}` These commands cite a theorems, terms or types. They just typeset the number instead of the verbose form used by the `citeHOL` and `mciteHOL` commands.

`\citeHiddenHOLthm{id}`, `\citeHiddenHOLtm{id}`, `\citeHiddenHOLty{id}` These commands cite a theorems, terms or types, but not typeset anything. These commands can be used to add a page to the list of pages a theorem, term or type is cited.

`\printHOLIndex`, `\printHOLShortIndex`, `\printHOLLongIndex` These commands typeset the index of all theorems, terms and types cited in the document. There are two types of entries in the index: long and short ones. Short entries contain a unique number, the label of the theorem, term or type and the pages it is cited. Long entries contain additionally a representation as it would be inserted by `\blockHOL...` as well as an optional description. Theorems use by default short entries, while terms and types use long ones. It is possible to change for each item whether a long or short entry should be used. `\printHOLIndex` prints the default index with mixed long and short entries. `\printHOLLongIndex` typesets just long entries and `\printHOLShortIndex` just short ones.

Defining and formating Terms, Types and Theorems Most of the Holindex commands require an identifier of a theorem, term or type as arguments. Theorem identifiers of the form `theory.theorem` are predefined. All other identifiers need defining. Additionally one might want to change the default formating options for these new identifiers as well as the old ones. HOL definition files can be used for defining and setting the formating options of identifiers. They are used by putting the command `\useHOLfile{`*`filename.hdf`*`}` in the header of your latex source file. These file use a syntax similar to BibTex. They consist of a list of entries of the form

```
@EntryType{id,
  option = value,
  boolFlag,
  ...
}
```

There are the following entry types

`Thm`, `Theorem` used to define and format a theorem. If the identifier is of the form `theory.theorem`, the `content` option can be skipped. Otherwise, the `content` option should be of this form and a new identifier is defined for the given theorem. This is for example useful if the theorem name contains special characters or if a theorem should be printed with different formatting options.

`Term` used to define and format a term.

`Type` used to define and format a type.

`Thms`, `Theorems` used to set formating options for a list of theorems. For example one might want to print long index entries for all theorems in a specific theory. For the `Theorems` entry the `id` part of the entry is given in the form `ids = [id,id,...]`. These `ids` may be theorem ids or special ids of the form `theorem.thmprefix*`. The

id `arithmetic.LESS_EQ*` for example represents all theorem in theory `arithmetic` whose name starts with `LESS_EQ`.

Options are name/value pairs. The value has to be quoted using quotation marks or HOL's quotation syntax. There are the following option names available:

`content` the content. For a term or type that's it's HOL definition. For theorems it is of the form `theory.theorem`.

`options` formating options for the munger as described in section 6.6.1. Please use the Holindex commands for typsetting inline or as a block instead of the options `tt` or `alltt`.

`label` the label that will appear in the index. For theorems the label is by default it's name and the label given here will be added after the name.

`comment` latex code that gets typeset as a comment / description for long index entries.

`latex` the latex code for the item. There are very rare cases, when it might be useful to provide handwritten LaTeX code instead of the one generated by the munger. This option overrides the LaTeX produced by the munger. It is recommended to use it very carefully.

Besides options, there are also boolean flags that change the formating of entries:

`force-index` adds the entry to the index, even if it is not cited in the document.

`long-index` use a long index-entry.

`short-index` use a long index-entry.

Here is an example of such a HOL definition file:

```
@Term{term_id_1,
   content = ``SOME_FUN = SUC a < 0 /\ 0 > SUC b``,
   options = "width=20",
   label = "a short description of term from external file",
   comment = "some lengthy\\comment

              with \textbf{formats} and newlines",
   force_index
}

@Type{type_id_1,
   content = ``:bool``
```

```
}

@Thm{arithmetic.LESS_SUCC_EQ_COR,
   force-index, long-index
}

@Thm{thm_1,
   label = "(second instance)",
   content = "arithmetic.LESS_SUC_EQ_COR"
}

@Theorems{
   ids = [arithmetic.LESS_ADD_SUC,
          arithmetic.LESS_EQ*],
   force-index
}
```

Configuring Holindex There are some commands that can be used to change the overall behaviour of Holindex. They should be used in the header directly after `holindex` is included.

`\setHOLlinewidth` sets the default line-width. This corresponds to the `-w` option of the munger.

`\setHOLoverrides` sets the "overrides file" to provide token-to-LaTeX replacements of what is pretty-printed.

`\useHOLfile` is used to include a HOL definition file. Several such files might be included.

Additional Documentation For more information about Holindex, please refer to the demonstration file `src/TeX/holindex-demo.tex`. This file contains documentation for rarely used commands as well as explanations of how to customise Holindex.

6.6.5 Making HOL Theories LaTeX-ready

Though one might specify all one's desired token-replacements in an `overrides` file, there is also support for specifying token replacements in the theory where tokens are first "defined". (Of course, *tokens* aren't defined *per se*, but the definition of particular constants will naturally give rise to the generation of corresponding tokens when those constants appear in HOL terms, types or theorems.)

A token's printing form is given in a script-file with the `TeX_notation` command (from the `TexTokenMap` module). This function has type

```
{ hol : string, TeX : string * int } -> unit
```

The `hol` field specifies the string of the token as HOL prints it. The `TeX` field specifies both the string that should be emitted into the LaTeX output, and the width that this string should be considered to have (as in the `overrides` file).

For example, in `boolScript.sml`, there are calls:

```
val _ = TeX_notation { hol = "!", TeX = ("\\HOLTokenForall{}", 1)}
val _ = TeX_notation { hol = UChar.forall,
                       TeX = ("\\HOLTokenForall{}", 1)}
```

The `UChar` structure is a local binding in the script-file that points at the standard list of UTF8-encoded Unicode strings in the distribution (`UnicodeChars`). Note also how the backslashes that are necessary for the LaTeX command have to be doubled because they are appearing in an SML string.

Finally, rather than mapping the token directly to the string `\forall` as one might expect, the mapping introduces another level of indirection by mapping to `\HOLTokenForall`. Bindings for this, and a number of other LaTeX commands are made in the file

```
src/TeX/holtexbasic.sty
```

which will need to be included in the LaTeX source file. (Such bindings can be overridden with the use of the command `\renewcommand`.)

Finally, all theory-bindings made with `TeX_notation` can be overridden with `overrides` files referenced at the time a munger is run.

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] Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–68, 1940.

[3] F.K. Hanna and N. Daeche. Specification and verification using higher-order logic: A case study. In G. Milne and P.A. Subrahmanyam, editors, *Formal Aspects of VLSI Design: Proceedings of the 1985 Edinburgh Workshop on VLSI*, pages 179–213. North-Holland, 1986.

[4] J. Harrison. *Theorem-proving with the Real Numbers*. CPHC/BCS Distinguished Dissertations. Springer, 1998.

[5] Joe Hurd. *Formal Verification of Probabilistic Algorithms*. PhD thesis, University of Cambridge, 2002.

[6] A. Leisenring. *Mathematical Logic and Hilbert's ϵ-Symbol*. University Mathematical Series. Macdonald & Co. Ltd., London, 1969.

[7] R. Milner M. Gordon and C. P. Wadsworth. *Edinburgh LCF: A Mechanised Logic of Computation*, volume 78 of *Lecture Notes in Computer Science*. Springer-Verlag, 1979.

[8] T.F. Melham. Automating recursive type definitions in higher order logic. In G. Birtwistle and P.A. Subrahmanyam, editors, *Current Trends in Hardware Verification and Automated Theorem Proving*, pages 341–386. Springer-Verlag, 1989.

[9] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[10] Konrad Slind. *Reasoning about Termination Functional Programs*. PhD thesis, Technical University of Munich, 1999.

Index